REINFORCEMENT LEARNING: BREAKOUT

Ellen Vanhove (s162248) Emil Tyge (s123259) Carl Kjaergaard (s123075)

DTU Compute

ABSTRACT

Learning a computer to solve tasks by itself in a model free environment has long been a milestone in the research of general AI. Reinforcement learning (RL) is one promising approach for solving problems in this format, and with the rise of deep learning it has been shown that larger tasks can be tackled[1]. In this project we have implemented the deep-Q algorithm, and learned to play breakout and flappy bird.

***Index Terms*—** Deep Learning, Reinforcement Learning, DeepQ, Atari, Breakout, Flappybird

1. INTRODUCTION

The task of of learning is a complex task. Within machine learning different approaches has been taken, and with the exponential increase of computing power the last decades, large and deep neural nets have shown to solve many different problems.

Another branch of machine learning, reinforcement learning RL, lets computers learn inspired by human learning. With the notion of actions and rewards, the machine learns to take some action in order to maximize its reward, learning from actions taken earlier. In other words, learning from earlier experiences of itself. This approach has been shown to be able to learn complex problems, from only a very simple description of reward.

Fusing the two branches have produced some very interesting results for playing games where the action space is large and very complex. Examples on this is the game of Go [1] and playing Atari games [2].

Breakout is a classic Atari game, where one has to keep a ball in the air with a paddle while hitting boxes to score points. For RL based on screen input this game provides a variety of problems. A large input space, with a smaller underlying statespace. Hidden temporal movement, and rewards disjoint from the taken actions.

Therefore solution general to this game should be applicable to other games, where the observation space is screen input and the output is a limited set of discrete actions.

2. BACKGROUND

2.1. Reinforcement Learning

Reinforcement learning is the practice of training a system to perform actions based on a reward system. The program seeks to take actions, such that it maximizes a certain reward, which is usually designed by the developers of the system that the program is being trained on. Usually, reinforcement learning assumes that a Markov Decision Process (MDP) can be applied to a problem. The MDP assumes, that for every state, every possible action in that state has a reward associated after making a transition using the action. One problem with using MDP is that it creates an explosion of states and can only be made for smaller problems. On larger problems such as video games, the number of possible states is too large for modern machines to process efficiently. A lot of work in reinforcement learning is made to build systems that can either solve the MDP or approximate them using different methods.

We can then model the problem with breakout as an agent interaction with the Atari environment. At each time-step, the agent performs an actions $(a|a \in A)$ and receives a reward r where A is the set of legal actions. The states of the game, or our MDP, is the resulting screen output x of the game as is raw RGB pixels after each action.

One thing about the MDP is that it assumes full observability for every state x, but this is not the case for most games with raw screen input. Since the states might be reliant on previous screen inputs x, we construct a sequence of action inputs and resultant screen states $S = a_1, x_1, a_2, x_2, a_3, x_3$ which gives a total reward for the entire sequence of r. So long as this sequence S is long enough to include any hidden variables on the final state that any previous state might have, the problem can be said to be a fully observable MDP.

There are several approaches to reinforcement learning. Two different approaches that uses almost the same technique is Q-Learning and SARSA. In on-policy reinforcement learning we attempt to learn more about a policy $\pi(a|s)$ by following the policy and updating the policy based on itself. Q-Learning is an off-policy algorithm, we do not have to follow a policy. Instead of estimating the policy on itself, we perform an exploratory step where we attempt to learn something about our policy with an off-policy measure. This is also done in the Deepmind paper, where the current policy is estimated

against the ideal policy.

2.2. Neural Networks

Neural networks is an advanced nonlinear function approximator. Originally developed in the forties [3], as an ideal model of the neurons in the brain. Each neuron provides a linear combination of inputs and some nonlinear activation function. Each of the weights providing a weight parameter per input and a bias to be learned. The learning is done with stochastic gradient descent based, where the error is propagated backwards through the net, with an algorithm called back-propagation. Such that the output converges towards the training data labels. Using the neurons as base elements, a larger structure is built with layers of similar properties, for faster computation. As the information progresses towards the output each layer abstracts the information of the previous until the output layer is reached.

2.3. Deep Q-Learning

Q learning seeks to find a value heuristic for the rewards left in the game, so as to pick the action that maximizes it.

The Q function can be evaluated using many different methods. In classical reinforcement learning the Q function is estimated by linear models, but also non-linear models such as neural networks can be used. The Q function is then represented by a neural network, that takes as input the state and outputs the value $Q(a|s,\theta)$ where θ are the parameters for the network.

DeepMind Technologies proposed an algorithm for Deep Q-learning[2]. It uses experience replay where the latest transitions, i.e. state, action and reward, are stored performed by the emulator. The following steps are executed for a number of episodes. First initialise the emulator and perform a maximum number of steps or until the game is finished. Use a ε-greedy policy to select an action. This means with a probability ε a random action is chosen or otherwise the action with the highest Q value is used. Taking a random action sometimes guarantees exploration.

$$\pi(a|s,\varepsilon) = \begin{cases} \text{argmax}_a \ Q(s,a,\theta) & \varepsilon > rand \\ a_{rand} & otherwise \end{cases}$$

Store the transition s_n, a_n, r_n, s_{n+1} in replay memory. Sample a batch from the replay memory and perform a gradient decent step estimate the distance between the prediction and the actual received reward. This reward is calculated taking in account future rewards.

The goal is then to be able to minimize the distance between our current policy and goal policy, so that we are reaching the *optimal* policy.

$$R_n = r_n + \gamma \max_{a'} Q(s_{n+1}, \pi(s_{n+1}, a'), \theta^*)$$

$$\min_\theta \sum_n (R_n - Q(s_n, a_n, \theta))^2 + \alpha |R_n - Q(s_n, a_n, \theta)|$$

Storing transitions in memory has as advantages that a batch can be sampled such that there is no strong correlation between the samples. It is also more efficient, as a sample can be used multiple times.

2.4. n-step Q-Learning

N-step Q-learning is a method for making training easier in neural networks [4]. The n-step Q-learning enhances the normal Q-learning variant by performing a forward view on the for a maximum of t_{max} steps and calculating gradients, states and rewards for each of these steps, without changing the current exploration policy followed. Each state action pair encountered in each of these steps then uses the return that can be estimated after the step to calculate the gradients, so step n uses a return of step $t_{max} - n$ and so on. Finally all the gradients are calculated in a single update step.

3. IMPLEMENTATION

3.1. Breakout

Breakout is an old Atari game where the target is to remove a wall of blocks with a ball. Every time a block is removed a reward is received, and as time progresses the ball moves faster. In the bottom of the screen, see figure 1, you are equipped with a paddle, with which you can direct the ball at the blocks, and more importantly, stop the ball from falling out of the screen. The game ends when you have lost the ball five times.

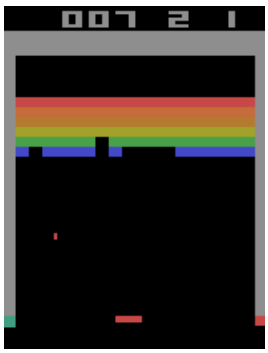


Fig. 1. Beakout

The game can be described with a deterministic Hidden Markov Model (HMM). The underlying state space describing the location of each of the blocks left in the wall, the position and velocity of the ball, the position of the paddle, the number of lives left, and the cummulated reward.

The observation space, is a 210x160 pixel RGB image. With these observations the whole state space except the velocity of the ball is observable.

The action space is a discrete set of button presses. At each timestep it is possible to choose one action among six; left, right, up, down, reset and no operation (NOP). The up and down actions, take the same step as the left and right actions, and are implemented due to the input method on the original Atari game

Rewards of $+1$ is given at each step where a ball removes a block in the wall. 0 reward is given otherwise.

3.2. Modeling

Due to the fact that the game state is not completely visible from an observation, it is a Hidden Markov model (HMM). The optimal action strategy cannot be learned from only a single frame. Therefore the input to the algorithm is extended with a temporal dimension, in order for the agent to observe the full action space, and transform the description such that the Markov property is held.

The action space contains two copies of the left and right command. In order for the network policy to converge faster, these are removed. Therefore the learned action space is four discrete actions. Left, right, reset and NOP.

The game is completely deterministic. Therefore the optimal action policy will be deterministic as well. The deterministic Q-learning policy based on the Bellman equation can also converge to the optimal strategy.

Q-learning searches to find the optimal value heuristic $Q(s, a)$, describing the expected future cumulative reward. The Q reward is discounted with a value $\gamma = 0.99$ because an action can have an effect far in the future when the ball reaches the wall. This choice results in exponential decay with a time constant of 100 frames. $Q(s, a)$ is implemented with a neural network with a preprocessed screen input that regularizes observations in shape, and each of the four actions as output. The choice of having a single network approximate all action value functions, is done to keep the computation time down, and more importantly reuse training across actions. With this setup the transformation from observation to state space is shared among all the actions. The network consists of two parts, inspired by neural network image recognition solutions. First a set of convolutional layers, filtering the image input. This type of layer provide spatial invariance. This is important in extracting the elements and their location relationship in different constellations. This helps with generalization, object positions not represented in the training set. Interlaced between some of the convolutional layers is max pooling layers, to reduce the computation time, and parameters to be learned, at the cost of the exact object location. After the convolutional network part a set of dense layers congregate the data into each of the action functions.

Different layer setups have been tried, but not tested extensively due to the long convergence time. The model seeming to provide somewhat fast convergence to a solution, as a greater number of layers than the original Atari paper[2], but is less aggressive with using stride to reduce the number of signals between layers.

- Input, 64x64x3 inputs
- CNN, 3x3, 16 filters, ReLU
- CNN, 3x3, 16 filters, ReLU
- CNN, 7x7, 16 filters, stride = 2, ReLU
- Dropout, probability 50%
- CNN, 3x3, 32 filters, ReLU
- Maxpool 2x2
- CNN, 3x3, 32 filters, ReLU
- Maxpool 2x2
- CNN, 5x5, 32 filters, ReLU
- Maxpool 2x2
- Dense, 128 nodes, ReLU
- Output, 4 nodes, Linear

The cost function for learning, is a linear combination of the squared error and error, with L2 regularization to limit the parameter sizes. Implemented such that it is only calculated based on a single action, namely the action that was taken for which information is available.

3.3. State Pre-Processing

In order to generalize the code to application on different sized problems, a state observation pre-processing step is taken. First the state observation is cropped to remove parts of the screen that does not contain information. For breakout, and for fast convergence, it was chosen that this area is only the area from below the wall and down. This choice was taken as a relatively good strategy can also be devised without knowledge of the brick location. Step two is conversion to black and white, and normalizing to between 0, black, and 1, white. Finally scaling the image to the standard form 64x64 and stacking 3 consecutive frames. In figure 2 three frames are displayed as the channels in a RGB image.

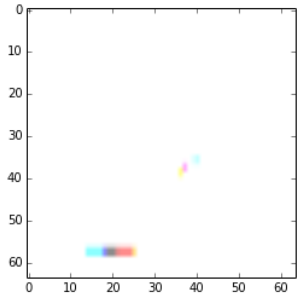


Fig. 2. Preprocessed frame: 3 consecutive frames displayed in one RGB image

3.4. Reward Augmentation

The original reward function only gives a positive reward when a block is hit. To get faster convergence, the reward function is augmented to include a negative, -1, reward when loosing the ball. This reward is given much closer to the eligible actions, and therefore much easier for the network to learn an action for.

Further as the network did not seem to be able to find a strategy getting more than three points, the reward was propagated backwards with discount (n-step Q learning) closing the gap between action and reward.

3.5. Replay Memory

The replay memory is important to the solution because sampling randomly provides a set of frames to train on that is not highly correlated due to the fact that they are collected in a sequence, by sampling actions. The larger a replay memory the better in order to reduce overfitting. We used a replay memory of approximately 250000 frames. For Flappy bird the closer relation between action and seemed to require fewer samples.

3.6. Flappy Bird

Flappy Bird was used as testing the generality of the implementation. For this game the background was removed, in order to simplify observations. Otherwise the algorithm is the same.

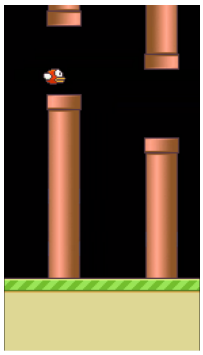
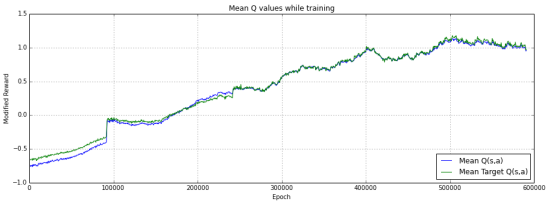


Fig. 3. Example frame from the flappy bird game.

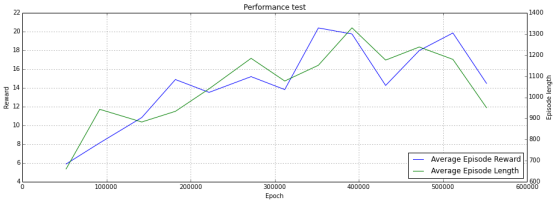
4. RESULTS

In figure 4 the results collected during training are displayed. In plot (a) the average Q value, the actual and the predicted, of the content int the replay memory during the training is shown. At a certain point our Amazon Instance got out bid, so we had to restart the training and thus reinitialise the replay memory. This moment can be observed by a big step in the means.

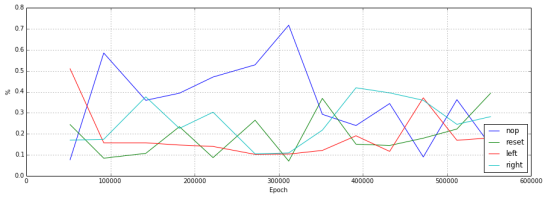
In plot (b) the average reward and the average length of 10 games played by our agent is displayed. The agent was set to play 10% random actions at every time. This was done every 20000 training steps. The agent had the highest average score after 400000 epochs, this was a score of $20,4$. The distribution of the actions during this performance test is shown in plot (c).



(a) Average Q value during training



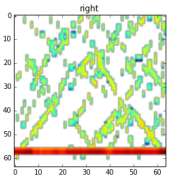
(b) Performance evaluation during training



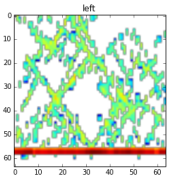
(c) Action distribution in the performance test

Fig. 4. Plots

Figure 5 a heat map of the states during a run of the agent with 50% random steps is shown. The displayed figures were recorded by the agent at initialised with the network at the 'best' point, when it had the highest average score.



(c) Heat map right



(d) Heat map left

Fig. 5. Heat maps when each action is performed

4.1. Flappy Bird

In figure 6 the action policy of Flappy Bird is visualized, after training 300000 steps. This resulted in a score average of 2

pipes.

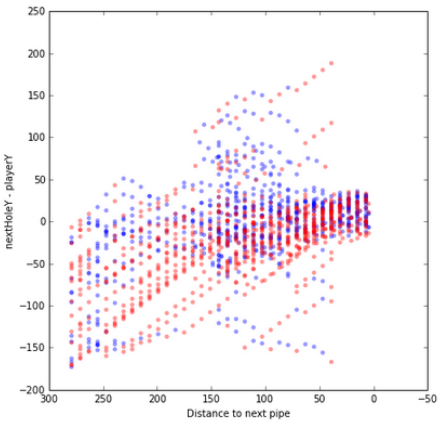


Fig. 6. Flappy Bird action strategy, showing the actions up (red), down(blue) given the distance to the next pipe and the error in height between the bird and the center of the hole. It is seen that in the lower half the strategy is mainly to go up while above the hole the strategy is to go down. Further it is seen how the samples are concentrated near the middle of the hole, as the good strategies become dominant in the replay memory.

5. DISCUSSION

The project has been an attempt to implement RL as a way of creating a program that can be used to play video-games without requiring large amounts modelling time. This turned out to produce satisfying results and without requiring the use of extensive modelling of the game that would otherwise be required in a planner.

As known from the initial paper describing the deepQ algorithm, it takes very long for the solution to converge. The original paper trains the network 5 million times, with a replay memory size of 4 million frames. Compared to that, we only train about an eight of the time, with an sixteenth of the replay memory.

We showed this as well, by having the algorithm work on Flappy Bird with few modifications to the hyper-parameters, and while this can be said to be similar to modifying the specifications in a generic STRIPS planner, modifying hyper-parameters guarantees some degree of solvability after a certain amount of time.

One of the problems with reinforcement learning however, is that we are not working in a closed mathematical model, and instead have to use human reasoning to derive why things work or why they do not. This includes not only modifying hyper-parameters, but also attempting to understand the input and output variables semantically, meaning that progress is determined also by our abilities to understand the system.

6. FUTURE WORK

Running the training for a longer time could result in a better score. Giving the complete image, including the blocks, as input could result in learning special tricks but would require much longer training. Future steps in the development of an better agent could be the comparison of different neural networks for the Q function and comparing hyper parameters. This is hard to do because the required time for training is relatively long.

7. CONCLUSION

In this paper an implementation of deep n-step Q learning using replay memory.

We used a series of neural networks to estimate the Q-function, applying such ideas on the network as allowing it to learn the element-to-element relationships in pictures by using convolutional layers, which the network then used as state input for the Q-Function.

In order to separate state/action pairs, so as to make them not immediately dependent on the following or previous state/action pair, we used replay memory, which stores the results of performing actions and allows us to use them later, making disjunct the current events from past events.

We have shown that it is possible to learn a strategy for playing breakout attaining 20.4 points on average per game, after 600 000 training samples. Applying the code to Flappy Bird resulted in an average score of 2 pipes after 300 000 samples.

8. REFERENCES

[1] Thore Graepel, "Alphago - mastering the game of go with deep neural networks and tree search," *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9852, pp. XXI, 2016.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller, "Playing atari with deep reinforcement learning," 2014.

[3] Warren S. McCulloch and Walter Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[4] Volodymyr Mnih, Adri Puigdomnech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016.