Reinforcement Learning: Autonomous Race Car

Olivier Pham*
Stanford University
Management Science and Engineering
mdopham@stanford.edu

Stephanie Sanchez*
Stanford University
Computational and Mathematical Engineering
ssanche2@stanford.edu

Vishal Subbiah*
Stanford University
Computational and Mathematical Engineering
svishal@stanford.edu

December 15, 2017

Abstract

*We have implemented various versions of the **Q-Learning** algorithm to train an agent to play the Atari video game Enduro. We have also created a **feature** and **reward extraction** to relay information from the game to our agent to help the learning process. All algorithms were compared by the position of the agent at the end state.*

1 Introduction

1.1 Motivation

Autonomous vehicles for the real world environment have been a long term ambition and in recent years endeavors towards this objective have shown very promising results. But what about autonomous driving for video games? If we can simulate autonomous driving in a game, it may give further insight to applying it in the real world. We propose to implement Reinforcement Learning (RL) to train an agent in the Open AI gym Atari game, Enduro-v0. The original goal of the game was to attain the maximum points but we decided to see if our agent could reach first place among 200 other cars. The goal is for the agent to learn from its environment and actions, the intention of the game and develop an optimal policy to win.

Figure 1: Screenshot from the game

1.2 Related Work

Previous work for RL and car racing used pixel values and rewards from the gym environment to learn optimal policy [1]. Other work that includes learning Atari games involves raw pixels as inputs and outputs a function that estimates future rewards that is generated by coupling of a Convolutional Neural Network (CNN) with a form of Q-Learning [2]. A last take on playing Atari games

[0]*All authors contributed equally to this work.

defines a constant model and hyperparameter settings for the RL algorithms with a classical input features from Atari environment into the algorithms [3].

Some of the challenges we will face is the large state space since its a continuous game (and not turn based) and we are using the raw pixels.

1.3 Inputs and Outputs

We will be using the OpenAI Gym environment for this project.
This environment provides us with pixels, rewards and a boolean indicating if the episode has ended (which happens after 13300 frames). The observation space is a 210*160 pixels 8-bit RGB image of the screen which is represented by an array of size (210, 160, 3).
There are 9 possible **actions** which are:

- No Operation
- Accelerate
- Move Right
- Move Left
- Brake
- Brake Right
- Brake Left
- Accelerate Right
- Accelerate Left

For every state and action, the default reward is defined the following way:

$$\text{Reward(s,a,s')} = \begin{cases} 1 & \text{when overtaking a car} \\ -1 & \text{when crashing into another car} \\ 0 & \text{otherwise} \end{cases}$$

However, as the goal of the project is to learn a policy for reaching the first position as quickly as possible, we will use the following reward:

$$\text{Reward(s,a,s')} = 200 - \text{Position in the race at the state s'}$$

1.4 Metrics

Our first goal will be to reach the first position before the end of the game while reducing the number of cars hit. To measure our success we will assess our position at the end of the race, since we bounce backwards if we hit another car.

We could also explore delaying the reward to see how well the agent behaves with limited feedback. Another possible exploration is to introduce stochasticity by assigning a certain probability to the action chosen.

If we manage to reach the goal and its variations, our next objective will be to reach the first position as quickly as possible. Therefore, the metric we will measure and try to minimize the number of frames required to reach the first position.

2 The Model

2.1 Player Class

The constructor for the Player class takes the Atari environment as input and utilizes an **MDP** class. The environment is used to trigger the **start state** for the player (by reseting the environment for each initialization of a player). The **actions** are as described in section 1.3. The success and reward function is used for running Q-Learning simulation.

We have defined a **success and probability** function that utilizes the Atari environment's step function to return a new observation based on the chosen action. The step function also returns a parameter that signals the end of the game which is our **end state**.

2.2 Reward Frame Extraction

As OpenAI only returns a 210*160 pixels image, we need to find a way to extract our position in the race from the screen image. Our first approach was to extract the image of the number as an array of size 9*8 and then feed it into Google Tesseract, a character recognition engine. However, the processing time was extremely slow (approximately 1s per frame) and made running any

learning learning algorithm inconceivable.

As digits were all the same (i.e. all the 1 are exactly alike for instance), and that there are 10 different possible characters (from 0 to 9), we figured that we needed at most 10 relevant pixels to identify the characters. Indeed, this makes it a system of 10 variables with 10 equations.

Therefore, after having identified the arrays corresponding to the 10 different digits, we looked for groups of pixels which would be different on all the 10 digits array. There were no couple nor triplet of pixels which were consistently different on all the 10 arrays, but there were several quadruplets which worked.

We chose one of them and simply mapped each combination to the corresponding number.

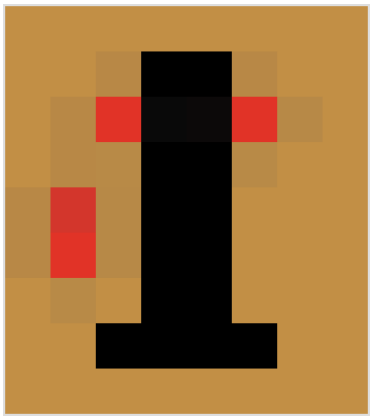

Figure 2: The four chosen pixels color values (in red) are different across all 10 digits

2.3 State Reduction

The original RGB image returned by the environment has 210*160 pixels, which can each take a value between 0 and 255 for Red, Green and Blue. Not only does the image contain too many irrelevant information, but it also greatly increases the observation space.
Therefore, we decided to focus only on the important part of the screen that is to say the portion of the road in front of our vehicle, where other cars might appear: this zone was obtained by measuring the curve of the road, which is given by the location of the road at the horizon. In order to reduce the observation space and hopefully decrease computation time, we reduced the quality of the image to a 4*7=28 pixels image. Finally, given that having RGB colors was not a necessity to play the game, we converted the image to greyscale, and then 3 level of black/grey/white. The final state was represented by a tuple with a length of 28.

Figure 3: Sample image fed into our algorithm: the white pixel represents a car appearing in front of our vehicle

3 Algorithm

3.1 Value Iteration

Value iteration did not perform well, since the number of frames an action runs is equally probable to be 2,3 or 4 frames. So we cannot tell the probability of each action and so value iteration only chooses a single action and does no better than the baseline.

3.2 Reduced Action Space

To test the framework of our algorithms we reduced the action space to be: [1,7,8]. Where action 1 is accelerate, action 7 is accelerate right, and 8 is accelerate left.

3.3 Q-Learning (exploration and learning)

We have implemented the Q-Learning algorithm seeing how it only relies on states and actions. The implementation utilizes an explore rate, β, and a learning rate, α which characterizes this Q-Learning as **unsupervised learning**.
The algorithm follows the idea of a matrix such that each row represents a state and each column is a connected

states through an action. The values of the columns are computed by the following:

$$Q(s,a) = \alpha * (Reward(s,a) + \gamma * max(Q(s', \forall a)))$$

where γ is the discount factor and the state inputs used are the states mentioned in section 2.3 (state reduction).
The idea is that we run the algorithm for a number of episodes and each episode trains the model such that the player explores its environment and collects an award until it reaches the end state. So with each episode we update/optimize our $Q(s,a)$ values.

The updates for β and α are as follows:

$$\beta = max\{\rho, min\{1, 1 - log\frac{t+1}{50}\}\}$$

$$\alpha = max\{\phi, min\{\frac{1}{2}, 1 - log\frac{t+1}{50}\}\}$$

where ρ = 0.01 is the min explore rate and ϕ=0.1 is the min learning rate.
The explore rate governs which actions to select. If any random generated number is smaller than β then the chosen action is randomly selected from the list of actions. Otherwise, we chose action with the max $Q(s,a)$ over all actions for the current s. So if the player is not exploring something new, then it will proceed with the optimal path. The results of the algorithm exhibited that the performance of the learned policy was actually worse than the random agent. The learned policy also chose to use all the actions 1,7, and 8.

3.4 Q-Learning (weighted)

The weighted Q-Learning with feature extraction is as the one seen in the class homework Blackjack, but modified to fit our environment. That is, the state inputs are the rgb images returned from the environment's step function. So we approximate the Q function by the following

$$\hat{Q}_{opt}(s,a) = w * \phi(s,a)$$

where w is the weights and ϕ is the feature extractor function. For now the feature extractor function is the identity and all feature values are set to 1. This is one parameter that can contribute to policy results. The weights are updated by

$$w_{f_k} = w_{f_k} - \alpha * (Q(s,a) - (Reward(s,a) + \gamma * V_{opt}(s',a)) * f_v$$

where f_k is the feature key, f_v is the feature value, α is the step size ($\frac{1}{iterations}$), and γ is the discount. Like the above Q-Learning algorithm this weighted one also uses an exploration parameter to determine random based actions or the best action, however the exploration term remains constant.

3.5 Preliminary Results

The exploring and learning Q algorithm was simulated with 800 iterations as seen below showing drastic fluctuation in rewards per episode.

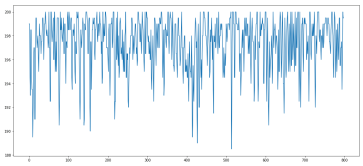

Figure 4: Plot of reward per episode with 800 iteration simulation

The weighted Q-Learning algorithm was performed by simulation for the following number of iterations: 10,20,30,40, and 50. Below are plots of the rewards returned per episode(game) in a simulation and average reward of number of iterations per simulation. It seemed that the average reward per simulation increased for the most part. The chosen actions for the learned policy were 8 for about half of the episode and 1 for the remainder.

Final Results

Feature Extraction

We wanted our feature extraction to be able to extract the number of vehicles on the road and relay this information to the Q-Learning Algorithm. We noticed that the frames are rather pixelated and not of high resolution, so image

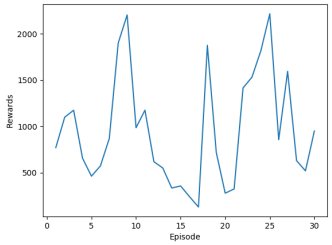

Figure 5: Plot of reward per episode in a 30 iteration simulation

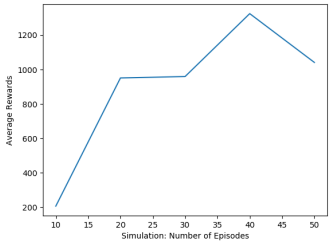

Figure 6: Plot of average reward from number of iterations per simulation.

segmentation/labeling would be beneficial.
So for each pair of state and action, (s, a), we take in the state as an rbg image (frame) and convert it to greyscale and crop it to size 81 x 300. This cropped image captures the road, grass, and cars on the road.
Then we apply Otsu's Thresholding Method which converts the greyscale image to a binary image by thresholding pixel values to be either of class 0 or 1. This effectively separates the cars and sides of the roads against the background of the road and grass.
The extraction would not always capture our agent but we also added it to the count of cars found on the road (see Figure 7). To ensure that our agent would not be counted twice, the cars that were found had one pixel value tested (from the state rbg image) to make sure it was not equal to our agent's pixel value (the agent is the only white car).

The cars found were also reasonably close to our agent, but not too close, so our agent could avoid hitting another car. After all cars had been accounted for, this value was added as the value to the feature of (s, a). That is,

$$\phi(s, a) = c$$

where c is the number of total cars on the road. There is always at least 1 car on the road and that is our agent.

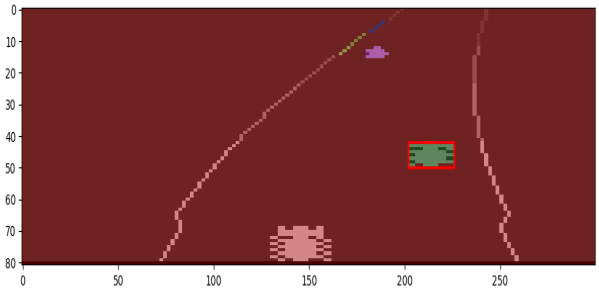

Figure 7: Car located around the agent.

Numerical Results

Figure 8 displays the results of all algorithms. The game has many terrain changes (day, night, snow, etc.) so all methods were run for the day part of the game and were compared by the final position (out of 200 positions) reached at the end state (end of the day terran). The frames (rgb images) were used for the inputs of all methods, except for the table Q-Learning which used state reduction mentioned in section 2.3.
The baseline model was for the agent to keep accelerating and reached a final position of 194. The Q-Learning Function Approximation was applied twice. Once with the identity feature extractor (all values were set to 1) and once with the feature extraction of cars mentioned in section 3.5. It is clear that the Q-Learning with the feature extraction of cars performed best by having the agent

reach a final position of 175 and by Figures 9 and 10. Figure 11 also show the the doubling of reward accumulation per episode for 50 episodes with the new feature fraction compared to the accumulation of rewards from the identity feature extractor shown in Figure 12.
The Q-Learning with the car feature extraction added a slight delay to the game and was ran on the Google Cloud Platform with 4 vCPUs and 15 GB of memory. This was completed in 14 hours.

	Baseline	Value Iteration	Q-learning	Q-learning (Function Approx.)	Q-learning (Feature Extraction)
Result	194/200	199/200	187/200	187/200	175/200

Figure 8: Results for algorithms.

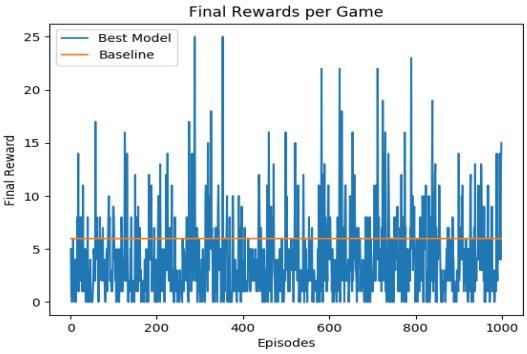

Figure 10: Final position (reward) of the end state per episode for 1,000 episodes.

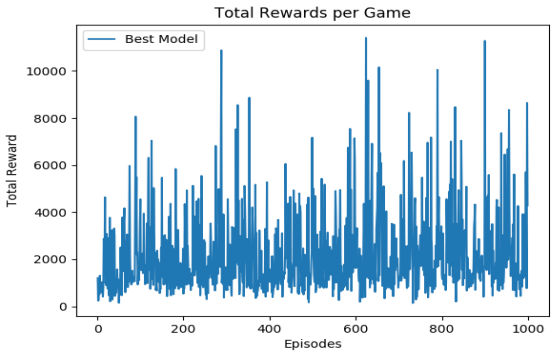

Figure 9: Accumulation of rewards per episode(game) for 1,000 episodes.

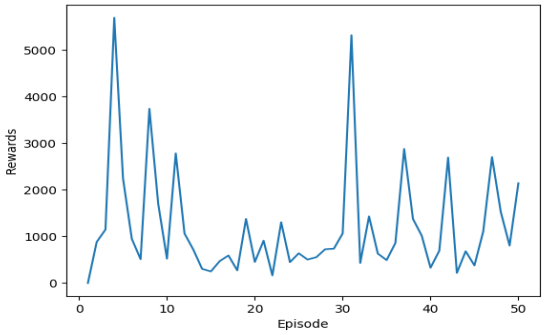

Figure 11: Reward accumulation per episode for 50 episodes using car feature extraction.

Future Work

Expanding the State Space

Our next steps would be to expand the actions space to all 9 moves from the current 3 we have shortlisted. We are currently running only the first 1000 iterations of the game to train the car in the same terrain. We would like to expand to work on the different terrains. From the limited games we have run we noticed a significant improvement with Q learning and so with enough episodes we should be able to win the race.

Improving the reward function

The reward function we used so far took into account the current position of the vehicle. Hence, a bad action (for instance crashing into another car) while being in a good position is considered more valuable than a good action while being in a bad position.
This is probably not the best reward function as our agent doesn't properly learn to drive: some (state, action) have high q-values simply because they were visited while in a good position.

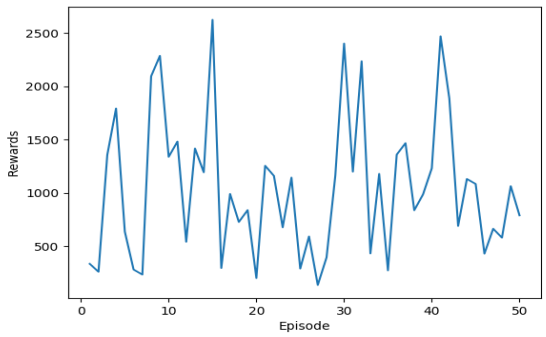

Figure 12: Reward accumulation per episode for 50 episodes using the identity feature extraction.

References

[1] M. A. Farhan Khan, Oguz H. Elibol *Car Racing using Reinforcement Learning*.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller *Playing Atari with Deep Reinforcement Learning*.

[3] David Hershey, Rush Moody, Blake Wulfe *Learning to Play Atari Games*.