Learning to play SLITHER.IO with deep reinforcement learning

CS229 project report. Category: Theory & Reinforcement Learning.

Joan Creus-Costa
<jcreus@stanford.edu>

Zhanpei Fang
<zhanpei@stanford.edu>

Abstract

This project uses deep reinforcement learning to train an agent to play the massively multiplayer online game SLITHER.IO. We collect raw image inputs from sample gameplay via an OpenAI Universe environment as training data. We implement deep Q-learning with Huber loss, incorporating human demonstrations, reward shaping and prioritized replay in order to improve stability and successfully learn a policy. Our best model performs significantly better than a random policy, but worse than a human benchmark, achieving a median difference in score at the end of each episode of 54.

1 Introduction

Games have long been a benchmark of reinforcement learning (RL), beginning with the 1990s breakthrough in backgammon [Tesauro, 1995] and evolving to video games with DeepMind's pioneering work in deep reinforcement learning [Mnih et al., 2013, 2015]. Various papers have shown state-of-the-art performance in Atari games, as well as in more complex games such as Go. Video games are challenging—and therefore interesting—systems due to their high-dimensional visual input, high frame rate, and, in some instances (such as OpenAI's work on *Dota 2*) their multiplayer nature.

SLITHER.IO is a massively multiplayer online (MMO) game in which players use a mouse to control a "snake" on a 2D map. The objective of the game is to maximize snake length by consuming multicolored food pellets while avoiding other players' snakes; if the player's snake's head collides with the body of another snake, the player will die and their body disintegrates into pellets for others to consume.

The goal of this project is to achieve human-like performance using reinforcement learning on SLITHER.IO, which presents a good target due to the relative simplicity of its game mechanics, along with the possibility for complex emergent behavior[1]. The massively-multiplayer nature of the game adds non-determinism and provides a clear benchmark with which to assess performance, relative to that of human players.

2 Related work

Reinforcement learning has been applied with great success to a wide variety of gameplay scenarios. RL is well-suited to solving the problem of how an agent can learn to make decisions and achieve goals in a complex, uncertain environment without too many prior assumptions about the system.

Deep Q-learning has in particular performed well in the video-game environment, as it is suited to problems with high-dimensional observation spaces and discrete and low-dimensional action spaces [Lillicrap et al., 2015]. The seminal work of Mnih et al. [2013] used deep Q-learning to play classic Atari 2600 games, surpassing the performance of all previous algorithms and achieving performance comparable to that of a professional games

[1]For examples of some of these complex behaviors, see this list of player-devised strategies: `https://slitherio.gamepedia.com/Strategies`.

Preprint. Under review.

tester across all 49 games. This *tour de force* re-energized reinforcement learning with novel applications of deep neural networks, and many variations have emerged to address its shortcomings, such as Double Q-learning, which corrects the tendency of the original algorithm to overestimate Q function values [Van Hasselt et al., 2016]. A variety of policy gradient methods, such as DDPG [Lillicrap et al., 2015] have proven to be very successful in continuous and stochastic environments.

SLITHER.IO is part of OpenAI Gym's[2] *Universe*,[3] a collection of environments with a wide variety of reinforcement learning tasks, primarily from retro video games or online games. As a result of this convenient packaging, previous work has tried to use various machine-learning methods to learn to play this game, though it has not become a standard benchmark like some other Gym environments, due to the inherent difficulties associated with a multiplayer Internet-based environment. Caudill [2017] combines hand-coded pre-processing of frames and deep Q-learning to achieve limited success in the early game, concluding that more training was required due to the game's complexity. Miller et al. [2019] use neuroevolution methods to achieve higher scores, though performance is still an order of magnitude short of custom hand-coded expert systems. Another online implementation [Barnes] appears to attempt a variety of deep reinforcement learning methods, but no benchmarks were made available, and we were unable to successfully run their code to retrieve comparison metrics. Overall, the limited body of work we found is largely modelled after the recent successes in beating Atari games, sometimes using variations of vanilla DQN.

Finally, recent relevant literature have identified the need to train in a distributed fashion, in order to satisfy the rapidly-increasing training requirements of modern reinforcement learning, as a result of their model-free nature. Asynchronous actor-critic methods (A3C) are one of the most cited techniques, allowing for scaling to multiple threads on a single machine. However, adding more computation does not always result in convergence, and recent research has tried to aid reinforcement learning algorithms by providing expert human demonstrations to guide the learning [Mandlekar et al., 2018, Hester et al., 2018].

3 Methods

3.1 Dataset and features

In order to interface with SLITHER.IO's online servers, we use OpenAI Gym in concert with OpenAI Universe. OpenAI Gym is a Python API for developing and comparing RL algorithms, compatible with numerical computation libraries such as TensorFlow. The `gym` library provides a collection of test problems (environments) which can be used to implement RL algorithms; the environment's `step` function implements the classic "agent-environment" loop. At each timestep, the agent chooses an `action`, and the environment returns an `observation` and a `reward`. Universe extends Gym to include a wider variety of environments, including Internet games such as SLITHER.IO. Unlike Gym's simulated environments, Universe interacts with the real edition of the game on the Internet by simulating a virtual keyboard on a VNC remote desktop. Data gathering is therefore rate-limited due to the VNC's running in real-time online.

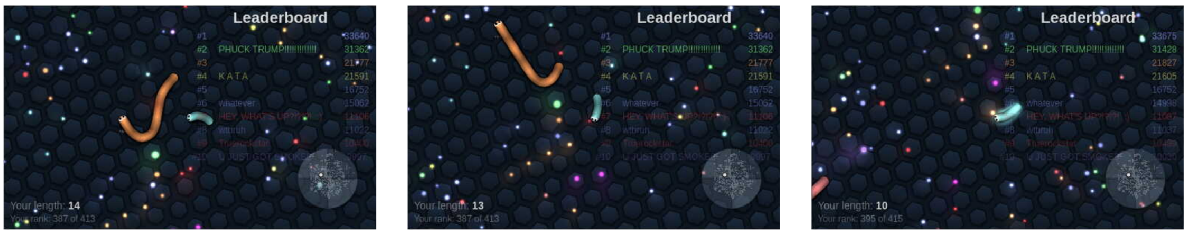

Figure 1: Sample (cropped) frames from a random agent playing SLITHER.IO on the remote desktop spun up by OpenAI Universe.

Our input data are therefore the raw frames captured from the remote desktop (Figure 1), and reward signals which correspond to the change in length of the snake. We preprocess the $768\times1024\times3$ (RGB) images by cropping them to $300\times500\times3$ to only show the relevant part of the map, then resizing them to $150\times250\times3$ to reduce the number of parameters. We perform frame-skipping from 60 to 5 frames per second, a more reasonable rate based on a rough approximation of a human's bandwidth when playing the game; this improves storage requirements and has

[2]Brockman et al. [2016a], `https://gym.openai.com/`

[3]Brockman et al. [2016b], `https://github.com/openai/universe`

been shown to make learning easier by resulting in shorter episode lengths. The color values are scaled from an 8-bit range (0 to 255) to floating-point values ($0.0 - 1.0$).

Additionally, because the Markov assumptions do not perfectly hold, there is relevant information in the frame before the current one, as well as implicit information which makes learning easier. In order to encode this while avoiding the complexity of a fully recurrent model, we merely compute the difference between the current frame and the previous one on a channel-by-channel basis, which results in the addition of three channels (for a total of six) that range between -1 and 1.

3.2 Agent model

Our agent model is based on the model presented in Mnih et al. [2013], modified to reflect differences in the input data. In particular, we begin with six channels, and then apply three convolutional layers with 64 filters on each layer, decreasing kernel size (8, 4, and 3) and decreasing stride (4, 2, and 1). We use batch normalization between each of the first three layers. In the end, a fully connected layer with rectified linear unit activations results in the estimated Q-values for each of four actions, corresponding to continuing straight, turning left, turning right, and traveling forward with a burst of speed (sprinting).

3.3 Training algorithms

We use vanilla deep Q-learning, which is a variant of traditional Q-learning in which the action-value function is learned using neural networks. The Q function tries to predict the value of taking a given action a in some state s. In traditional Q-learning, all pairs of states and actions $(s, a) \in \mathcal{S} \times \mathcal{A}$ are considered in a table and updated iteratively. For larger problems, such as this one—in which the input space spans close to a million variables—this is computationally unfeasible, and it is desireable to approximate Q with a neural network.

The algorithm we use is given in the appendix of Mnih et al. [2015]. We use a deep convolutional neural network to approximate the optimal action-value function Q from pixels s:

$$Q^*(s, a) = \max_{\pi} \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = s, a_t = a, \pi]$$

which gives the maximum sum of rewards r_t discounted by γ at each time-step t, achievable by a behaviour policy $\pi = P(a|s)$. The Q-learning update at iteration i uses the Huber loss, defined relative to the interval $[-\delta, \delta]$ (with $\delta = 1$ in this case) around zero:

$$L_\delta(a) = \begin{cases} \frac{a^2}{2} & \text{if } |a| \leq \delta \\ \delta|a| - \frac{\delta^2}{2} & \text{otherwise.} \end{cases}$$

This effectively uses mean squared error (MSE) for low error values and mean absolute error (MAE) for large values, so as to be less sensitive to outliers in the data compared to a purely-MSE approach.

We implement DQN along with various techniques outlined in the sections below using PyTorch [Paszke et al., 2017]. Besides checking for convergence of the Q function, we evaluate real performance by running the policy on the (networked) environment against other players, and measuring the median difference in length from the beginning until the snake dies (the mean is easily biased by both very lucky runs and early deaths due to random initialization) and the average reward collected per time step (200 ms). We compare those metrics to those achieved by an agent taking random actions as a baseline, and with the performance attained by a human.

3.4 Reward shaping

Due to the complexity of the game and the sample inefficiency of deep RL, we resorted to reward shaping [Ng et al., 1999] as a method to encourage or discourage various behaviors and hasten learning. The specific techniques were applied *ad hoc* after looking at the results of early training and observing its weaknesses.

Typically, the reward function returned by the environment is simply the difference in length of the snake, being > 0 if the snake becomes longer. However, the end of the episode is not associated with a negative reward—the episode simply ends. In order to disincentivize getting into scenarios that might result in the end of an episode—such as speeding head-on into the side of another snake—we manually apply this first reshaping of the reward function:

$$r'_t = \begin{cases} -10 & T - t \leq 10 \\ r_t & \text{otherwise} \end{cases}$$

where T is the end of the episode. In other words, we apply a large negative reward for the last 2 seconds of every episode. Another reshaping we performed was subtracting 1 from the reward when trying to sprint (action 3) if

below a certain size threshold, since by default the environment does not penalize this move and it results in the agent being unable to learn to avoid this behavior.

3.5 Replay buffer

Mnih et al. [2013] introduce using an experience replay buffer which stores past observations and uses them as training inputs, to reduce correlations between successive samples and smooth over changes in the data distribution. We consider a replay buffer of previous transitions collected by the agents, either fully random or under an ϵ-greedy policy. This helps stabilize training, and makes it more efficient by taking random batches which are significantly less correlated than consecutive samples. Sample efficiency is also increased by allowing for reuse of transitions in the learning. Our entries in the replay buffer consist of tuples (s_t, a_t, r_t, s_{t+1}), where $s_t, s_{t+1} \in \mathbf{R}^{150\times250\times3}$, and a_t and r_t are the action and reward respectively. In order to maximize the size of the buffer and maximize its utility, we size it to be as large as possible while fitting in RAM, and apply a variety of tricks to minimize its size, including storing it as 8-bit values without converting to floating point, not storing s_{t+1} explicitly, and only generating the delta-frame channels when a batch of samples is requested.

Additionally, inspired by Schaul et al. [2015], we do not sample from the replay buffer uniformly. Instead, using the intuition that transitions in which rewards happen are more important than rewards in which no rewards happen—and rewards are sparse enough that under a Huber loss, merely predicting always zero gives small errors and gradients for a small enough γ—we sample the transitions e_t with a probability density given by:

$$p_t = \frac{1}{4} + \log\left(1 + \sum_{i=0}^{K} |r'_{t-i}|\right)$$

with $K = 5$. This ensures a non-zero probability of seeing every transition, and assigns greater probability around the neighborhood of rewards, in a log scale accounting for the wide range of values. In future work we may try using the more commonly-used prioritized replay based on the error of the Q-function; the function above was considered due to its ease of computation.

3.6 Human demonstrations

Due to difficulty in learning a better-than-random policy, we implemented a variant of deep Q-learning from human demonstrations as introduced in Hester et al. [2018]. These are used to pretrain the Q-network and obtain a better initial policy, which improves the quality of the experiences observed and leads to faster training overall. In particular, for some number H of randomly sampled human transitions (as in 3.5) we add the following term to the loss function:

$$J_E(Q) = \max_{a\in A}\left[Q(s,a) + l(a_e, a)\right] - Q(s, a_E)$$

Here a_E is the expert action from the human demonstration, and a is the action chosen by the policy network as the argmax. The margin function $l(a_e, a)$ is $= 0$ when $a = a_E$ (i.e. when they match) and $= 1$ otherwise, as in Hester et al. [2018]. We collect the demonstrations by writing a custom script which intercepts the user's keyboard presses and feeds them to the environment. As such, the demonstrations are collected in the same manner as the rest of the experiences without distributional shift.

4 Results and discussion

Successful training of the deep Q-learning algorithm was highly nontrivial and required applying a variety of techniques to achieve a policy better than that of the random agent, indicating somewhat brittle and unstable training. Training happened on a single machine with 16 threads, 32 GB RAM, and an NVidia GTX 1060 GPU. In order to maximize the use of the machine, we developed multiple Python scripts to handle the different parts of the training. An agent program ran the environment—with multiple retries and automatic detection of soft failures of the virtual desktop—with an ϵ-greedy policy (except during evaluation), periodically updated its weights from a specified file in the filesystem, and pushed compressed files with all of its transitions to a particular folder. A learner Python script performed iterations of deep reinforcement learning with PyTorch on the GPU by pulling from a replay buffer kept in RAM that periodically checked subdirectories for new compressed files with training data, and pushed its weights every $5,000$ iterations to a file that the agents then read. This allowed multiple agents to collect data and therefore train more quickly.

Vanilla deep Q-learning did not successfully learn a good policy despite exploring a variety of hyperparameters (such as learning rates, neural network size, the target network update rate, ϵ or γ) and the techniques in Sec. 3.1–3.5.

Model	Median final score	Average reward
Random policy	3^{+1}_{-0}	0.08
Human*	145^{+36}_{-38}	0.68
Pretrain on human demonstrations, 1.5×10^4 batches	11^{+6}_{-3}	0.14
$\cdots$ followed by 3×10^4 ϵ-greedy batches	54^{+16}_{-22}	0.26
No human demonstrations, 1.5×10^5 ϵ-greedy batches	17^{+1}_{-8}	0.10

Table 1: Comparison of results of our algorithm with a random policy and a human player. Score represents the difference between the initial and final length, and so is approximately 10 points lower than the score reported by the online scoreboard. Note that average reward is a less-noisy signal, because there are significantly more transitions than there are episodes. Median rather than mean score is used because it would otherwise be biased by a small number of good runs or early deaths depending on the initial position on the map. Error bounds on the median score are estimated using the bootstrap method Efron [1982], computing the median and giving the difference between the 16th and 84th percentile, equivalent to 68% confidence intervals.

*Note that human play was benchmarked while using the constraints of the packaged environment (via a Python script written to intercept the keyboard), which is more difficult than normal play in a web browser due to input via keyboard arrows at 5 Hz, which is a far lower resolution than using a mouse at 60 Hz. The same human achieved a median score in the high hundreds or low thousands with a web browser.

As we can see in Table 1, even 150,000 iterations of ϵ-greedy resulted in a median final score of only 17, compared to 3 for a random policy, and a very similar average reward. By comparison, with the setup as in Sec. 3, $\gamma = 0.95$, $\alpha_{\text{Adam}} = 10^{-4}$, and a batch size $B = 128$, pretraining on 50 human demonstrations with only 15,000 iterations with the additional cost term J_E as in Sec. 3.6, yielded a better average reward and a comparable median final score as in the 150,000 iterations without. By taking this pretrained policy and executing otherwise the same ϵ-greedy steps, the final policy we obtained was significantly better than either, with a median final score of 54, and an average reward more than three times higher than a random policy.

The comparison above shows the high impact that a mere 50 human demonstrations had on the learning speed and potential of deep reinforcement in this game. Together with other techniques, such as reward shaping and delta frame stacking, this allowed for the successful learning of a policy. Despite being better than a random policy or one learned by vanilla Q-learning, the performance still falls short of that of an experienced human player. We can explain these shortfalls with a qualitative examination of the videos generated during the evaluation of the final policy. The biggest difference between the learned policy and a human one is the general lack of offensiveness of the RL agent, which except for an occasional attack on other snakes plays a mostly defensive and evasive game. While this leads to decent returns, it ultimately limits the achievable size. We hypothesize that such behavior could potentially be learned with more human demonstrations. An interesting behavior is the evolution of its use of sprinting, a move in the game that results in an immediate loss of size but gives a speed boost. The agent first learns to not use it at all—to avoid losing mass—but in the final policy it uses it occasionally (as humans do) to get out of an area without food and towards one with more pellets, which maximizes the discounted return. It also learns the basic mechanics of the game, turning around with spatial awareness and doing very coarse path planning to move towards food, which, while trivial for a human, is complicated to learn directly from pixels.

5 Conclusions and future work

We have demonstrated the use of deep reinforcement learning to learn a policy for the online game SLITHER.IO. Overall, the results are statistically significantly better than a random policy, but they are still significantly below that of a human player, which is not unexpected given the high degree of complexity of the game and its real-time multiplayer nature. We explored the effect of a variety of techniques which can be used to lead to a good policy that vanilla DQN was unable to learn, including reward shaping, a custom priority replay buffer, and the addition of human demonstrations to pretrain the network, which resulted in our best performing agent. This further confirms results of the effects of human demonstrations in other environments, such as Atari games [Hester et al., 2018].

For future work, we would like to spend more time training and tuning hyperparameters, both of which take a long time due to the bottleneck of playing in real-time over a network; we believe that given enough time it should be possible to learn offensive tactics. Additionally, we would like to try more advanced models than the somewhat-standard deep Q-network, using instead asynchronous actor-critic methods in a distributed fashion.

Contributions

J.C.C. and Z.F. contributed equally to the project conception, setup of initial training data pipelines and writing of milestone. J.C.C. carried out deep Q-learning model generation, learning infrastructure and training, and human demonstrations. Z.F. led poster design and final write-up.

The code is available at `https://drive.google.com/file/d/1iroTygoadpIeutSNyx_tnM8ADleIYmy2/view?usp=sharing`. For reproducibility: getting the Docker Python integration was tricky and due to Universe no longer being supported an old version of Gym had to be used that is incompatible with recent Python version, which can be fixed by editing a couple of lines in the library.

References

Z. Barnes. Slither RL agent. URL `https://github.com/zachabarnes/slither-rl-agent`. Github repository.

G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016a.

G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI Universe, 2016b.

J. Caudill. Slither.io Deep Learning Bot. 2017.

B. Efron. *The Jackknife, the Bootstrap, and Other Resampling Plans*, volume 38. Siam, 1982.

T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, D. Horgan, J. Quan, A. Sendonaris, I. Osband, et al. Deep Q-Learning from demonstrations. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

A. Mandlekar, Y. Zhu, A. Garg, J. Booher, M. Spero, A. Tung, J. Gao, J. Emmons, A. Gupta, E. Orbay, et al. ROBOTURK: A Crowdsourcing Platform for Robotic Skill Learning through Imitation. *arXiv preprint arXiv:1811.02790*, 2018.

M. Miller, M. Washburn, and F. Khosmood. Evolving unsupervised neural networks for Slither. io. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, page 55. ACM, 2019.

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.

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, 2015.

A. Y. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287, 1999.

A. Paszke, S. Gross, S. Chintala, and G. Chanan. PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration. *PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration*, 6, 2017.

T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized Experience Replay. *arXiv preprint arXiv:1511.05952*, 2015.

G. Tesauro. Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, 38(3), March 1995. URL `http://www.bkgm.com/articles/tesauro/tdl.html`.

H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double Q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.