Reinforcement Learning for Rapid Roll

Bera Shi, Zhecheng Wang, Yang Li

Abstract — **Our project aimed at implementing Classical Reinforcement Learning on training an agent to play "Rapid Roll", a popular video game, which involved decision making in the complex and continuous environment. In our project, we have examined how Deep Reinforcement Learning (Deep Q-Learning), taking the power of Deep Neural Network (DNN), could outperform the classical Reinforcement Learning on certain tasks. While the classical Reinforcement Learning learned the backstage hand-crafted features of the game, which were of low dimensions, we fed our Deep Reinforcement Learning with images of the game interface, which were of high dimension. Both of these two algorithms took real-time features as inputs from the game simulator, returned the best choice of actions of next frame to the simulator and made optimization of the fitted value functions. Result showed that agent with Deep Reinforcement Learning could get far more scores than agent with classical Reinforcement Learning and human players, and adding human-playing frames into the replay memory will significantly speed up the training.**

I. Introduction

REINFORCEMENT learning is an exciting topic in the machine learning field, in which we concern about how software agents make decision and take actions in an environment to maximize some notion of cumulative rewards. In traditional Reinforcement Learning the problem spaces were very limited and the possible states in an environment were only a few. This is one of the major limitations of the traditional approaches. Throughout the years there have been a couple of relative successful approaches that were able to deal with larger state spaces, and combination of deep learning and reinforcement learning becomes a trend in Artificial Intelligence. On the other hand, video games provide a rich testbed for artificial intelligence methods. The implementation of Deep Reinforcement Learning (DRL) on video games will visually present how intelligent a machine can be, and to what extent machine can perform better than humans on definite tasks. In this project, we sought for a single agent trained with DRL which can outperform human in playing a popular video game, Rapid Roll, and compared its performance with that of traditional reinforcement learning algorithm.

In the Rapid Roll game, a player is required to keep the ball going downstairs with all platforms rolling upwards. It is killed if it drops down directly or collides with the ceiling at the top. The player can move the ball left or right to jump from one platform to another. Bonus may appear on some platform, providing additional life. There are 4 lives in total

for each around. The rules are very simple, but it is not easy to keep making right instant decisions especially when the ascending speed of platforms increases. Thus it is an ideal criterion to evaluate the power of classical reinforcement learning and deep reinforcement learning.

II. Related Work

This project was inspired by the prior research *Human-level control through deep reinforcement learning* conducted by Google DeepMind[1][2]. The authors developed a novel artificial agent, implemented with deep Q-learning, that can learn successful policies directly from high-dimensional sensory inputs using the end-to-end reinforcement learning method. The deep Q-learning agent received only the image pixels of the game interface and the game scores as inputs and was able to surpass the performance of all previous algorithms. The method of our work was mainly developed on the basis of this work.

Another project *Deep Reinforcement Learning for Flappy Bird*[3] conducted by Kevin Chen trained an agent to play Flappy Bird game by implementing deep reinforcement learning. The author used a feature extractor and built a Deep Q-network to deal with the task. By learning screenshot pixels, the author has found optimal policy and achieved results exceeding human level. Compared with this project, our model has more possible actions. Moreover, the agent needed to make decision according to the positions of all platforms which filled the whole interface instead of just the position of the next obstacle, as in the Flappy Bird.

III. Methods

A. Game simulator

In order to facilitate our training algorithm, we have developed two game simulators: The simulator F took action as the single input and returned the state of next frame. The simulator G took both actions and current states as inputs and returns the state of the next frame, which was implemented in the classical reinforcement learning algorithm to observe the state of the next frame with different actions being taken. The frameworks of these two simulators are shown in figure 1a and figure 1b.

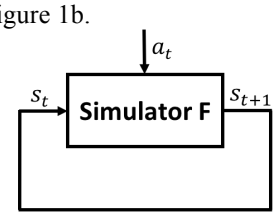

Figure 1a Simulator F

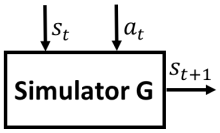

Figure 1b Simulator G

Both of them were of deterministic models. Given fixed action a_t(or state s_t and action a_t), the state of next frame could be exactly determined based on the rules of the game.

In the deep Q-learning, the simulators returned the UI images as states, while in classical reinforcement learning, the simulators returned a vector containing the x and y coordinates of all current platforms, player and possible bonus.

B. Classical Reinforcement Learning

We regarded playing "Rapid Roll" as a continuous-state Markov Decision Process (MDP) and implemented the Fitted Value Iteration algorithm to approximate the value function in the process. In our case, the number of the actions was 3. They are, do nothing, move left, move right, denoted as a_0, a_1, a_2.The game states were represented by a vector space $S = R^{11}$, which contained the y coordinate of the player, and the distances between players and bonuses, as well as the distances between players and platforms. There could be at most 5 platforms and 5 bonuses at any time, and the total number of distance features was 10. We used these 10 distance features and the y coordinate of the player to characterize each state, as is shown in figure 2. Note that we did not use the x coordinate as feature since the game was a symmetric system where we did not expect the value function to linearly increase or decrease with the x coordinate of any element. The state vector can be transformed from the x and y coordinates of all current platforms, the player and possible bonus, which were returned by the simulators.

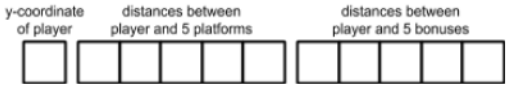

Figure 2 State vector

In the deterministic model of continuous-state MDP, the value function was updated as below:

$$V(s) = R(s) + \gamma max_a V(s^{'})ds^{'} \quad (1)$$

where $s^{'}$ was the next state followed by s after a certain action.

In Fitted Value Iteration, we used linear regression to approximate the value function as a linear function of the states:

$$V(s) = \theta^T s \quad (2)$$

where $\theta \in R^{12}$, since we added an intercept term θ_0. Our targets in the algorithm were to optimize θ to fit the V(s) and choose the action at each step according to V(s). The training samples for fitting θ were collected by recording every state during the simulation and putting them into a collection called "replay memory". For each step, we randomly sampled a batch of states from "replay memory" and implemented stochastic minibatch gradient descent. The algorithm is as follow:

1. Initialize θ
2. Repeat {
   get state for simulator F: $s_t = F(s_{t-1})$
   // Choose action
   if in observation period:
      With probability ε choose action randomly.
   else:
      for i = 0 : 2
         use simulator G to get $G(s_t, a_i)$.
      end
      choose action $a = \text{argmax}\, V(G(s_t, a_i))$.
   // Accumulate experience
   put (s_t, r_t, a_t, s_{t+1}) into the replay memory D.
   // Update V(s) (only after observation period)
   randomly sample minibatch from D (batch size = m).
   For j = 1 : m
      use simulator G to get $G(s_j, a_i)$
      set $y_j = R(s_j) + \gamma maxV(G(s_j, a_i))$
   end
   // Gradient descent
   $\theta_i := \theta_i - \alpha \sum_{j=1}^{m} (\theta^T s_j - y_j) s_{ji}$
}

The observation period was used for accumulating sufficient training samples in the replay memory. In our settings, the observation period was 13,000 steps, the capacity of replay memory was 50,000, the batch size was 50 and the γ was 0.99. The reward of death was -100, the reward of eating bonus was +150, and the reward of descent was +0.2 (corresponding to the score increasing rule of the game). We expected the player to live longer instead of killing itself very soon, we also set the reward of keeping alive to be +0.1. In order to ensure adequate exploration of the state space, we also implemented ε greedy approach to randomly pick an action with probability ε, and anneal ε from 0.3 to 0.001 during the training process.

C. Deep Q-Learning

In deep Q-learning, we expected to build an end-to-end learning algorithm, where the agent could only get the same information as what a human player could get during the game. They were the UI image at each state, the flag of game over as well as the flag of scoring. Thus the input states would be high dimensional images, instead of hand-crafted backstage information, and a linear model could not be well fit. In terms of this reason, we resorted to deep Convolutional Neural Network (CNN) to approximate the value function. In this setting, the value function was a function of states and actions. It was called the state-action value function Q, and Q = Q(s, a). In this case, the action-choosing decision at each step was made as follow:

$$a_t = maxQ(s_t, a; \theta) \quad (3)$$

And the update of Q function was:

$$Q_t = r_t + \gamma maxQ(s_{t+1}, a_t; \theta) \quad (4)$$

Since it was more reasonable to make action decision based on not only the positions but also the velocities, we chose a sequence of images of continuous frames as the state. Despite of high dimensional states, the more complex fitted model as well as the form of value function, the deep Q-learning algorithm shared the same framework with classical reinforcement learning, including the setting of the replay

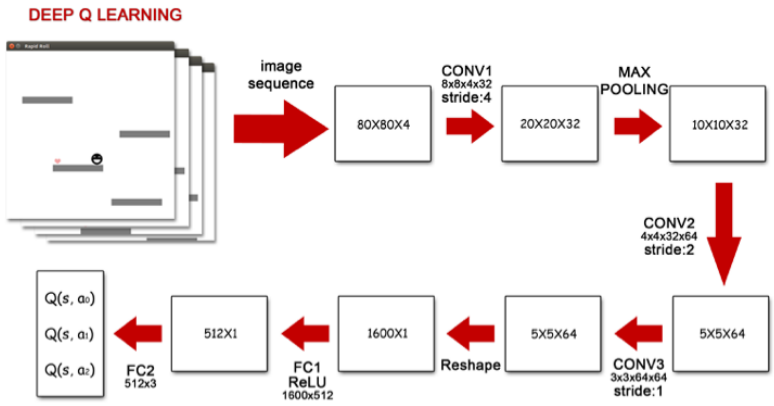

Figure 3 CNN Architecture

memory, greedily action choosing and gradient descent. The model was implemented with TensorFlow[4].

Pre-processing: We conducted pre-processing over the image pixels. At each step, we obtained an image from the simulator F. The original size of UI image was 500 x 500 pixels in three channels (RGB). We converted the picture captured into the grayscale format and rescaled it to 80 x 80. The 4 most recent frames were enqueued into the replay memory.

CNN architecture: The CNN in the learning algorithm contained 3 convolutional layers, 1 max pooling layer and 2 fully-connected layers in the end. At the end of the last convolutional layer, we could reshape the output from 5x5x64 array into 1600x1 array, keeping the total entry numbers. Note that we did not add the dropout layer into the framework since the algorithm continuously added new training samples into the replay memory, there would be no issue of overfitting. The architecture of the CNN is shown in Figure 3.

We executed the minibatch gradient descent with respect to the parameters of CNN. The pseudocode of Deep Q Learning algorithm is as follow:

1. Construct replay memory D and randomly initialize neural network.
2. Repeat {
  Initialize state by capturing initial image of the game.
  for t = 1 : T
    With probability ε (ε decayed in exploration period) choose random action a_t,
    otherwise choose $a_t = max\mathrm{Q}(s_t, a;\ \theta)$
  end
  Input a_t into the game simulator F and observe r_t and s_{t+1}.
  Store $(s_t,\ r_t,\ a_t,\ s_{t+1})$ into the replay memory D.
  if in exploration or training period:
    Sample a minibatch of $(s_j,\ r_j,\ a_j,\ s_{j+1})$ from the D.
    $y_j = r_j$ (if it is a terminal state)
    $y_j = r_j + \gamma max Q(s_{j+1}, a_j; \theta)$ (if it is not a terminal state)
    Perform a gradient descent step on $(y_j\text{-}Q(s_j,\ a_j;\ \theta))^2$ with respect to θ.
  end
end
}

During the training, we took a batch size of 32 and a replay memory capacity of 200,000. We took the Adam Optimizer for optimization and the learning rate was 10e-6. In the deep Q-learning algorithm, we also implemented ε-greedy approach for exploration.

IV. Results and Discussion

A video of playing Rapid Roll can be found at https://youtu.be/trBjzhb0oiA. The left side is the agent trained with the Deep Reinforcement Learning while the right side is human playing. We used the average game score over 20 games to evaluate the performance.

A. *Overall performance*

After training 3,400,000 iterations, the performance of the agent with deep Q-network (DQN) has already been much better than humans, and also better than the results of the classical Fitted Value Iteration (FVI). The comparison of random selected actions, human player, FVI and DQN is shown in Table 1. Note that the game levels were different in the ascending velocity of the platforms: 5 pixels/frames for a hard level and 3pixels/frames for an easy level.

Game level	Random	Human	FVI	DQN
easy	163.7	Inf	577.7	Inf
hard	102.1	314.7	529.8	Inf

Table 1. Average score of random actions, human player, fitted value iteration (FVI) Agent and deep Q-network (DQN) Agent at two different game difficulty level

The performance of DQN was much better than human while the performance of FVI was not so good. If the score was larger than 5000, it would be regarded to be infinity. In terms of an easy level, the scores of human and DQN were both infinity. But DQN was considered to be better because human could not

keep playing without taking a rest. The score of FVI was only at a level of hundreds, but still much higher than random actions and a bit higher than human player. In terms of a hard level, the score of human was very low while the score of DQN was still infinity and the score of FVI was still at a level of hundreds. This meant that the DQN worked very well especially when the game difficulty increased. The performance of FVI was far poorer than DQN since its features and value function were too simple to model the complex processes of this game. Although we provided DQN with nothing but the raw pixels of the game interface, it performed better than FVI which was provided with hand-crafted features. This showed that DQN was a powerful end-to-end learning framework which could learn the high-level state of a black box, just like human learning how to play a game from scratch.

B. *Training steps*

In this part, we discussed how the performance of DQN was influenced by the number of training iterations. The result is shown in Figure 4 that with more training steps, DQN can get higher scores (hard level). Figure 5 shows that the loss function gets lower with more training iterations. The change of loss matches with the change of scores.

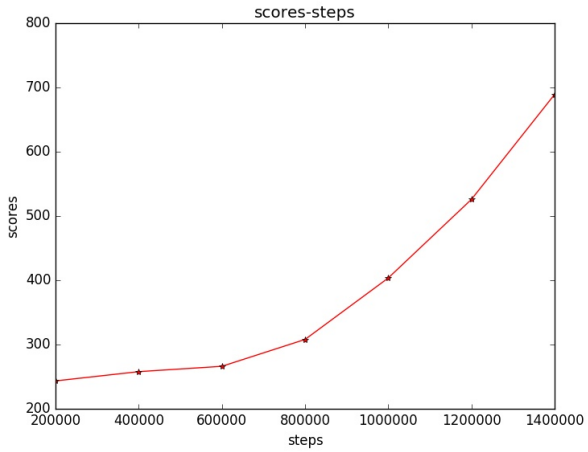

Figure 4 Score with training steps

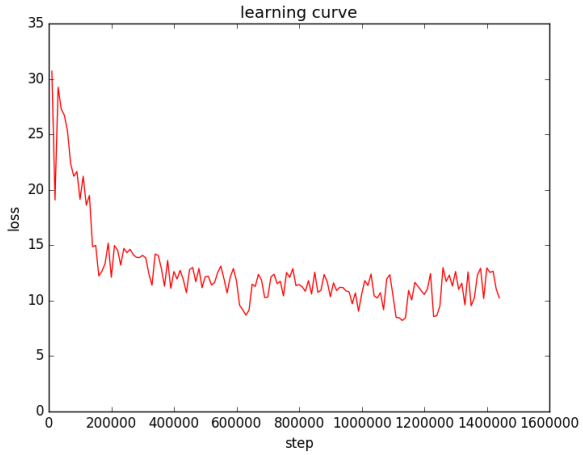

Figure 5 Learning curve

C. *Adding the reward of eating bonus*

In this part, we tested that whether adding reward of bonus could influence the training speed.

Training iterations	BonusReward=150		BonusReward=0	
	Easy	Hard	Easy	Hard
600,000	271.2	261.3	297.4	254.0
1000,000	406.7	400.6	430.4	299.2
1400,000	702.6	695.4	456.7	443.6

Table 2. Average score of Deep Q Learning (DQL) Agent with and without reward of bonus at two different game difficulty level

According to Table 2, after adding reward of bonus, the score of DQN agent is almost always higher than without reward of bonus . This shows that adding reward of bonus can make the agent eat the bonus more actively and stay alive longer.

D. *Adding human playing to replay memory*

In this part, we expected to test that whether adding human playing experience to the replay memory could expedite the training. We trained another model for 1,000,000 steps. For every 200,000 steps, we added 20,000 human-playing frames into the replay memory to replace random exploration frames by having human controlled the game simulator. The scores versus steps curve is shown in Figure 6. The results showed that the DQN with the human-playing frames can always get higher scores. It proved that adding human playing to replay memory could accelerate training process. This was because that human playing process could accumulate much more different states than random choosing actions in a short time. During the training, we noticed that in the observation and exploration period when the agent choosing action randomly with probability ε, the agent player was always moving back and forward on a single platform and collided with the ceiling quickly. Obviously, it was very slow to explore many other states, such as the states where the agent player was loading on the platforms at the bottom. In contrast, with human interference, it could efficiently collect more states like this and add them into the replay memory, and thus speed up the training.

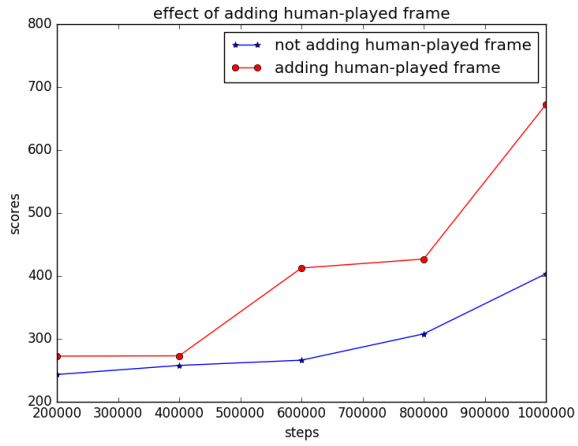

Figure 6 Effect of adding human-played frames

Furthermore, we noticed that if adding human-playing frames into the replay memory, there would be less

probabilities for the agent player to stay in the rightmost or leftmost corner of platforms and be waiting to be killed by the ceiling. This was because that we showed the agent a large number of "correct" actions under certain situations which could yield to larger rewards by adding human-playing frames. In comparison, if the agent tended to randomly observe and explore at the very beginning, it might never know that moving out of the rightmost or leftmost corners of platforms could yield to larger rewards, since there were few such states in the replay memory, i.e, the model reached a local optimum. Therefore, human-playing exploration might be a better exploration method, ε greedy random exploration, which could speed up the training and reduce the possibility of local convergence.

V. Conclusion

In our project, the agent with the Deep Reinforcement Learning outperformed that of the classical Reinforcement Learning as well as a real human player in Rapid Roll. This showed the power of the Deep Reinforcement Learning in learning high dimension features and fitting complex models. Moreover, it provided an end-to-end learning framework to directly learn raw pixels without any hand-crafted features, which could be applied in a wide range of MDP-based problems without specific modifications. Adding human-playing frames into the replay memory could efficiently accumulate more possible states for training and thus enhance the training speed, which indicated that adding "supervised" process into the exploration period might yield to better performance. However, in rapid roll and other video games, some states should be more important than others in decision-making (such as the states when the player was close to the ceiling). Therefore, we could assign more weights (for example, take cost-sensitive trick) to these states during the training process to expedite the training. This might be one of the promising directions of future work in the Deep Reinforcement Learning research field.

References

[1] Mnih, Volodymyr, et al., "Human-level control through deep reinforcement learning," Nature 518.7540(2015):529-533.

[2] Mnih, Volodymyr, et al., "Playing atari with deep reinforcement learning," arXiv preprint arXiv:1312.5602(2013).

[3] Kevin Chen, "Deep Reinforcement Learning for Flappy Bird".

[4] Google. Inc, TensorFlow: An open-source software library for Machine Intelligence.