Reinforcement Learning for PixelCopter

Zainab Khan
Electrical Engineering
Stanford University

Nikka Mofid
Electrical Engineering
Stanford University

Jeffrey Woo
Computer Science
Stanford University

Abstract

Pixelcopter is a PyGame Learning Environment side-scroller and simplified version of a classic arcade game. While the rules are simple, the game is deceptively challenging to solve using artificial intelligence. Most current approaches either simplify the game model or use deep learning. We experiment with using significantly less computationally expensive methods on the original game model. Ultimately, we found that through hyper parameter tuning and using strategically discretized state representations, we outperform a neural network that has been trained for about an hour with a fraction of the training time.

1 Introduction

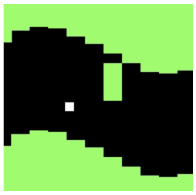

Figure 1: Example frame of Pixelcopter

Pixelcopter is a side-scroller where the player, represented as a large white square, must traverse through an infinitely generated maze of obstacles to travel for as long as possible (1). Infinitely navigating around obstacles is a classic game format found in many forms like Flappy Bird (2). PyGame Learning Environment (PLE) is a module that allows the user to play simplified versions of classic arcade games including Pixelcopter (3). Pixelcopter is a particularly interesting game on PLE because its rules are so simple, and yet the game is not trivial to master.

We explore the trade-offs of various reinforcement learning (RL) algorithms and state-space representations and compare them to two baselines: a random action policy and a Deep Q-Learning Approach implemented by user "cuongqn" on Github (4). We find that our best state-space-algorithm combinations are Q-Learning and SARSA (tied) on "y Distance from Block". On the development (dev) set, both outperform the random baseline and the gold standard DQN baseline trained for 45 minutes by about 60 and 12 frames per episode respectively. Furthermore our algorithms train in less than 1%[1] of the time as the DQN for similar performance, suggesting that while deep approaches are useful for solving games, strategic state representation can also provide suitable results.

2 Relevant Work

To our knowledge, there is limited exploration into the Pixelcopter challenge. Our project is inspired by Github user cuongqn's Deep Q-Learning approach, and we use their code as a starting point.

[1]This is calculated using a conservative estimate of the length of time for one run of our algorithm based on our training time for running 60 iterations.

Cuongqn uses a neural network based off of the DeepMind Atari paper (7), taking in the raw grey-scale pixel image as state input instead of the game state. Since cuongqn is using a famous deep learning approach to this problem and neural networks are often considered the future of reinforcement learning, we chose their results as the gold standard for our project. Specifically, we were interested in seeing if we could outperform their results using significantly less expensive computation methods.

After implementation, we did additional research to explore the space further and found two approaches for Pixelcopter, neither tackling the problem in the same way. A team at Stanford implemented Q-Learning with a one state-space implementation in CS221 (5). The team modifies the problem parameters to move the blocks further out of the way of the agent. Their project chooses to focus on one specific approach rather than exploring multiple different state-space representations and learning algorithms. According to their poster, their agent survives for an average of 25 frames. We also found another deep learning approach by Eddie Wang (6). Wang chooses to simplify the problem by removing all of the blocks. With their 2-hidden layer neural network, they were able to build an implementation that could successfully navigate the cavern.

In both approaches, the developers chose to simplify the problem structure. However, we chose to tackle the original game as is to see how our approaches could fare against a more complex challenge.

3 Relevant Development Information

3.1 Pixelcopter API

Pixelcopter provides an API that returns both the image representation of the game and the continuous game state of the problem. The agent has two possible actions of either accelerating upwards slightly or doing nothing. The agent eventually falls downward when no action is taken. While the API provides a reward based on number of blocks crossed, or a -5 for losing the game, we binarized the reward to our agent as done by cuongqn(4). The player receives +1 reward each frame until receiving a -5 for losing the game (which occurs when the agent hits a block, the floor, or the ceiling).

3.2 Datasets

We set up our codebase with the classic training, dev, and test set approach. We trained our agent for a set number of episodes (i.e. run of the game) before evaluating on a dev set to provide an unbiased evaluation of our model while tuning hyper parameters. Once we finished tuning and model building, we chose our best combinations of algorithm and state-space representation to run on the test set.

To generate these datasets, we used seeding to set the random generation point of the PLE environment at different places. All game environments used for training are seeded with PLE's default seed of 24. We set the dev set's environment seed randomly to 536[2]. Our test set's environment was also seeded randomly to 1176, and we did not run our agent on the test set until the end of our project. For evaluation of the DQN baseline, we use the same dev seed and test seed so that all models see the same/similar environments.

4 Methodology

As the main goal of Pixelcopter is for the agent to travel for as long as possible through the cavern without crashing, we use mean number of frames per episode as the metric for success on the dev set.

4.1 Baseline

Our primary baseline for this project is a fixed random policy agent. Before running the simulation, we set a random action for each state in the state-space and then test on this policy. Our second baseline is the performance of cuongqn's DQN on Pixelcopter and serves as a gold standard. We trained their model for various amounts of iterations and measured the performance.

In Table 1, we detail the performance of the DQN's baseline training time and performance after various numbers of episodes. Fixed random policy performance can be found in Section 5.

[2]We noticed that it was important to have a seed for proper comparison as the random environments changed drastically from episode to episode

Table 1: Baseline DQN Perfomance on Dev Set

Abbreviated Name	Number Training Episodes	Approximate Training Time	Mean Frames
10K DQN	10,000	10 min	20.72
15K DQN	15,000	30 min	49.04
20K DQN	20,000	45 min	66.74
50K DQN	50,000	3 hours	90.01
75K DQN	75,000	5 hours	98.42
100K DQN	100,000	6 hours	100.18

4.2 Algorithms

We implemented three classic reinforcement learning algorithms to compare and contrast: SARSA, SARSA-Lambda and Q-Learning. For each algorithm, we use an epsilon-greedy exploration strategy to help balance exploration vs. exploitation, and we zero-initialize our Q Array.

4.2.1 Q-Learning

Q-Learning is a popular model-free reinforcement algorithm. It is based off the Bellman equation, but uses the next state s' and reward r' to obtain the update rule instead of transition and reward models:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'}(Q(s',a')) - Q(s,a))$$

4.2.2 SARSA

SARSA is a popular alternative to Q-Learning. It uses an update equation similar to Q-Learning, but uses the actual action taken instead of maximizing over all possible actions:

$$Q(s,a) = Q(s_t,a_t) + \alpha(r + \gamma(Q(s_{t+1},a_{t+1})) - Q(s_t,a_t))$$

4.2.3 SARSA-Lambda

We also implemented SARSA-Lambda, which is a modified version of SARSA that uses eligibility traces in order to backpropagate reward throughout the entire state-space table. Though SARSA-Lambda converges faster on optimal performance, it has a higher time complexity because it has to propagate reward throughout the entire table at each iteration. In SARSA-Lambda, we keep track of the exponentially decaying visit count $N(s,a)$ for all state-action pairs. When we take action $= a_t$ in state s_t, $N(s_t,a_t)$ is incremented by 1. We then update the $Q(s,a)$ by adding $\alpha * \zeta * N(s,a)$ at every state s and for every action a. This is shown by the SARSA-Lambda update equation:

$$\zeta = r_t + \gamma * Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)$$

where we exponentially decay $N(s,a) = \lambda * \gamma * N(s,a)$ after performing updates.

4.3 State Representations

The Pixelcopter API's game state contains the agent's vertical position, agent distance from floor and ceiling, agent velocity, and horizontal and vertical distance from the next generated obstacle block. When using the image state, we binarize the input to be either an obstacle (white) or safe space (black). Using these two sources of information, we constructed several different state representations in order to see which combination of state representation and algorithm would help us get the highest number of mean frames per episode. Furthermore, we also evaluate combinations of these discrete state representations to see how the extra information benefits or hurts the agent.

4.3.1 5 State Proximity

Our first representation is a 5 State Proximity design that evaluates the player's proximity to obstacles, using the 15×15 pixel image around the Pixelcopter (x,y) and classify the distribution of obstacles in the image into 5 different states. We hypothesize that Pixelcopter can get better spatial awareness

to develop a policy which will help it avoid getting dangerously close to obstacles. State 0 represents a safe zone where percentage of obstacles above and below is less than 10%. States 1 and 2 represent the degree of danger from above: when the number of white pixels above is greater than 50% or between 10-50% respectively. States 3 and 4 represent the degree of danger from below using similar thresholds. If the Pixelcopter is surrounded by pixels both above and below, the state only indicates the danger from above.

4.3.2 Velocity

This state representation was inspired by observing that Pixelcopter often accelerates too quickly as a result of receiving several "up" actions, crashing into the ceiling. We hypothesized that using velocity could help the agent avoid crashing upwards. To do so, we discretized the velocity with a step size of 0.5 to represent the velocity states. Since continuous velocity typically ranges from -3 to 3, we clipped the velocity before discretizing such that we only had 7 velocity states.

4.3.3 Player y Distance to Block

On further observation, we noticed that hitting obstacle blocks was the common cause for failure cases for our first agents. To address this, we created a state representation of the location of the block relative to the player's location. Because navigating around a block requires dexterity, and requires knowing whether it is easier to go above or below a block, we ravel the player's vertical distance from the top of the block with the player's vertical distance from the bottom of the block. For our game size, these distances range to about 40, so the state-space size is 1600.

4.3.4 Downsampled Image

Observing the success of deep learning on the image size, we experimented with a downsampled image of size 5×5, with each pixel ravelled together (2^{25} states). We hypothesized that using the downsampled image might capture more information about Pixelcopter's overall surroundings, relative to the 5 state proximity solution.

4.3.5 Combination

We hypothesized that each representation alone cannot capture the full problem. Thus, we experimented with combining "5 State Proximity," "Velocity," and "y Distance to Block" into one representation by ravelling them together.

4.4 Hyperparameter Tuning

Random hyper parameter search has risen in popularity for limiting the need for manual tuning (8). We first set uniform distributions for each hyper parameter and sample a batch of parameters to train and validate the model. It is thought that 60 iterations of random parameter sampling leads to a 95% likelihood of finding top 5% optimal parameters (9). Since Pixelcopter training is not expensive, it is a good candidate for 60 iterations of random tuning. We implemented a testing framework that performs training with or without random hyper parameter tuning based on a specified flag.

4.5 Additional Approaches

We chose to standardize training to 100 iterations regardless of algorithm or state-space. We originally hypothesized that more training iterations would lead to better performance for the larger state-spaces, and compared 100, 500, and 1000 training iterations for a few of the state-spaces with SARSA (See Table 5 in Appendix). However, we found that validation performance decreased marginally with more training. Therefore, we standardized to 100 iterations to reduce time complexity and simplify training. We also briefly explored varying the reward function to either magnify or minimize positive and negative rewards. All combinations of variations seemed to have no impact on validation performance.

5 Results

5.1 Dev Set Performance

The following tables detail how each algorithm performs with each state representation. Training time is represented as total time to tune the hyper parameters.

Table 2: Dev Set Mean Frames per Episode for Different RL Algorithm and State Space Combination

State Representation	Random Fixed Policy	QLearning	SARSA	SARSA-Lambda
5 State Proximity	4	57.13	57.13	57.13
Velocity	4	20.72	20.72	20.72
y Distance to Block	11.75	**79.15**	**79.15**	73.27
Image	10.98	30.64	30.64	NA
Combination	17.78	76.67	76.67	NA

Table 3: Dev Set Train Time (mins) for Different RL Algorithm and State Space Combinations

State Representation	Random Fixed Policy	QLearning	SARSA	SARSA-Lambda
5 State Proximity	0	8.497	8.531	9.047
Velocity	0	3.833	3.575	3.715
y Distance to Block	0	7.862	7.730	117.138
Image	0	5.081	4.888	NA
Combination	0	8.746	8.805	NA

5.2 Test Set Performance

Based on our results from the dev set, we chose "y Distance to Block" and "Combination" state-spaces to evaluate on the test set. We compared them to the 20K DQN, which we slightly outperformed in the dev set, and the 100K DQN, which is the best of the baselines. Clearly, we can beat the DQN which trained for 45 minutes, but our solutions are not as strong as the longest-trained DQN.

Table 4: Test Set Mean Frames per Episodes for Best Q-Learning and Baseline Models

20K DQN (Trained 45 min)	Combined State	y Distance to Block	100K DQN (Trained 6 hrs)
65.63	70.9	**71.31**	**105.76**

6 Discussion and Analysis

6.1 Efficacy of Algorithms

As expected, we were able to beat the random policy. Interestingly, we observed similar mean frames per episode performance across all three learning algorithms. Similarities in performance could indicate that all three algorithms are converging on optimal policy for each state space with marginal differences attributed to hyper parameter tuning.

Finally, we observe that SARSA-Lambda's training time was too significant to run for larger state representations, like downsampled image, since for each iteration, the entire state-space must be propagated with rewards. For this reason, we did not proceed with the runs for "Image" and "Combine" state-space representations for SARSA-Lambda. In general, we do not recommend using this algorithm on large state-space problems. We hypothesize that SARSA-Lambda may show benefits compared to SARSA with non-binarized and more sparse rewards, but with our current problem structure, the propagation of rewards adds little extra information.

6.2 Efficacy of State Space Representation

Our results suggest that "y Distance to Block Based" state representation and the "Combination" state representation performed the best, with mean frames per episode of 79.15 and 76.67 respectively in the dev set. We hypothesize these two state-space representations achieved the highest performance because they provided the agent with detailed information of its relation to the randomly generated obstacle blocks. We observed that the cavern itself was fairly wide and easy to navigate. However, the generated obstacle blocks often took up significant space, sometimes making it very difficult even for a human player to maneuver. While "5 State Proximity" performs well by avoiding obstacles above and below it, it does not have enough timely information about the location of the block to navigate around it. In turn, we hypothesize that for the "Downsampled Image" representation, the image becomes too downsampled to converge to a good solution. From this we can see that state-space representation based on problem structure is critical to performance.

We expected the "Combination" state representation to outperform "y Distance to Block Based" state representation as it was larger and contained a combination of information we believed was useful to solving Pixelcopter. Additionally, we observed that training the "Combination" state representation for 500 or 1000 iterations did not lead to improved performance. It is possible that this larger state-space performed worse because many of the states were not encountered. On the other hand, combining these state representations could make the overall model more sensitive. Had we used a flexible training mechanism like early stopping, perhaps performance would have improved. Finally, it is possible that combining these state representations together perhaps is not the most powerful way of representing how the agent should navigate.

6.3 Comparison to DQN Baseline

Our greatest finding from our experiments was that despite training for significantly less time than the DQN, our performance was still comparable. It took our best model less than 8 minutes to train 60 full times for hyper parameter tuning whereas the neural network took significantly longer to train just once: 45 minutes to score similarly to our best model and 6 hours for best performance. Furthermore, neural networks are known for being more difficult to analyze and infer insights from. It is exciting to discover that deep learning is not required to approach this challenge. While the DQN's upper bound of performance is larger than our best model, it had to be trained for much longer, and we hypothesize that we have not yet found the optimal state space representation for a non-deep learning approach.

It is important to note that our algorithms were trained on a separate machine than the baseline DQN, so differences in processing power may effect the timing results. However, the difference in timings is significant enough that we can see our models were much better than the DQNs in terms of time-to-convergence. Additionally, we must acknowledge an edge case; we did not implement early stopping into our methods or the baseline neural network. Without it, we do not know how much stronger the DQN would have performed if we let it train even longer.

Regardless, training deep neural networks is computationally expensive. Additionally, all actions chosen in test time require much more compute using the DQN than using our methods. Our findings suggest that less expensive techniques are still an area of interest for RL applications and that computationally expensive methods should not be immediately applied to problems because of their prior record of success.

7 Conclusion and Next Steps

We hypothesize that a significant source of performance error occurs from needing better representation of avoiding obstacle blocks. With more time, we would also like to solidify the rigor of our results by adding training mechanisms like early stopping to understand the trade offs of fixed training iterations on performance.

In conclusion, using traditional RL algorithms with strategically designed state representations achieved reasonable results compared to a DQN at a fraction of the training time. This is relevant because expensive computational processors are not accessible to everyone, and our findings lay the groundwork for further research into the success of non-deep learning approaches to Pixelcopter.

Individual Contributions

Zainab worked on developing SARSA, creating and combining state-spaces, pipeling the train/dev/test flow, timing different algorithm, and writing the paper. Jeff worked on creating SARSA-Lambda, hyper parameter tuning, global seeding, state space discussion, model testing, and writing the paper. Nikka trained the neural network, developed Q learning, experimented with different state spaces and models, contributed to state space discussion, performed numerous tests, and worked on writing the paper

References

[1] Pixelcopter. (2016). Retrieved from `https://pygame-learning-environment.readthedocs.io/en/latest/user/games/pixelcopter.html`.

[2] McDonnell, M. (2014). Flappy Bird. Retrieved from `https://flappybird.io/`.

[3] Ntasfi. (2016, February 21). ntasfi/PyGame-Learning-Environment. Retrieved from `https://github.com/ntasfi/PyGame-Learning-Environment`.

[4] cuongqn. (2018, March 4). pixelcopter-DQN. Retrieved from `https://github.com/cuongqn/pixelcopter-DQN`.

[5] Mei, D., Nguyen, R., Jobson, E. (2017). Pixel Copter? I Hardly Even Know Her! Retrieved from `http://web.stanford.edu/class/archive/cs/cs221/cs221.1192/2018/restricted/posters/dmei/poster.pdf`.

[6] Wang, E. (2018, August 12). eddieshengyuwang/PixelCopter-RL. Retrieved from `https://github.com/eddieshengyuwang/PixelCopter-RL`.

[7] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. NIPS. Retrieved from `https://deepmind.com/research/publications/playing-atari-deep-reinforcement-learning`

[8] Bergstra, J. (2012). Random Search for Hyper-Parameter Optimization. Journal of Machine Learning Research. Retrieved from `http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf`

[9] Zheng, A. (2015, October 16). Evaluating Machine Learning Models. Retrieved from `https://www.oreilly.com/ideas/evaluating-machine-learning-models/page/5/hyperparameter-tuning`.

Appendix

Table 5: SARSA Performance for Varying Training Episodes for Different State-Space Sizes

	100	500	1000
5 State	57.13		57.13
y Distance to Block	79.15	78.7	78.75
Combination 2	74.69	73.59	72.95