Applying Deep Reinforcement learning to Sonic the Hedgehog 2

Eyþór Einarsson
Háskóli Íslands, Rafmagns- og tölvuverkfræði

30 November 2018

Abstract

Many successful implementations of reinforcement learning have shown to be able to achieve human level performance when applied to video games. However many of the published results using state-of the art methods require a large amount of training time, often spanning many days. When trying out different methods on a problem it can be quite limiting if days are required to get results from experiments and limits the number of hyperparameters that can be tested on a given method. Here it is shown that it is possible to train a DQN network to perform on human level in a single level in a video game with just hours of training time.

1 Introduction

Programming a computer to play a game has been a research topic ever since the emergence of the first computers in the late 1940s. Claude Shannon, contemplated programming a computer to play chess in a paper puplished in march 1950.[8]. It took some time before computers could perform on human level and in 1997 an IBM computer system called Deep Blue first beat the world champion in the game of chess.[5] Since Deep Blue many other games have had a computer perform better than humans and just recently the program AlphaGo developed by DeepMind beat the world No.1 ranked player at the time in the game of Go.[9]

Reinforcement learning has successfully been used by reasearchers at DeepMind to learn to control an agent in an video game environment with human level performance using only the score of the game and the high-dimensional video output of the game as input, this is possible by the use of recent advances in deep learning which have made breakthroughs in fields such as computer vision. The method used they call deep Q-learning and was implemented on multiple games for the Atari 2600 system which is a home video game console realesed in 1977.[7] Since, many improvements have been proposed that enhance the speed and stability of the deep Q-learning. Some of those improvements are in an improved algorithm named Rainbow which was presented and the performance of those improvements discussed.[3]

However the methods discussed require very large computational time, for example the Rainbow method can take up to 10 days of training for the 200 million frames used in the original work on a computer with a Tesla P100 GPU.[4] Here the methods discussed will be implemented on the video game Sonic the Hedgehog 2 released in 1992 and it will be discussed if it is feasible to implement reinforcement learning methods with less training time and still get human level performance.

2 Methods

The goal of this work is to implement an reinforcement learning algorithm on the video game Sonic the Hedgehog 2. To get a programmable interface to this game a wrapper for video game emulators called Gym Retro is used. Gym Retro is a python package which extends on OpenAI Gym[1] a toolkit for reinforcement learning research. This gives us a platform for applying the reinforcement learning algorithms on the game while focusing on the algorithm itself and not the code for controlling the game.

2.1 Background

We consider a basic reinforcement learning setting where the agent interacts with an environment over discrete time steps t. At every step t the agent chooses an action according to some policy $\pi(a|s)$ which gives the probability that action a is taken if in some state s. When the agent performs some action A from the possible actions it recieves a scalar reward R_t and the next state s_t. The goal of the agent is then to maximize the return $G_t = \sum_{k=0}^{T} \gamma^k R_{t+k}$ where $\gamma \in (0, 1]$ is a discount factor. That is the goal of the agent is to maximize the discounted future cumulative reward from each time step t until some terminal state T.

The value function $v^\pi(s) = \mathbb{E}[R_t|s_t = s, a]$ is the expected reward following policy π from state s and the action-value function $Q^\pi(s, a) = \mathbb{E}[G_t|s_t = s, a]$ is the expected return for selecting action a in state s and then following policy π. In the value-based reinforcement learning methods implemented here the action-value function is estimated with a deep artificial neural network and the policy is purely based on that value function, in other methods the policy $\pi(a|s)$ can also be represented by a neural network which results in a policy based algorithm.

2.2 Sonic environment

In this work the environment is the game Sonic the Hedgehog 2 where the state will be represented by frames from the video output of the game with some simple preprocessing involved. The possible actions in the game are the buttons on a controller a person could press {up, down, left, right, A , B, C}. To make the possible actions discrete and keep them as small as possible a combination of a subset of those actions is used that should provide the agent with all required action for winning the game, 9 possible actions where chosen

$$[\{\downarrow\}, \{\leftarrow\}, \{\rightarrow\}, \{A\}, \{\downarrow, A\}, \{\leftarrow, A\}, \{\rightarrow, A\}, \{\downarrow, \leftarrow\}, \{\downarrow, \rightarrow\}]$$

Commonly the reward r_t in video game reinforcement learning settings is in relation to the score of the game, here we will use the x-position of the agents character in the game. Specifically the reward will be how many pixels the agent has progressed forward in the horizontal direction since the last state. This is done because in order to progress forward in this game the score is irrelevant and using this reward gives the agent more information on how it is performing. The goal of the agent is to finish the episode as fast as it can and to introduce this into the reward function we add a negative reward r_{step} for each time step. The reward for time step t is then

$$r_t = \max(\textit{current x position} - \textit{last x position}, 0) - r_{step} \tag{1}$$

The max is introduced since we do not want to punish (give negative reward) if the agent backtracks and it x position is less than before.

While the goal is to learn to play the game purely from the frames of the video game and the reward from each time step some preprocessing of the video frames is done before feeding them into a neural network. The frames are converted to gray-scale, down-sample by factor 2x and then the final state representation will be the weighted average of the current processed frame f_t and the frame at time $f_{t-\Delta t}$ where $\Delta t = 5$ the final state representation is then $s_t = w^- \cdot f_{t-\Delta t} + w \cdot f_t$ where $w^- = \frac{1}{3}$ and $w = \frac{2}{3}$. Using the average of two frames should give a state representation which includes some information about the motion in the game and down-sampling the frame and converting it to gray-scale should save some computation time when propagating it through a neural network. Finally the state representation is scaled to take on values in the range -1 to 1.

2.3 Algorithm

The algorithm will be tested on the first level of the game. The first level is a relatively easy level compared to other levels in the game in the sense that it is the most linear and requires the least backtracking. That is holding the left controller button and occasionally pressing jump is a good policy for this level of the game, this should give us a good starting point for testing the reinforcement learning algorithm with different parameters and modifications on the environment.

2.3.1 Convolutional neural network

The algorithms implemented requires a function approximator for approximating $Q(s, a)$, for that a convolutional neural network was used with the architecture shown in table 1.

Input:	Image of size (with,height,channels) $= (112, 160, 1)$
Layer 1:	conv2D, filters $= 32$, kernel size $= (8, 8)$, stride $= 4$
Layer 2:	conv2D, filters $= 64$, kernel size $= (4, 4)$, stride $= 2$
Layer 3:	conv2D, filters $= 64$, kernel size $= (3, 3)$, stride $= 1$
Layer 4:	FC, number of neurons $= 512$
Output layer:	FC, number of neurons $=$ number of actions $= 9$

Table 1: The CNN used for aproximating $Q(s, a)$.

The activation used for all layers was the ReLU activation function except for the output layer which has a linear activation, *same* padding was used for all convolutions and for optimization the momentum optimizer was used. The input to the network is the state s and the output of the network is used as estimations for the Q values for each of the 9 possible action. Thus the output of the network represents a vector $Q(s) = [Q(s, a_1), Q(s, a_2), \cdots, Q(s, a_9)]$.

2.3.2 Deep Q-learning

The algorithm used is the DQN descriped in [6], this is a Q-learning (action-value based) method with a neural network estimating the Q state-action values. The implementiotion used is based on tensorflow code given in [2]. The Deep Q-learning uses two networks for estimating the Q-values, one is used for choosing the action a the agent takes and another is used only in the target value for the neural network representing $Q(s, a)$, the other network $\hat{Q}(s, a)$ is not updated with gradient descent but is a copy of $Q(s, a)$ made after a predetermined gradient descent update steps C. This introduces more stability by reducing the variance of the update method described in [7]. To update the Q network with gradient descent a replay memory is used, this makes it possible to update the weights using supervised learning methods. The update to the Q network is then only done in a predetermined number of steps s_{update}. To choose an action a_t at time step t the epsilon-greedy method is used, epsilon-greedy selects action a_t by choosing a uniform-random action a from all possible actions with probability ϵ, else it chooses the action which gives the highest Q-value.

$$\texttt{epsilon greedy}(Q(s_t, a), \epsilon) = \begin{cases} \texttt{random action}, & \texttt{with propability } \epsilon \\ \arg\max_a(Q(s_t, a)) & else \end{cases} \tag{2}$$

Epsilon decay is also used which makes ϵ smaller over time during training until it reach some minimum value ϵ_{min}, this ensures that the agent performs less and less random actions as he learns to behave in the environment. Specifically ϵ was decayed linearly from 1 (100% random actions) to ϵ_{min} in predetermined training steps ϵ_{steps}. The full algorithm is given in Algorithm 1.

Algorithm 1 Deep Q-learning with Experience Replay

D is the Replay Memory
$Q(s,a)$ is the action-value network
$\hat{Q}(s,a)$ is the target action-value function

$\theta_Q \leftarrow$ `Inital weights`
$\theta_{\hat{Q}} \leftarrow$ `Inital weights`
$D \leftarrow$ empty list of tuples (s,a,r,s') of size N
for every episode **do**
  $s_1 \leftarrow$ `first state`
  for every time-step t in episode until terminal state $(t = 1,T)$ **do**
    $a_t =$ `epsilon greedy`$(Q(s_t,a),\epsilon)$
    $s_{t+1}, r_t \leftarrow$ `State and reward from taking action` a_t
    Store (s_t,a_t,r_t,s_{t+1}) in Replay Memory
    if s_{update} steps have past **then**
      Sample random minibatch of (s_j,a_j,r_j,s'_j) from D
      $y_j \leftarrow \begin{cases} r, & \texttt{if } s'_j \texttt{ is terminal state} \\ r_j + \gamma \max_a \hat{Q}(s'_j,a) & \texttt{else} \end{cases}$
      Perform gradient descent step on $(y_j - Q(s_j,a_j))^2$
      After C gradient descent steps $\theta_{\hat{Q}} \leftarrow \theta_Q$
    end if
    decay ϵ if $\epsilon > \epsilon_{min}$
  end for
end for

3 Results

3.1 Experimental results

Three main training runs were completed over which the method was refined, the resulting performance and modifications to the method are decriped here. All training in this section where done on a system with a Nvidia GTX 780 GPU and AMD Ryzen 1600 CPU.

First training run The Deep Q-learning method was trained over 10 million training steps which resulted in about 6 days of training. The resulting agent did not come close to completing a single episode and resulted in an agent which could get around 1000 in cumulative reward per episode. For comparison a good human player can finish the level in under 30 seconds which results in cumulative reward of over 9000. The parameters not yet specified used in this run are shown in table 2.

Optimizer learning rate	0.001
Optimizer momentum	0.95
Minibatch size	50
Size of replay memory	$N = 10.000$
Copy Q to $\hat{Q}$ steps	$C = 1000$
Steps before GD update	$s_{update} = 4$
Discount factor	$\gamma = 0.99$
Minimal random action	$\epsilon_{min} = 0.01$
Step to achieve ϵ_{min}	$\epsilon_{steps} = 2.000.000$
Extra reward per time step	$r_{step} = -1$

Table 2: Parameters used for the first run

Second training run The results of the first run did not seem promising so a couple of modifications were made to the environment. First since the agent is trying to decide what action to take every frame this corresponds to trying to choose a action 60 times per second or approximately every 17 ms of game time. Human response time is definitely greater than these 17 ms and accordingly it should be possible to play the game on a human level without

this fine control. The agent was limited to seeing 3 frames out of 60 per second and would play the same chosen action over the next 19 unseen frames, this makes the agent only have to choose 3 actions for every second of game time. To accommodate this modification the reward the agent sees needs to be modified, the reward was set to be the average reward over the 20 frames over which a particular action was repeated.

Another observation is that the goal is to finish the level as quickly as possible, therefor a limit could be set on how long a episode can last and if set reasonably should give the agent enough time to finish the level but cut him off if he spends too much time in one episode. There is a 10 minute time limit for a player in this game gets to complete a level and therefore sets a limit on how long a episode for the agent lasts but since it is possible to finish the level in much shorter time, the lower time limit was set to be 90 seconds of game time. This modification does give the agent more than enough time to finish the level and should give him more time to learn what actions are best to achieve in this limited time and could prevent him to getting stuck in a particular portion of the level for too long. Since some effort was made in tuning the hyper-parameters in the previous run most parameters where kept the same, these modifications and the changed parameters can be seen in table 3.

Minimal random action	$\epsilon_{min} = 0.05$
Step to achieve ϵ_{min}	$\epsilon_{steps} = 1.500.000$
max episode duration	90 seconds
Action repetition	20 frames

Table 3: Parameters used for the second run

With these modifications to the environment the agent was trained over 2.5 million training steps which resulted in 17500 played episodes and took about 3 days of training. After these modifications the agent learns much faster to progress through the level and after 2.5 million training steps the agent is able to finish the level in some episodes played although in most runs it only comes close. Figure 1 shows the cumulative reward of full episodes which is averaged over the last 500 episodes played.

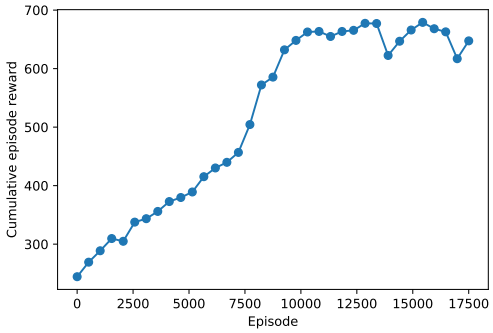

Figure 1: Training progress for the second run, datapoint correspond to the episode cumulative evaluation reward averaged over the last 500 episodes.

The reason the agent does not achieve to finish a level is that it seems it doesn't want to, the agent gets arbitrarily close to the end then it just stops and starts doing something else rather than proceeding to finish the level. This likable stems from the agent not being incentivised to finish the level and is most likely do to the the agent being able to achieve higher rewards if it does not proceed to finish the level.

Third training run To accommodate for the agent not being incentivised to finish the level the reward for each time step r_{step} was decreased from -1 to -2 and the agent retrained. After some time of training it was observed that the agent was learning faster than in previous runs but was performing obvious random actions which seemed to hold back progress, to decrease those random actions ϵ_{steps} was lowered to 100.000 steps which lowers the number of steps required before the minimum rate of random actions ϵ_{min} is

achieved. The agent was trained for 200.000 training steps which resulted in 4000 played episodes and took 7,5 hours of training. This agent learned a lot faster and after only 4000 played episodes learned to consistently finish the level. The avarage cumulitave reward per episode during training can be seen in figure 2. Again the reward function has changed so a direct comparison of the rewards between runs is not possible.

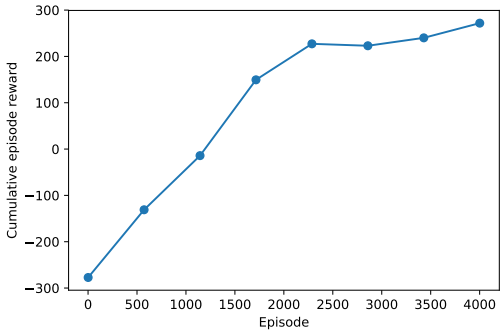

Figure 2: Training progress for the third run, data-points correspond to episodes cumulative reward averaged over the last 500 episodes.

3.2 Comparison

Since the goal of the agent is to maximize the reward it is appropriate to keep track of the reward function but since the rewards are not the same between runs an alternative reward function only used for evaluation was constructed and kept the same between all training runs. The evaluation reward function is the same reward function as laid out in equation (1) but instead of taking the average over the frames skipped, as done for training, it was recorded for every frame. This results in the evaluation reward function

$$r_{frame} = \max(\textit{current frame x position} - \textit{last frame x position}, 0) - r_{step} \tag{3}$$

where $r_{step} = 2$ was set to be the same as in the third training run. Here we will test out different parameters for the learning method comparable to the final run of the previous section. All tests in this section where run on a Nvidia GTX 2080 ti GPU and a AMD Ryzen 1600 CPU.

In previous sections the batch size used in training have been kept constant of 50 training examples per mini-batch. To find out how much affect the batch size has on training the agent was trained with parameters described in the third training run of the previous section but with a batch size of both 50 and 256. Also different values of the epsilon decay parameter ϵ_{steps} was used in the last section and in figures 1 and 2 it looks like training stops in approximatly the middle of the run which is the time that ϵ reaches the minimum value and to verify that the value of ϵ_{steps} has been chosen reasonably, a longer run with a higher value of ϵ_{steps} was also performed. The cumulative evaluation reward per episode was recorded for the three runs described here and can be seen in figure 3, for a human level performance a run performed by two human players is used. Parameters used can be seen in table 4 along with the training time.

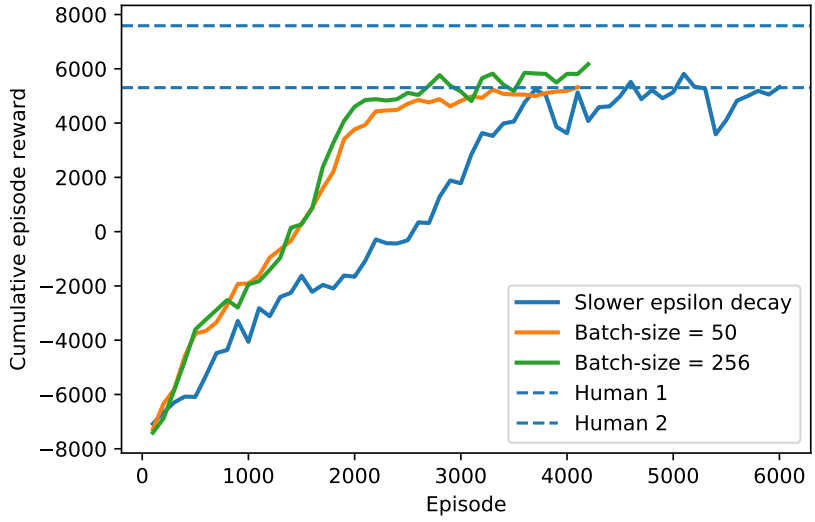

Figure 3: Training progress for different batch sizes and a longer training run with a slower epsilon decay (higher value of ϵ_{steps}). The figure shows episode cumulative reward averaged over the last 100 episodes of training.

Shared parameters			
Optimizer learning rate	0.001		
Optimizer momentum	0.95		
Size of replay memory	$N = 10.000$		
Copy Q to $\hat{Q}$ steps	$C = 1000$		
Steps before GD update	$s_{update} = 4$		
Discount factor	$\gamma = 0.99$		
Extra reward per time step	$r_{step} = -2$		
max episode duration	90 seconds		
Action repetition	20 frames		
Minimal random action	$\epsilon_{min} = 0.01$		
Results			
Training steps	200.000		300.000
ϵ_{steps}	100.000		200.000
Minibatch size	50	256	50
Episode played	4100	4200	6100
Training time	4h:26m	6h:13m	7h:2m

Table 4: Comparison between training with different parameters

Using a larger batch size does not seem to provide much difference in the performance as can be seen in figure 3 but does result in a slower training. Setting ϵ_{steps} to higher step number does not result in the agent being able to learn a better policy and seems to have the only effect of increasing the number steps required before it converges to a similar performance as the one with lower ϵ_{steps}.

All parameters tested here result in an agent that achieves performance comparable with human-level performance and the fastest one in about 4,5 hours of training with the RTX 2080 ti GPU. In the previous section it can be seen that the training time for the same number of training step on the lower end GTX 780 GPU is 7,5 hours.

4 Discussion

It has been verified that it is feasible to implement reinforcement learning method and get human-level performance with under 5 hours of training on the first level of the video game Sonic the Hedgehog 2. The biggest challenge in training a agent is the parameter search and how the reward and the environment is defined. The number one factor in decreasing the training time is as one would expect is to decrease the number of actions the agents has to choose. A correct choice of reward function and balancing the exploration/exploitation dilemma in reinforcement learning is still a big challenge as has been long known and when reinforcement learning methods are combined with deep learning methods the parameters that require tuning have a quite large search space. Having a short training time in a interesting environment can be a big help when tuning hyper-parameters for a reinforcement learning algorithm.

For applying reinforcement learning method on a video game it seems reasonable to limit the number of actions the agent has to take to a similar number of actions a human is able to think about and choose, that is to limit the agent to a response time similar to that of a human since video games are made to be controlled by a human and also since the performance of the agent is compared to human performance.

It remains to be seen how much the training time scales with harder game environments and if the agent trained here can achieve human performance on a more complex game levels in a similar reasonable training time. The code used in this project is available in the GitHub archive https://github.com/EythorE/sonic2_reinforcement_learning.

References

[1] Greg Brockman et al. "Openai gym". In: *arXiv preprint arXiv:1606.01540* (2016).

[2] Aurélien Géron. "Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems". In: " O'Reilly Media, Inc.", 2017, pp. 462–470.

[3] Matteo Hessel et al. "Rainbow: Combining improvements in deep reinforcement learning". In: *arXiv preprint arXiv:1710.02298* (2017).

[4] Dan Horgan et al. "Distributed prioritized experience replay". In: *arXiv preprint arXiv:1803.00933* (2018).

[5] Feng-hsiung Hsu. "IBM's deep blue chess grandmaster chips". In: *IEEE Micro* 19.2 (1999), pp. 70–81.

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

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

[8] Claude E Shannon. "XXII. Programming a computer for playing chess". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41.314 (1950), pp. 256–275.

[9] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550.7676 (2017), p. 354.