Reinforcement Learning in Sonic the Hedgehog

ACML - Project

Uvir Bhagirathi : 1141886

Storm Menges : 1102107

June 8, 2018

Chapter 1

Introduction

In this report we implement different reinforcement learning (RL) algorithms on the Sonic the Hedgehog game. The results of these algorithms are then analyzed in terms of percentage completion of the level and compared against one another.

We intend to explore the effectiveness of various reinforcement algorithms with regards to a complex environment in which the agent is required to learn to progress through the level. The platform we decided to apply the algorithms on is the game Sonic The Hedgehog which was initially released on the 23rd of June 1991.

This game is 2 dimensional and consists of different enemies and obstacles that the agent is required to overcome in order to complete the level. We will test the algorithms on the first level of the game and then further test the best performing agents on further levels.

Chapter 2

System Specifications

CPU	Intel Core i7-7700 CPU @ 3.60GHz x 8
RAM	16Gb
GPU	GeForce GTX 1060
CUDA Cores	1280
GPU RAM	6Gb
Operating System	Ubuntu 16.04 LTS 64-bit

Chapter 3

Background

3.1 Gym Retro

Gym Retro, a wrapper for video game emulator cores, uses Libretro API to turn the game into Gym environments. Gym Retro runs on Linux, macOS and Windows with Python 3.5/3.6 support, and includes support for multiple classic game consoles and a dataset of various games.

Each game has files listing memory locations for in-game variables, reward functions based on those variables, episode end conditions, savestates at the beginning of levels and a file containing hashes of ROMs that work with these files. ROM files must be imported before it can be used in the emulator.

The Gym-Retro platform is unique in the way that it supplies the reward function as well as various information about the current state of the game including a screenshot of the current state which is ultimately fed into the DQN. The aforementioned information also consists of the x and y co-ordinates of the agent as well as data like the end of the level x co-ordinate and the amount of rings

This will be used to emulate the game, Sonic the Hedgehog, and allow our agent to interact with the game.

3.2 Level

The Green-Hill Zone level will serve to train our agent and was selected primarily because it was the first level in the game. However, other levels were used to train agents as there was sufficient time.

Figure 3.1: Green-Hill Zone

3.3 Reinforcement Learning

Reinforcement learning, an approach to learn agents based on reward functions, allows the agent to retain a good agent-function solely from experience. The basic idea here is that good actions or actions that aid the agent in achieving it's goal, obtain a positive reward (i.e. a positive number); whereas those actions that hinder such achievement are rewarded negatively or receive a zero-reward. This will allow the agent to learn the actions that maximize its long-term rewards as seen in figure 3.2 below.

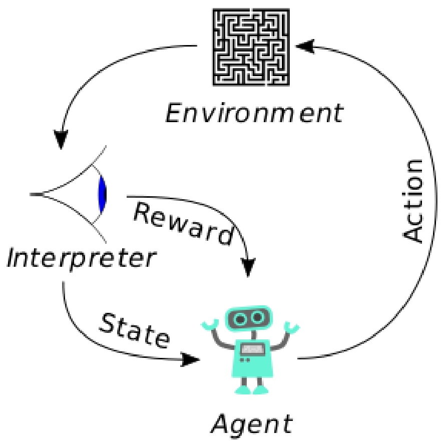


Figure 3.2: Reinforcement Learning Illustration

3.4 Algorithms

The following subsections serve as a list of the algorithms that were implemented. All were attempted and the method for each will be further discussed in the Methodology section.

3.4.1 Q-Learning

Q-learning is a reinforcement learning technique used in machine learning. The technique does not require a model of the environment and manages problems with stochastic transitions and rewards without requiring any adaptations.

Reinforcement learning involves an agent, a set of states S, and a set of actions per state A. By performing an action $a \in A$, the agent transitions from state to state. Executing an action in a specific state provides the agent with a reward and the goal of the agent is to maximize its total reward. This is achieved by adding the maximum reward attainable from the future state to the reward in its current state, effectively influencing the current action by the potential reward in the future. This reward is a weighted sum of the expected values of the rewards of all future steps starting from the current state.

3.4.2 SARSA

State-action-reward-state-action (SARSA) is a reinforcement learning technique that uses an on-policy learning technique. This means that it updates the policy based on the action taken.

A SARSA agent interacts with the environment and updates the policy based on the actions taken. This is called on-policy learning algorithm. The Q-value for each state-action is updated by an error which are adjusted by the learning rate α. Q-values represent the possible reward received in the next time step for taking action a in state s, plus the discounted future reward received from the next state-action observation.

3.4.3 Watkin's Q(λ)

The Q(λ) algorithm is similar to the Q-learning algorithm with the exception that eligibility traces are used and learning for an episode stops at the first non-greedy action taken. Watkins Q(λ) is an off-policy method: as long as the policy being implemented selects greedy actions, the algorithm will continue learning the action-value function for the greedy policy. However; when an exploratory action is selected by the behaviour policy, the eligibility traces for all state-action pairs are set to zero and hence, learning stops for that episode.

3.4.4 SARSA(λ)

The SARSA(λ) algorithm refers to the SARSA algorithm upon which eligibility traces are added. The basic algorithm is similar to the SARSA algorithm, except that backups are carried out over n-steps and not just over one. An eligibility trace is kept for every state-action pair.

3.4.5 JERK

JERK stands for Just Enough Retained Knowledge and is an algorithm created by OpenAI specifically for Sonic. The main idea is to explore using a simple algorithm, then to replay the best action sequences more and more frequently as training progresses, however it has the core concept that the agent should primarily move right and jump periodically with a chance to move left.

3.4.6 DQN

A Deep Q Network (DQN) is a complex model that consists of two convolutional neural networks and an update function that is based on the Q-Learning update function. The reason we use two convolutional neural networks (CNNs) instead of one is due to the fact that the Q-learning update rule makes the network unstable if the weights are fluctuating every iteration. Thus we introduce the second CNN in which the weights are frozen and are updated to the new weights after C steps to increase stability.

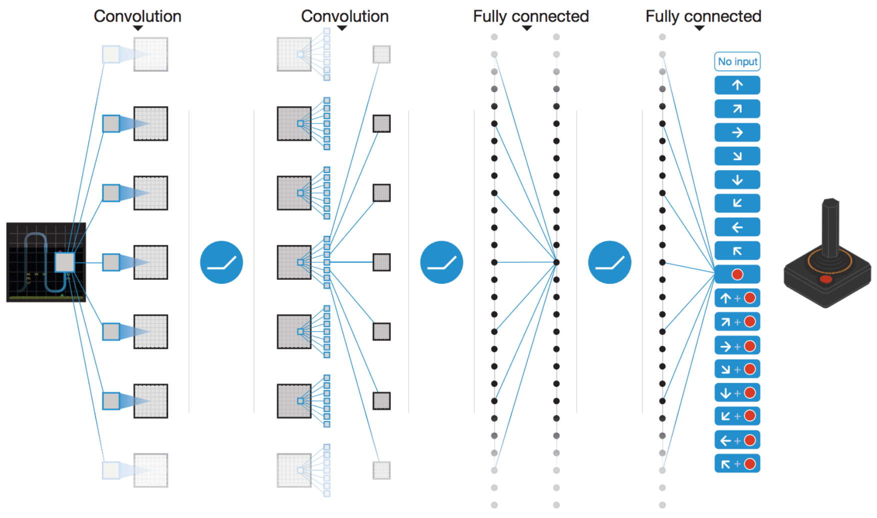


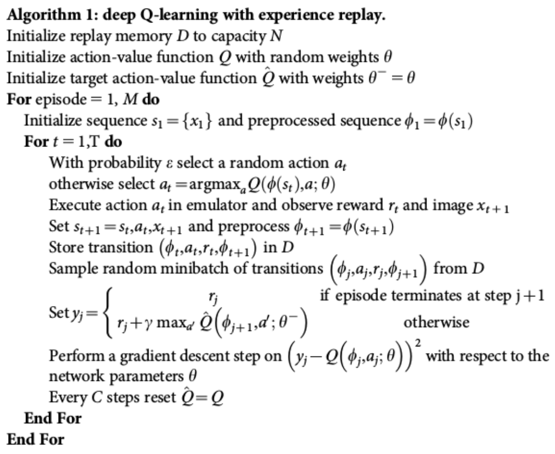
Figure 3.3: Deep Q Network

Chapter 4

Methodology

4.1 DQN

Below is a simple description of some of the attributes of the DQN algorithm.



```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, $M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T do
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the
    network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  End For
End For
```

Figure 4.1: DQN Pseudocode

4.1.1 Experience Replay

A DQN easily over-fits the current episodes of a game. When a DQN has been over-fit it becomes hard to produce various experiences for different situations. An Experience Replay is used to solve this problem. The Experience Replay stores experiences including state transitions, rewards and actions which are the necessary data to perform Q learning, and makes mini-batches to update neural networks. This technique expects the following merits.

1. Reduces correlation between experiences in updating DQN.
2. Increases learning speed with mini-batches.
3. Reuses past transitions to avoid catastrophic forgetting.

4.1.2 Target Network

In TD error calculation, target function is changed frequently with DQN. An unstable target function can make training fairly difficult. So at every thousand steps of the algorithm this Target Network technique fixes parameters of target function and replaces them with the latest network parameters.

$$Q(s_t, a) \leftarrow Q(s_t, a) + \alpha \left[r_{t+1} + \gamma \max_p Q(s_{t+1}, p) - Q(s_t, a) \right]$$

Figure 4.2: Temporal difference (TD) Error

4.2 Episode Boundaries

Experience in the game is divided up into episodes, which roughly correspond to an in-game life. At the end of each episode, the environment is reset to its original save state. Episodes can end on three conditions:

1. The player completes a level successfully. This corresponds to the agent passing a certain horizontal offset within the level.
2. The player loses a life.
3. 4500 time-steps have elapsed in the current episode. This is about 5 minutes of in-game time.

Note: The done condition of each level with a boss fight is defined as the horizontal offset just before the boss fight.

4.3 Frame Skip

Each step of the gym-retro environment progresses the game by $\frac{1}{60}$th of a second. However, frame skips of 4 are used at each step of the environment. With a frame skip of 4, a time-step is roughly $\frac{1}{15}$th of a second. We believe that this is a good enough resolution to play Sonic well whilst saving time and resources.

4.4 Observations

A gym-retro environment produces an observation at the beginning of every time-step. This observation is always a 24-bit RGB image. For Sonic, the screen images are 320 pixels wide and 224 pixels tall.

4.5 Actions

At every time-step in the environment the agent does an action. Each action represents a button combination. Actions are a one-hot/binary vector where a '1' represents a button press and a '0' does not. The action space of the Sega Genesis console contains the following buttons:

$B, A, MODE, START, UP, DOWN, LEFT, RIGHT, C, Y, X, Z$

However, only a few possible button combinations are used in Sonic. These 8 combinations are :

```
{{}, {LEFT}, {RIGHT}, {LEFT, DOWN},
{RIGHT, DOWN}, {DOWN}, {DOWN, B}, {B}}
```

4.6 Rewards

During an episode, an agent is rewarded a cumulative reward at any point in time in proportion to the horizontal offset from the initial position of the agent. Therefore, going right always provides a positive reward.

The two main components of the reward function are the horizontal offset and the level completion bonus. The horizontal offset reward is normalized to 9000 per level. The level completion bonus is 1000 and drops linearly to 0 at 4500 time-steps. This encourages the agent to finish the level as fast as possible.

Note: Immediate rewards can be deceptive, it is sometimes necessary to go backwards for prolonged amounts of time. Therefore, an environment wrapper is used to to not punish the agent for backtracking.

4.7 Wrappers

Wrappers are applied to the gym-retro environment to alter certain attributes. This helps the agent by producing the desired output.

4.7.1 Sonic Discretizer

The Sonic Discretizer wrapper allows for discrete actions to be taken in the Sonic game. This basically rounds the input values given to the step function and makes sure the correct action takes place.

4.7.2 Warp Frame

This wrapper warps the observations to a 84x84 sized image as done in the *Human-level control through deep reinforcement learning.*

4.7.3 Frame Stack

The Frame Stack wrapper stacks 4 of the frames from the Warp Frame wrapper into one image. This stacked image is used as input into the DQN.

4.7.4 Allow Backtracking

This wrapper uses the max x coordinate of the agent as part of the reward rather than the change in x. By doing this the agent is not heavily discouraged from exploring backwards if there is no way of advancing.

4.7.5 PyTorch Frame

The PyTorch Frame wrapper rearranges the way the dimensions of the frames are stored which allows the frames to be submitted to pyTorch directly.

Chapter 5

Results and Analysis

5.1 Q-Learning and SARSA

The Naive Q-Learning and SARSA approaches were able to get through most of the first level however they got stuck at at the same spot in the level which require a significant amount of speed from the agent to pass, which the agent was not able to learn.

5.2 Watkin's Q(λ) and SARSA(λ)

Watkins Q(λ) and Sarsa(λ) were better performing than the two original algorithms as they finished the first level occasionally but they didn't complete the level often enough to be considered a success, despite having the eligibility tracing in the implementation

5.3 Watkin's Q(λ) with JERK and SARSA(λ) with JERK

These algorithms were both considered successes as they often completed the first level due to the JERK concept that was added to the algorithms. Although these algorithms were considered successes, as soon as they were train on other levels they never completed the level due to the increased complexity present in the higher levels. This left us with barely any options besides deep learning so eventually we opted to implement a DQN.

5.4 DQN

Due to the results in the *Human-level control through deep reinforcement learning* paper we assumed that the DQN would be able to beat the first level, thus we started training the DQN on the second level which it was able to beat where other algorithms were unable. It should be noted that the implemented DQN is trained on the GPU due to the fact that training it without CUDA on the CPU would take days just to start converging where the GPU allows us to do get results within a day.

Chapter 6

Conclusion

In conclusion we deduced, through the implementation of several algorithms, that reinforcement learning can be applied to the Sonic the Hedgehog game and provide decent results, which include the ability to finish multiple levels. When we compare the 7 different algorithms we implemented we found that the deep learning approach was superior to all of the other implementations simply due to the fact that it has the ability to learn based on the pixels in the image using the CNN and not the co-ordinates like the other approaches.

This resulted in far better results when sufficiently trained as confirmed by *Human-level control through deep reinforcement learning*

Chapter 7

References

[1] Nichol, A., Pfau, V., Hesse, C., Klimov, O. and Schulman, J., 2018. Gotta Learn Fast: A New Benchmark for Generalization in RL. arXiv preprint arXiv:1804.03720.

[2] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. and Petersen, S., 2015. Human-level control through deep reinforcement learning. Nature, 518(7540), p.529.

[3] James, S. and van Niekerk, B., 2018. University of Witwatersrand, Internal PyTorch Tutorial.

[4] Gupta, K.M., Performance Comparison of Sarsa () and Watkins Q () Algorithms. nd): n. pag. Print.

[5] Lin, L.J., 1993. Reinforcement learning for robots using neural networks (No. CMU-CS-93-103). Carnegie-Mellon Univ Pittsburgh PA School of Computer Science.

[6] OpenAI Blog. 2018. Retro Contest. [ONLINE] Available at:
https://blog.openai.com/retro-contest/.

[7] OpenAI Blog. 2018. Gym Retro. [ONLINE] Available at:
https://blog.openai.com/gym-retro/.