Creating a Robust Algorithm for Reinforcement Learning on Atari Games

Jan Kudelka, Thibault Douzon, Lioutas George and Robin van Soelen
January 2019

Abstract

This study is about founding a way to create a robust reinforcement learning algorithm that can be applied to a wide range of different Atari 2600 games. This is being done using the RAM of the game. Both an NN and CNN model are being tested on three different games: Breakout, Seaquest and Ms. Pacman. The results indicate that learning using these models works well, despite the too short amount of training per game and the fact that game specific features were not extracted.

Introduction

Reinforcement learning hasu become a very big subject within the field of machine learning. This is because reinforcement learning works by getting data directly from the environment, making it possible for the dataset to learn on to become infinitely large. This allows for a lot of opportunities and interesting applications.

One of the most popular usages of RL is the application on games. For each game there is a limited amount actions, and features to be extracted. This makes it possible for an agent to accurately learn how to play a certain game. When looking at applications of this concept it becomes clear that RL algorithms are often tailored for a specific game, by extracting specific features for this game. However, the framework created by openAI gym provides a way to apply reinforcement learning to a wide range of Atari games. This works by either applying Convolutional Neural Networks to the screen as input, or by applying Neural Networks to the RAM of the game. The RAM of the game consists of 128 bytes which is filled with information about the current game state, like for example, the positions of characters or amount of life left. One disposition of the RAM represents one unique state of the game and any (small) change in the game induce (small) change in the RAM.

Because this framework makes it so easy to test different games with the same algorithm, the goal of this project will be to create a robust algorithm that is able to learn

a different variety of games with different complexities, by only using the RAM of the game.

To test the robustness of the algorithm, three different games of different complexities will be used to test it on. These games are breakout, seaquest and ms. pacman (fig 1). The goal of breakout is to bounce a ball into bricks to coloured bricks to make them disappear, the game is lost when the ball hits the bottom of the screen. The goal of seaquest is to avoid enemies and shoot at them and to save drowning people. There is also an oxygen tank which has to be filled regularly. The goal of ms pacman is to collect coins and avoid ghosts. When collecting a big coin it is possible to eat the ghosts to gain more points.

Therefore the question that will be tried to answer in this report, is how to design an algorithm that is able to perform well on these games using reinforcement learning.

Fig 1A: Breakout

Fig 1B: Seaquest

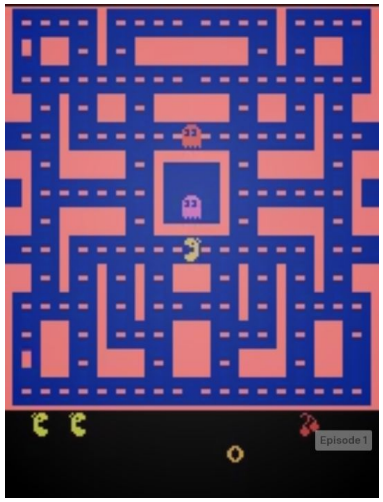

Fig 1C: Ms. pacman

Background

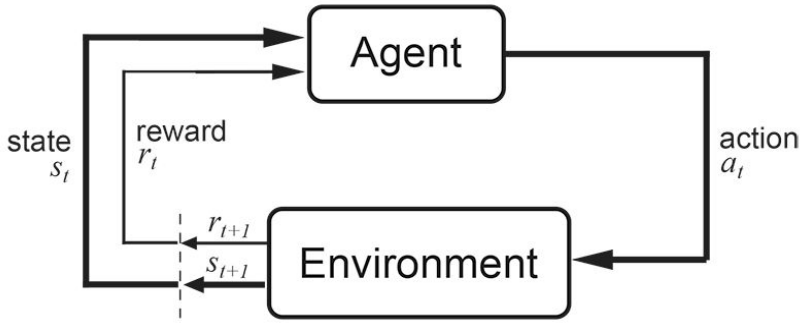

Figure 2: Reinforcement learning explained

Reinforcement learning works by letting an agent choose certain actions with the biggest reward based on its environment. Which actions are chosen is dependent on the so called Q-value. The Q-value is the expected reward, given a certain state and action. This Q-value is multiplied by gamma, which is a factor that makes sure that rewards which can be received earlier will get a higher reward. Where the first possible reward will get a gamma of 1 and for the rewards after this value will exponentially decrease. Therefore the best possible strategy will look like:

$$Q(s_t,a_t)=r_t+\text{gamma}\times\max_a Q(s_{t+1},a)$$

This method is often too intensive too calculate, since it requires to have visited all states in order to make a decision. By applying convolutional neural networks, the value of Q(s,a) can be estimated, by looking at the transitions between states instead of the states itself. This is called deep q learning. Using deep-q learning the error can be calculated by:

$$\text{error}=(Q(s_t,a_t)-(r_t+\text{gamma}\times\max_a Q(s_{t+1},a)))^2$$

Related work

One of the earliest applications of RL is td-gammon [1]. td-gammon is an RL algorithm made in 1995 that made it possible for a computer to reach levels of backgammon beyond of what humans are capable of, which used a technique similar to q-learning.

Nowadays the primary focus in machine learning research is on deep q-learning. The most similar project to this one is creating a robust algorithm for atari games, using the output of the screen and deep q-learning [2]. The results of this algorithm varied a lot between games. For example, learning worked extremely well on simple games like breakout and pong where it outperformed humans by a lot, however during the more complicated games like q*bert and seaquest the performance got far off from the performance of a human. This is due to the difference in the amount of actions in both of the these games. For example, where breakout only has 3 different actions, Seaquest already has 18, including an oxygen tank which has to stay full, which creates an additional strategy to the game.

Errors from the deep q-learning algorithm could also be explained by the fact that this type of learning can be subject to the overestimation of certain actions. A study done by Hado van Hasselt , Arthur Guez, and David Silver proves that this overestimation has effect on the performance in certain cases [3]. They found that implementing double q-learning instead can improve this. Double q-learning makes use of double estimator, which makes sure that actions are occasionally underestimated as well.

Another interesting technique is deep recurrent learning which adds recurrency to a Deep Q-Network (DQN) by replacing the first post-convolutional fully-connected layer with a recurrent LSTM [4]. This technique integrates information through time allowing for a better performance during partial observations. This is specifically useful when observing the output of the screen, because of flickering for example.

Scientists at Google have compared DQN algorithm with human baseline on 49 different Atari games [5]. They used convolutional neural network working with the image of the game. Results show DQN learning efficiency varies a lot and some games are easier to learn for DQN because their action space is smaller or the rules are simpler.

Methodology

Creating the algorithm

In this study, a framework called openAI Gym is being used. This framework provides access to a collection of Atari games while having some built in functions, which can be used to apply reinforcement learning to these games.

```
state, reward, done = env.step(action)
```

This function takes a specific action as an input and gives information about the state of the game, the amount of reward this action resulted in and whether its time to reset the environment again.This step function represents one frame within the game.

The challenge is to create a model that will be able to choose the right action based on the reward and the game state. This model will be created using tensorflow.

The model will get at each frame, together with the reward, 128 bytes from the RAM as input. The RAM consists of information about the current game state which is for example the positions of characters or the amount of life left. The desired output will be a q-value for each possible action. This can be achieved by implementing deep q-learning in the model. After some exploration by referring to literature [2] and creating small experiments, a Neural Network with 4 hidden layers and 128 neurons seemed to be working well. To avoid overfitting that tends to happen in DQN's [3], a dropout layer is added to the network with a p value of 0.5 when learning and 0 when testing.

However after running some first tests it became clear that the network needs some more improvements in terms of strategy.

Epsilon-greedy

The model was due to finding a strategy that worked reasonably well and to sticking to that specific strategy while other strategies were not explored yet. This is why an epsilon greedy strategy has been implemented. Such a strategy forces the agent to sometimes choose a random action which allows it to explore the entire state space. These random actions occur an ε amount of time. This factor starts of during training

but is linearly or exponentially decreasing over time. During testing it has decreased entirely.

Memory and replay, Minibatch

Another problem that occured was the high correlation between frames. When just looking at the previous frame every time, it's not possible to figure out complex strategies. This is why a mini batch gradient descent has been implemented instead of stochastic gradient descent. The output (State, action, reward, next_state) of a certain amount of steps are saved in a random sequence and fed into the network, to decrease correlation.

When the agent plays, it stores the sequence of states encountered, action chosen and rewards acquired and performs a batch learning session at each frame of the game.

Frameskip

Frame skip is a method that repeatedly performs an action for a set amount of frames before selecting the next one. This operation results in reduced learning time and to a more human-like behaviour because it makes it impossible for the agent to develop inhuman reflexes. The amount of frames skipped differs between methods, the most common values are from 0 to 4 while the DQN model suggested values are 2-3.

The higher the frameskip value is set the quicker the episode is, however different games need different values for optimal training.

Testing

Two different network models have been designed to be tested against each other, which can be found in table x. These two models will be applied to the three games: Breakout, Seaquest and Ms. Pacman. To get these results into context, they will also be tested against the performance of an average human and the program running on random.

DQN model	A	B
Epsilon (min, max, decay)	1, 0.1, 500000 frames	1, 0.1, 500000 frames
Network design	4 hidden layers, 128 neurons, dropout 0.5	3 hidden layers: 2 convolutions-maxpool + 1 dense with 128 neurons, dropout 0.5
Learning rate	0.0001	0.0001
Minibatch size	32	32
Memory size	500000	500000

Table 1: Network model designs

Fully connected DQN working on the RAM of the game has already been used in previous research papers [6]. This will be a baseline to compare with a 1-dimension convolutional network. The idea behind the convolution is space locality in the RAM: objects with the same semantic are store in sequential indices in the RAM. For example an array of coordinates in memory would be stored continuously. Convoluting might help finding those patterns and better learn games involving those mechanics.

Results

	Breakout	**Seaquest**	**Ms. Pacman**
Random	2	60	
Human	46	420	10160
DQN A (after 25.000 episodes)	22	400	**Missing data**
DQN B (after 25.000 episodes)	2	360	**Missing data**

Table 2: Average performance on games.

This computation was running on virtual machines and took days to finish, we wish we could have run those algorithms for longer. It would have given better performance and more trustworthy results. Indeed we have experienced agents not improving for thousands of iterations and doubling their score in a few hundreds batches.

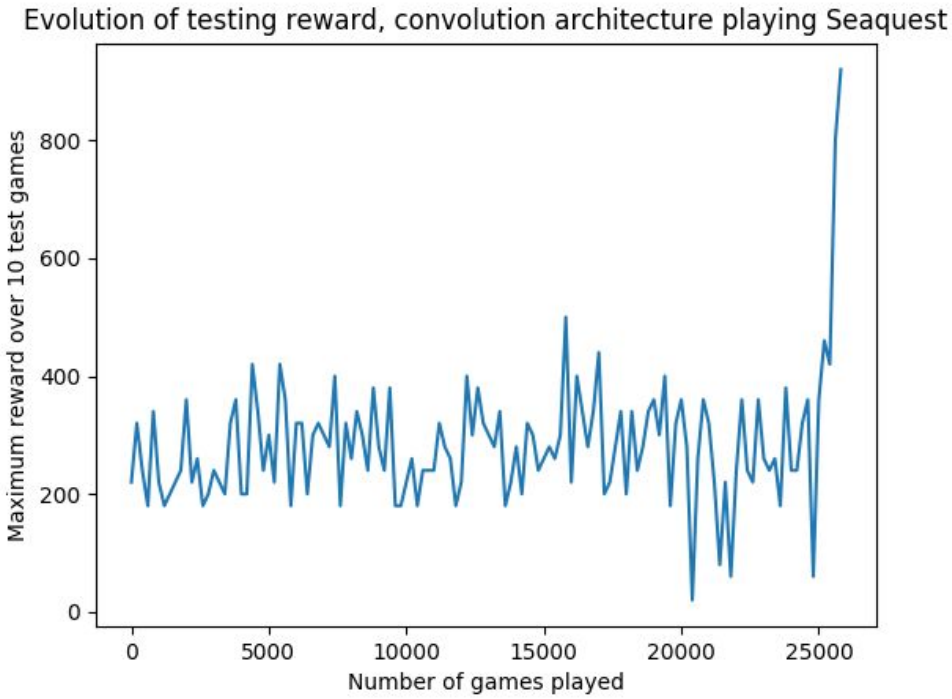

Figure 3: Testing rewards

For instance, fig(3) shows all the testing rewards of the DQN B agent on Seaquest. Every 200 iterations, 10 test games are played and the best score is saved as ‘maximum reward’. For nearly 24 thousands iterations this metric barely improved over time. But in the end, the agent started understanding the oxygen mechanics and thus could improve a lot its score because he was no more time constrained by the oxygen bar.

Those great improvements are mostly due to random space exploration and keep in mind it is possible for an agent to never explore the right direction.

Discussion and critical reflection

From the results in Table 2, it becomes clear that the Algorithm definitely learns something, but is still far off from the what a human is capable off. A big reason for this is the short amount of training done on the models. For the testing in this report, an amount of 25.000 episodes has been chosen, this is equivalent to about 24 hours of training. Due to time constraints this was the maximum amount of training per game possible. However it is still far off from the amount of episodes that related work used to train their models. One example which showed good results trained about 5.000.000 episodes per game [2]. This is also the amount of episodes where in this example the average reward would stop increasing. For future research it is therefore definitely necessary to train the model for longer.

Another reason for why the algorithm is not running perfectly is the fact that it is not retrieving specific features per game. This is intentional since the goal of the report is to create a generic algorithm that works on all games, instead of tweaking it specifically for one game. However if specific features are extracted, the performance will increase significantly. Especially for the complex games like Seaquest and Ms Pacman these features become more important. Since these games require multiple strategies for different parts of the game, these more game specific features could contribute to learning these. Another method that would increase the learning of different strategies is to slowly increase the complexity of the game. For Seaquest for example this would mean to initially remove the oxygen tank from the game until it the agent performs well enough and then add it. For Pacman this would mean to initially get rid of the part of the game where Pacman has to eat the ghosts, to make sure that it is able to avoid them first. However this concept would also interfere with the algorithm being reusable.

When comparing the 2 architectures, DQN B had a lot of trouble learning Breakout whereas DQN A learned quite well the mechanics of the game, the position of the ball and the platform. On Seaquest, DQN B did an impressive learning job, almost understanding the oxygen mechanic, allowing it to outperform DQN A when the game last longer. But when talking about mean score, DQN A is more consistent dealing with enemies and shooting.

Replay agents

We provide a small python script ‘review.py’ that allows anyone to replay a game and chose the agent that will play it. Few dependencies are needed: a Linux or Mac OS (VM are possible for windows users), python 3.6 preferably with the following libraries:

- Numpy, see here: https://www.scipy.org/scipylib/download.html
- Tensorflow, see here: https://www.tensorflow.org/install/
- Gym (with atari), see here: https://github.com/openai/gym

You will be able to run the script and chose the agent playing to see the differences between them. By default a agent playing random action is chosen but you can specify between 3 agents (1, 2, 3). All of them are convolutional agents, 1 has been trained for 2500 iteration, 2 for 10000 iterations and 3 is the final agent with the ‘-a’ argument.

All 3 seems similar until the oxygen bar is low, then agent 3 tries to surface and start again.

Conclusions

In conclusion, it seems possible to make a RL agent that can be used in a wide variety of games but there are a lot of issues that need to be dealt with. Starting with the parameters of the training, since different games have different optimal settings there is a need to find the values that satisfies the needs of most of them. Also, it was observed that the time needed for the training of these agents was long before they actually presented abilities that came close to the average human has. More testing needs to be done and more methods to be used, however the results look promising.

References

[1] Gerald Tesauro. Temporal difference learning and td-gammon. Communications of the ACM, 38(3):58–68, 1995.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. arXiv:1312.5602.

[3] Hado van Hasselt , Arthur Guez, and David Silver. Deep Reinforcement Learning with Double Q-Learning, Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)

[4] Matthew Hausknecht and Peter Stone. Deep Recurrent Q-Learning for Partially Observable MDPs. 2015

[5] Mnih V., Kavukcuoglu K., Silver D., Rusu A., Veness J., Bellemare M., Graves A., Riedmiller M., Fidjeland A., Ostrovski G., Petersen S., Beattie C., Sadik A., Antonoglou I., King H., Kumaran D., Wierstra D., Legg S. and Hassabis D. Human-level control through deep reinforcement learning. Nature volume 518, pages 529–533 (26 February 2015)

[6] Jakub Sygnowski and Henryk Michalewski, Learning from the memory of Atari 2600, arXiv:1605.01335v1 [cs.LG] 4 May 2016