Learning The World’s Hardest Game

Yasyf Mohamedali, yasyf@mit.edu

I. Introduction

The World’s Hardest Game [6] (TWHG) is a widely-available online Flash game known for its infuriatingly difficult gameplay. In the game, a player selects a move from a discrete set (up, down, left, right, stay) to make at each time step, with the goal of navigating through a series of levels. In order to complete a level, the player must collect all on-screen coins, avoid collision with any enemies, and reach the area designated as the end zone. While the game is deterministic (modulo some random initialization of the enemies), it is a very difficult task for humans to complete, requiring precision, patience, and often a difficult-to-discover strategy for movement. However, the set of actions to be taken at any given time step is finite, and the reward for an action is easily calculated. Thus, the game is a good candidate for a reinforcement learning-based model.

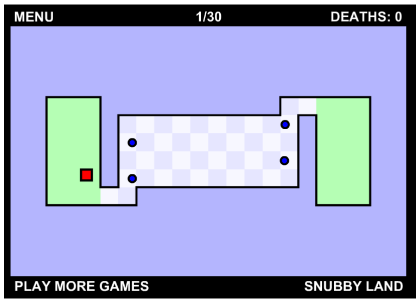

Fig. 1: Level 1 of TWHG

Deep learning, combined with classic reinforcement learning techniques, allows one to learn a action-selection policy on complicated, long games that have significant delays between action and reward. Furthermore, a CNN is capable of taking pixel inputs from frames of a game simulator and learning a representation of the game state from them. Thus, it is theoretically possible to have a game-winning strategy learned simply by “playing” a game over and over, and observing the results.

The original goal of this paper was to use the Deep Q-learning with Experience Replay technique pioneered by DeepMind [8] to train a model that was capable of beating TWHG, surpassing the performance of even the most skilled human player. After weeks of training, it became clear that achieving this is beyond the scope of the paper, with the best models barely managing to navigate the player out of the starting area. Therefore, we instead explored several related “toy” games (all with a similar premise), testing the techniques and hyperparameters laid out in the DeepMind paper against two baselines: a random model, and a ϵ-greedy model.

We tested our architecture and model against increasingly complicated toy games, attempting learning both with pixel inputs as well as by directly feeding in a representative state vector. The results show that training with image inputs takes prohibitively long, where as directly feeding in state vectors results in a model that outperforms both the baselines within days. Furthermore, the learned models take actions that are intuitive to a human player, with no seeding required.

All the referenced implementation code for this paper can be found online at https://github.com/yasyf/hardest-game.

II. Background

A. Reinforcement Learning

Q-learning, in which an agent optimizes a game-specific *reward function* by taking an *action* which maximizes the discounted sum of future rewards (*Q values*), is a popular reinforcement learning technique for interacting with an *environment* and set of actions that can be modeled as a finite Markov decision process (MDP). The solution can be discovered via a simple value-iteration update algorithm [11].

At its heart, the process works as follows. At each time step, the actor selects an action from $\mathcal{A}$, the space of actions available in the given environment ε. This action is passed to a simulator (implemented in our case for each game learned), which modifies ε, without revealing its internal state. The agent then observes state vector representing the ε, as well as a reward $r_t \in \mathbb{R}$. This sequence of actions, observations, and rewards is then used to solve the MDP they represent. Future rewards are discounted by a rate γ.

B. Deep Reinforcement Learning

Advances in deep learning have brought about the advent of Deep Q-learning, which relies on the same underlying principles as Q-learning, but replaces the central lookup table with a neural net, allowing one to learn a function approximator to the Q function in a much larger and sparser action space, without exploring every possible set of moves (which can often be intractable for more complicated games). Furthermore, convolutional layers of the net allow the model to learn directly from game frames (or in our case, several stacked frames), decreasing the amount of structured information that needs to be supplied.

Gradient descent techniques were used to optimize the parameters of the network. The architecture was a CNN which took $84 \times 84 \times 4$ vectors, and contained two hidden layers: 16 8×8 filters of stride 4 and 32 4×4 filters of stride 2. This was followed by a 256-node fully-connected hidden layer of ReLU, and an output layer with $|\mathcal{A}|$ linear nodes. This will become the starting point of the models described in this paper.

C. Simulators

As a Flash-based game, TWHG is easy to simulate. Using headless browser wrappers such as Selenium, it is possible to launch instances of Google Chrome and load the SWF containing the game’s code without user intervention. Simulating user moves and extracting state is done through the bare-bones JavaScript API exposed by the Flash runtime [1]. Internal state can be read using the `TGetProperty` and

`GetVariable` methods, and written using the corresponding `Set*` methods. Part of the contribution of this paper is a generic Python-based simulator [10] which can interact with Flash-based games and report observable state, as well as capture frames for later processing.

D. *Game Policies*

Techniques similar to the above (albeit with a fully-fledged console emulator) were successfully used to play Atari 2600 games at an expert level by the DeepMind team [8] in 2013. Previously, attempts were made by Brodman and Voldstad [2] to play games like QWOP [4] by modeling the game in detail, and extracting precise state vectors, with classic RL techniques. Another recent application of RL (though not Deep RL) to games is the use of Q-learning to play Super Mario with a 90% win rate[7]. We could not find any published examples of using Deep Q-learning on a simulated Flash game, though recently-launched platforms like the OpenAI Universe (https://universe.openai.com) open the gateway for many similar attempts.

III. Methods

A. *Toy Games*

In addition to TWHG, described above, we implemented several simple games of varying difficulty, in order to test the efficacy of the techniques and models presented in this paper within a limited span of time.

1) TG1D: The first game (TG1D) is a very simple game, involving a 1-dimensional 4-square grid that has one enemy randomly placed on one of the middle two squares. The enemy e appears and disappears with a random phase, and the goal of the player p is to move from the first square to the last, without ever being in the same square as the in-phase enemy. The possible moves for the player are $\{\texttt{left}, \texttt{right}, \texttt{stay}\}$. The observable state vector S for this game is as follows.

$$S_x = \begin{cases} -2 & \text{if } e = x \text{ and } p = x \\ -1 & \text{if } e = x \\ 1 & \text{if } p = x \\ 0 & \text{else} \end{cases}$$

2) TG2D: TG2D is the natural 2-dimensional progression of this simple game. The game is played on a 4×4 grid, with the starting location for p at $(0, 0)$ and the goal location at $(3, 3)$. A single enemy occupies a random square selected from those that are not the start or goal, with a random phase. The objective is the same. The possible moves for the player are $\{\texttt{left}, \texttt{right}, \texttt{down}, \texttt{up}, \texttt{stay}\}$. S for this game follows as expected from the 1-dimensional case, though it is now a matrix (which is flattened to produce the final state vector).

3) TG2D-H: TG2D-H is the exact same game as TG2D, but with multiple enemies. Each enemy is sampled from the squares remaining after removing the start, goal, and any square that currently has an enemy assigned to it.

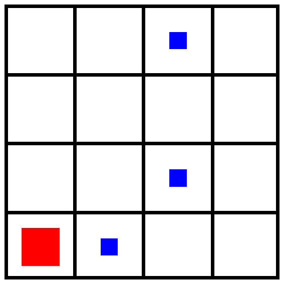
Fig. 2: TG2D-H

B. *Learning*

To implement the neural net architecture for this model, we used the popular TensorFlow library. Simple abstractions were build around 2D-convolutional and fully-connected layers, which were strung together to test various architectures. At times, reference was made to various existing implementation of Deep Q-learning in TensorFlow [5] [3].

The output of the net represented the Q-estimate for each action, with the best action defined as $\arg\max_{a \in \mathcal{A}} Q(t, a)$. The loss function used for optimization is the mean-squared-error of the Q-estimate predicted for a given action a at time t, and the "realized" Q-value, which is the known reward for a at t added to the discounted Q-estimate of the best action at the state $t + 1$, given a was made at t. Below we give the formula for the loss of a minibatch $\mathcal{B}$, sampled from the replay memory $\mathcal{M}$.

$$\ell = \frac{1}{|\mathcal{B}|} \sum_{t \in \mathcal{B}} \left[\texttt{clip} \left(Q\left(t, a_t\right) - \left[r_{t,a} + \gamma \max_{a \in \mathcal{A}} Q\left(t+1, a\right) \right] \right) \right]^2$$

The difference in Q-estimate for t and "realized" Q-value is clipped to be in the range $[\delta_-, \delta_+]$, as recommended by the DeepMind paper, in order to limit the scale of the error gradient. Several values where tried for the bounds of this range. Finally, the Adam optimizer was used to optimize this loss function given the game samples.

Samples were provided using the Experience Replay technique described in the DeepMind paper, with an ϵ-greedy policy for selecting optimal moves (with ϵ controlling how often random moves were made, in order to force exploration, as opposed to the move suggested by the current Q-estimates). The replay memory of each episode (from which minibatch training points are sampled) is first seeded by filling the history $\mathcal{H}$ (which holds the stacked inputs to the model for the given timestep) with $|\mathcal{H}|$ frames of the player making the `stay` move. This has the nice property of exposing the model to the motion patterns of enemies, which aids in predicting where enemies will appear when the player starts moving.

When feeding in frame images, the architecture used was exactly as described in the DeepMind paper [8] (CNN), with either 3 or 5 output nodes, depending on the game. When feeding in state vectors, we instead use 2 fully-connected (FC) hidden layers, one with 128 and one with 64 ReLU nodes. Some alternate architectures were experimented with; further work could include testing out more, particularly in the non-image case. Figure 3 shows the exponentially-weighted moving average of the win rate during training for two architectures: $[128, 64]$ (orange) and $[256, 256]$ (purple).

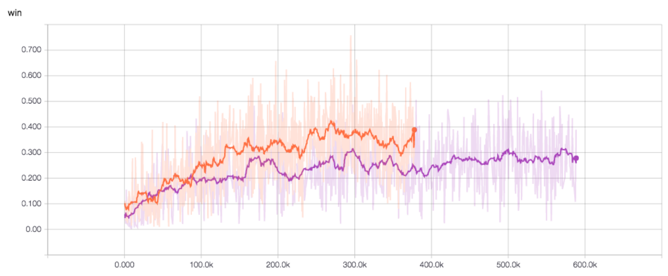

Fig. 3: Training Win Rates of TG2D-H

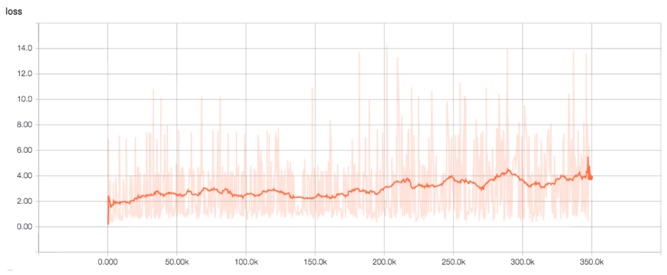

Fig. 4: Loss Over Time For TG2D with $|\mathcal{H}| = 3$

C. *Hyperparameters*

Most of the hyperparameters in the system were copied from the DeepMind paper, then modified and tuned as appropriate for this set of games. Where the value of the hyperparameters

A γ value of 0.90 was used initially, which sufficiently penalizes future rewards in order to discourage the player from getting "stuck", which occurs when all non-`stay` actions have negative Q-values (common near the start of a training session). We experimented increasing this to 0.99, expecting that increasing the value of future rewards would make the resulting player more risk-averse. Indeed, this was confirmed, with the resulting model often causing the player to `stay` ad infinitum.

ϵ, the parameter which controls how often a random action is selected when the model is exploring, starts at 0.9 and is linearly annealed to 0.1, over a lifetime of 10^5 frames. This was originally much lower, but was increased to closer match DeepMind's 10^6, as lower values would allow the model to converge on local solutions that were sub-optimal due to a lack of sufficient exploring. Thus we see that some level of stochasticity is required in order to sufficiently uncover the state space.

A minibatch size of 32 was used. This was sampled from a replay memory size of 10^5, an order of magnitude less than the DeepMind paper. While 10^6 was initially used, using a lower value allowed us a discernible increase in training time and performance, with negligible impact on the results.

The difference between predicted and expected Q-values was initially not clipped, resulting in blowup of the error gradient and huge sensitivity to slight changes in the reward function. As a result, we initially clipped the values to be in $[-1, 1]$, as in the DeepMind paper. However, this did not allow for enough variance to accurately reflect the games at hand, so this was ultimately changed to be $[-10, 10]$.

Determining $|\mathcal{H}|$, the number of stacked images to feed into the CNN, required some experimentation. For the smaller, simpler games, a history size of 2 was optimal. As the games grow more complex, we settle on $|\mathcal{H}| = 3$ (an average loss of 1.36 vs 2.11 after 150k frames).

A final decision to be made was whether to feed in pixel matrices directly (thereby using the CNN architecture), or to extract state vectors and use the FC architecture. As is shown in the results, using CNN allows us to encode much less information about the game when learning the model, but has a prohibitively long training period. Thus we ultimately report our best results with state vector inputs.

IV. Results

Following the tuning of the above hyperparameters, we ran our best models, as well as two baselines, on each of the three toy games. We also attempted to train a model on TWHG. However, the training time was prohibitive, on account of the Flash simulation taking three to four orders of magnitude longer to modify ε than the simulators we built for the toy games. Thus, after over a week of training, we did not have any significant results to report.

The win rates reported below are averaged over 10 runs, each containing 1000 games, for a total of 10000 samples.

A. *Baselines*

1) Random: We implemented a basic random model as a baseline. The model samples an action from a uniform distribution over $\mathcal{A}$.

Game	Win Rate (%)
TG1D	28.85
TG2D	36.95
TG2D-H	9.18

2) ϵ-Greedy: Our next baseline was a ϵ-greedy model that sampled a uniformly random action with probability ϵ, and took the greedy action at $t+1$ based on the rewards at t with probability $1-\epsilon$. For the below values, we used $\epsilon = 0.05$.

Game	Win Rate (%)
TG1D	92.06
TG2D	77.76
TG2D-H	47.98

B. *Deep Model*

We ran the best model over several training runs for each game, and present the results below. For any game more complex than the basic 1-D toy game, we see that our Deep Q-learning model outperforms the naive ϵ-greedy model.

For the simple TG1D, we hypothesize that such a simple game can effectively be mastered with a greedy strategy, and that our function approximator could not match these results given the limited training time.

V. Discussion

An interesting note is that the best-performing deep model for TG1D simply learned to always move `right` unless there is an

Game	Win Rate (%)	# Training Frames
TG1D	71.28	800k
TG2D	88.91	900k
TG2D-H	54.81	1.5M

enemy present, in which case it `stays` for one turn, then continues to move right. This is almost the correct (greedy) strategy, but does not take into account the case of enemies which are on screen for more than one turn.

Focusing on the more complicated (toy) games, we can make some revealing observations. We see that as our model improves, the average Q-value estimate improves over time. This gives us a good indicator into the model's increasing confidence in its actions, and is what we used as input to the stopping condition for training (though a stopping point was never reached).

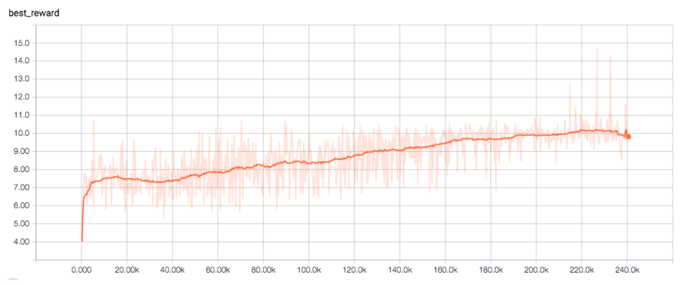

Fig. 5: Max Q-value Estimate Over Time For TG2D

We see an fascinating development in the evolution of the distribution of optimal actions over time. As expected, given the starting and goal coordinates of the player in the toy games, the `down` and `right` actions often dominate the distribution, with the model learning that there is limited utility to moving `up` or `left` (save to avoid an enemy). However, though the distribution is skewed over time, the expected reward of each action remains roughly equal, indicating that the model can tell that a wasted move is not necessarily the end of the world. Proving this is the result that tweaking the reward function to penalize staying still, along with increasing the discount on future rewards, results in the Q-value estimates for the actions `up` and `left` to drop.

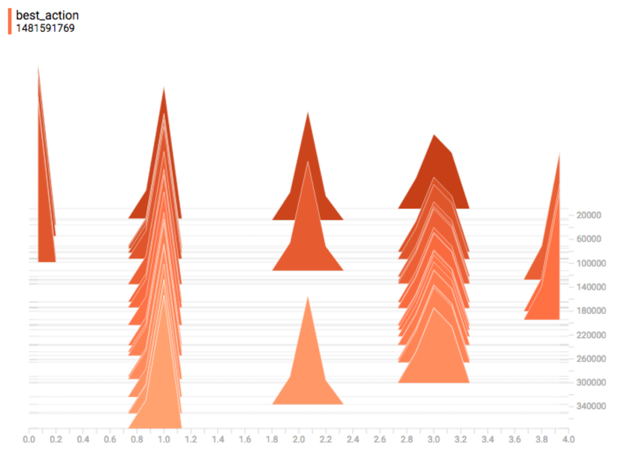

Fig. 6: Time over Distribution of Best Action for TG2D-H

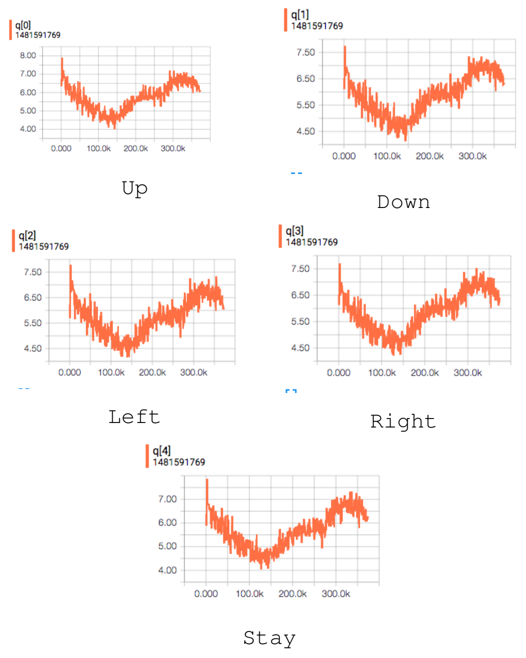

Fig. 7: Per-Action Q-value Estimates Over Time For TG2D-H

A. Challenges

There were many difficult challenges in implementing the simulation and modeling code discussed above. One unique challenge involved deciphering which “registers” the Flash runtime assigned to various game state values, since the JavaScript API only accepts register numbers when fetching variables.

Another significant and obvious challenge was the time involved in simulating a Flash game. Between browser overhead and the inability to speed up framerate, we were met with significant obstacles that prevented us from successfully training a model on our target game. While we were able to mitigate this obstacle by creating a multi-threaded pool of simulators [9] for the model training to use, this was not a sufficient speedup.

A final design challenge faced by the authors was how much human intuition to engineer into the reward function. While models performed better with highly-tuned rewards, this sometimes prevented the necessary exploration for finding a super-human strategy. This tradeoff is something that could be explored more in future work.

VI. Conclusion

In this paper, we set up a model for evaluating a Deep Q-learning technique on games that resemble The World's Hardest Game. Though we ultimately were unable to train a model on the Flash game itself, we were able to achieve win rates that exceeded our baseline on increasingly complicated replica games. We were not able to beat a human player in any these games, and struggled to train sufficient models given highly constrained training times.

References

[1] The Closure Compiler Authors. *flash.js*. URL: https://developer.pubref.org/static/apidoc/global/externs/flash.js.html.

[2] Gustav Brodman and Ryan Voldstad. “Qwop Learning”. In: 2012.

[3] DEVSISTERS. *DQN-tensorflow*. URL: https://github.com/devsisters/DQN-tensorflow.

[4] Foddy.net. *QWOP*. 2015. URL: https://www.foddy.net/Athletics.html.

[5] Taehoon Kim. *deep-rl-tensorflow*. URL: https://github.com/carpedm20/deep-rl-tensorflow.

[6] Snubby Land. *World’s Hardest Game*. URL: http://www.worldshardestgame.org/.

[7] Yizheng Liao, Kun Yi, and Zhe Yang. “Cs229 Final Report Reinforcement Learning to Play Mario”. In: 2012.

[8] Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: *CoRR* abs/1312.5602 (2013). URL: http://arxiv.org/abs/1312.5602.

[9] Yasyf Mohamedali. *simulator_pool.py*. URL: https://github.com/yasyf/hardest-game/blob/master/hardest_game/shared/simulator_pool.py.

[10] Yasyf Mohamedali. *simulator.py*. URL: https://github.com/yasyf/hardest-game/blob/master/hardest_game/hardest_game/simulator.py.

[11] Christopher J.C.H. Watkins and Peter Dayan. “Technical Note: Q-Learning”. In: *Machine Learning* 8.3 (1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1023/A:1022676722315. URL: http://dx.doi.org/10.1023/A:1022676722315.