Space Cadet Pinball using Deep Q-Learning

Valter Schütz, Elias Stenhede, *Chalmers University of Technology*

***Abstract*—Using ball position, velocity and flipper/plunger status as states we were able to train a Deep Q-Network to play Space Cadet Pinball, using methods from [3] such as prioritized experience replay and a target model. The performance achieved was reasonably close to but not better than a skilled human player.**

Results showed that the agent learned a distinctive strategy of rapidly flickering the flippers in order to hit the ball, and shows some understanding of how actions impact the game.

I. Introduction

This report examines if Deep Q-Networks (DQN) can be used to achieve reasonable performance on a realistic pinball game. Pinball was chosen due to the authors' fondness for it, and Q-learning was selected because of its widespread use. It has also been shown before that DQN can achieve superhuman performance on many video games [2]. The specific game in question was a decompiled version of Space Cadet Pinball, which is significantly more complex than its Atari counterpart [2]. A screenshot of the playing field is seen in fig. 1. Due to the increase in complexity, it was not obvious that the methods and/or architecture described in previous research would work well.

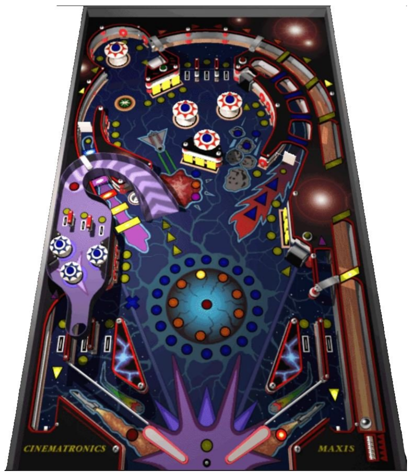

Fig. 1. The playing field of Space Cadet Pinball. The score is increased a bit whenever the ball hits the bumpers, spinners, drop targets, etc. The score is increased by a lot when missions are completed, missions are completed after hitting multiple targets in correct order.

Note: For readers not familiar pinball terminology, see section VII.

II. Background theory

The general problem of reinforcement learning is for an *agent* to act optimally in a Markov decision process (MDP), defined by a set of states $S_t \in \mathcal{S}$, available actions in each state $A_t \in \mathcal{A}(S_t)$ and scalar rewards R_t. Given a state S_t and action A_t, the next state S_{t+1} and reward R_{t+1} is stochastic, modelled by the transition probabilities

$$p(s', r|s, a) \doteq P(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a). \quad (1)$$

When interacting with an MDP we generate a trajectory $(S_0, A_0, R_1, S_1, ...)$ and by acting optimally we mean that we maximize the expected return,

$$\mathbb{E}[G_t] \doteq \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \dots] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+1+k}\right] \quad (2)$$

where $0 \leq \gamma \leq 1$ is a discount factor that decides how much future rewards are valued.

The function that decides which actions to take contingent on the current state is called the policy

$$\pi(a|s) \doteq P(A_t = a|S_t = s) \quad (3)$$

and the goal of reinforcement learning is to find the optimal policy π_* that maximizes the expected return.

Methods for achieving this fall into one of two categories: model-based or model-free. As can be deduced from their names, the model-based methods require knowledge about the system dynamics, i.e of $p(s', r|s, a)$, while model-free methods are able to learn the dynamics indirectly from experience. While incorporating prior knowledge might be useful, in most cases the system is not well understood enough, promoting the use of model-free methods over model-based ones. Another advantage with model-free methods is that they often perform surprisingly well even on systems that do not have the Markov property.

Model-free methods can be further classified into subgroups. Policy gradient methods aim to directly learn the optimal policy π_* while value iteration methods learn π_* indirectly by alternating between learning the state-value function v_π or action-value function q_π for some initial policy π,

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] \quad (4)$$

$$q_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \quad (5)$$

and then improving it.

Methods can further be classified depending on whether they can learn from state/action pairs collected from a different policy (off-policy) or the same policy (on-policy). In this context, Q-learning is a very popular off-policy action value iteration method.

Q-learning utilizes the fact that the optimal action-value function satisfies the Bellman optimality equation

$$q_*(S_t, A_t) = \mathbb{E}\left[R_{t+1} + \max_{a'} q_*(S_{t+1}, a)|S_t = s, A_t = a\right] \quad (6)$$

Thanks to github user `k4zmu2a` for providing us with a decompiled version of the game.

and uses this as the loss when updating the Q-estimate (sampling from RHS)

$$L = \Big(Q(S_t, A_t) - R_{t+1} - \max_{a'} q_*(S_{t+1}, A_{t+1})\Big)^2. \quad (7)$$

Since Q-learning is an off-policy method, $S_t, A_t, R_{t+1}, S_{t+1}$ can be stored during interaction with the environment and potentially be used in a much later update when the policy has altered significantly.

The most natural way to choose actions during learning is to choose what the agent thinks is currently optimal, i.e

$$\text{action} = \arg\max_{a'}(q_\pi(s, a')). \quad (8)$$

However, this often leads to convergence to sub-optimal policies since the agent sticks with what is "good enough" and does not bother with exploring other strategies. A popular remedy to this exploration-exploitation problem is to instead use an ϵ-greedy learning policy during training

$$\text{action} = \begin{cases} \arg\max_{a'}(q_\pi(s,a')) & \text{with prob.} \quad (1-\epsilon) \\ \text{random action} & \text{with prob.} \quad \epsilon \end{cases} \quad (9)$$

Since DQN is an off-policy method, it is still possible to converge towards an optimal policy, using a "suboptimal" (with $\epsilon > 0$) policy.

III. Related work

Q-learning for discrete state spaces was first introduced by Watkins [1] in 1989. Watkins introduced and proved convergence of the Q-learning algorithm. This method uses a Q-table, keeping track of the expected value of each state/action pair. For an agent to act in an informed way, all the relevant state/action pairs have to be encountered during training, so that the Q-table can be filled. Due to the curse of dimensionality, Watkins original Q-learning algorithm becomes unfeasible for continuous or high-dimensional state spaces.

Using deep neural networks to approximate the action-value function, Google DeepMind [2] managed to modify the Q-learning algorithm to make it suitable for high-dimensional and continuous state spaces, the first use of a DQN. Instead of having both the state s and action a as input to the network they only used the state as input and outputted $Q(s, a)$ for all actions a in the action space for computational efficiency. With these improvements they were able to outperform humans on a wide variety of Atari games [2].

Due to the convoluted structure of a DQN, several new problems arose. For example, states that are visited are usually highly correlated (if the state space changes continuously, successive snapshots are very similar) which might cause the learning to become unstable and inhibit optimal learning.

To remedy the problem of correlated states, Google Deep Mind separated sampling and learning using a method called *experience replay*. Firstly, the agent explores the state space and samples containing `(state, action, next state, reward)` are saved in the *experience replay buffer*. When the buffer is filled, minibatches are uniformly sampled. With a large buffer, the samples in each minibatch will be uncorrelated with high probability.

In the following years, numerous improvements to DQN were proposed [3]. When combined, much faster convergence and much better performance was accomplished. When using *prioritized* experience replay, sampling is not done uniformly from the experience replay buffer, since some samples `(state, action, next state, reward)` might contain more useful information than others.

IV. Method

The state was comprised of the ball's position and velocity, as well as the statuses of the flippers/plunger (pressed/released). The available actions consisted of left flipper up, left flipper down, right flipper up, right flipper down, plunger up, plunger down, and pause (no-op). Since we have a 7-dimensional state space and 7 possible actions, a Q-network with 7 inputs and 7 outputs was used. To enable complex and nonlinear mappings from state to Q-values, a deep architecture consisting of 4 hidden layers with 256 elements in each layer was implemented, see fig. 2. Between each layer, ReLU was used as an activation function.

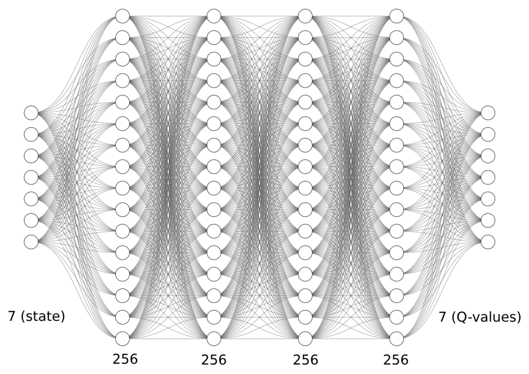

Fig. 2. The Q-network transforming states to Q-values, which are used to choose the optimal action.

To train the Q-network, a virtual Google cloud machine equipped with a Nvidia L4 GPU was used.

The unmodified game runs at around 120 frames per second. To speed up the simulation the C++ source code was modified. The connection between "real world time" and "game time" was removed, enabling the game speed to be limited only by CPU capacity. The actual video display was piped to a dummy screen using X Virtual Framebuffer (Xvfb), and the same was done for the audio, since the virtual machines used did not have audio or video enabled.

Furthermore, the code had to be modified so that the agent implemented in PyTorch could receive the game state and rewards, as well as input actions into the game. A synchronization mechanism using shared memory between the Python and C++ code was implemented, letting the agent act on the state before proceeding.

To help the agent learn what states are good (making them correspond to high Q-values), the game was somewhat modified

- In the original game, a fixed end of game bonus was awarded upon the end of each round. This end of game bonus was removed, since giving positive reward for ”bad” actions might hinder efficient learning.

- In the original game, each round consists of three consecutive balls being played. It takes some time for a new ball to be deployed, there is nothing useful to learn from these game states without a ball in frame, so the number of balls per game was reduced to one.
- When the game starts, it takes about 500 frames before the ball is deployed. There is nothing useful to learn from these frames, so the first 500 frames were skipped in each game.
- In the original game, the player gets a *ball save* when *draining* too fast, this feature was removed to simplify training.

The main objective was to make the agent score as much as possible, but to train the agent, some reward has to be defined. The reward shaping is extremely important as it is what the agent is trying to maximize. It makes intuitive sense to let the reward be related to the difference in score between each frame, however there are some details that have to be accounted for.

Score is earned in chunks of between 500 and 2,000,000 (very rare, only when completing special mission, i.e multiple successful shots). Large reward variance can pose a problem for efficient learning; since MSE loss was used, the largest rewards would totally dominate the gradients.

The reward was therefore initially chosen to be

$$R_i = \min(\frac{\Delta\texttt{score}_i}{20000}, 1). \tag{10}$$

where $\Delta\texttt{score}_i$ is defined as score in frame i minus score in frame $i-1$.

The result of this was to make the agent value high scoring actions more than low scoring actions, while still not making it overfit to extremely rare ”jackpots”. However, when training with this reward the agent learned to *cradle* (see section VII, terminology) the ball for excessively long amounts of time.

To remedy this, some punishment for keeping the flippers up was added, namely the reward was redefined as

$$R_i = \begin{cases} \min(\frac{\Delta\texttt{score}_i}{20000}, 1) - \delta & \text{if any of the flippers are up} \\ \min(\frac{\Delta\texttt{score}_i}{20000}, 1) & \text{otherwise.} \end{cases} \tag{11}$$

The punishment $\delta = 0.01$ was chosen.

During training, the actual learning was made through sampling from the experience replay buffer. The maximum size of the replay buffer was chosen to 4,000,000 elements, mini-batches of size 32 were sampled between each frame in the game and a learning rate of 5×10^{-7} was chosen. To begin with, ϵ was 1 but decreased to 0.1 over the first 500,000 frames. Starting with a high ϵ value is standard procedure, since focus in the beginning is on exploration of the state space.

During training, the game ran continuously on a single CPU core, while the decision making (forward propagating current state through Q-network) and learning (backward prop on mini-batch from experience replay buffer) was done on the GPU.

An alternative would be to instead first play all episodes with a $\epsilon = 1$ policy and only afterwards train the agent, but it appears more effective to interlace the simulation and learning, making it possible to utilize both the GPU and the CPU at the same time. This approach also allows the agent to start learning from data before the buffer reaches its full capacity of 4,000,000 items. Additionally, it enables the continuous replacement of elements in the buffer, reducing the risk of overfitting.

V. Results and discussion

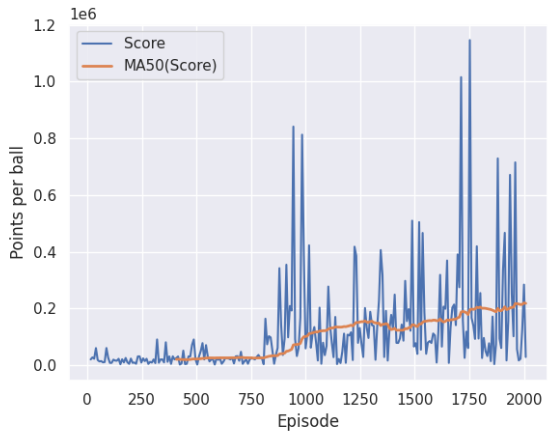

Fig. 3. Evaluation score obtained during training, here, only one ball was played, since that’s how the network was trained.

The agent learned to play the game using an unusual strategy, namely flickering the flippers rapidly when the ball is approaching. When evaluated on the original game (that is, with the training modifications removed), the results are as shown in fig. 4, where a comparison has been made with a skilled human player (former head of a pinball association, who had played the game before). The human had the advantage of seeing which missions were available and take appropriate actions in turn. The fact that the agent performs worse than the human is not surprising since large scores are only given when completing missions and the agent did not have access to mission states. If the pixels had been used instead as the input and if the network had been more complex, the agent could in theory have learned to decode the text on the right side of the screen which gives information about mission states, and then aimed at the appropriate targets.

The training loss is shown in fig. 5. Initially, the loss is low since rewards are very rare. As the agent learns the game dynamics and starts to get larger rewards, the loss increases but eventually decreases as the policy improves. In fully deterministic games, loss can reach zero with a globally optimal policy, but in the presence of randomness, an optimal policy will yield a nonzero expected loss.

In fig. 6 we have plotted the optimal action across the board gathered during a few episodes. Note that each position on the board can have states with different ball velocities and flipper states, resulting in the mixing of colors for a given area on the board. It is clear that the agent has learned that it is pointless

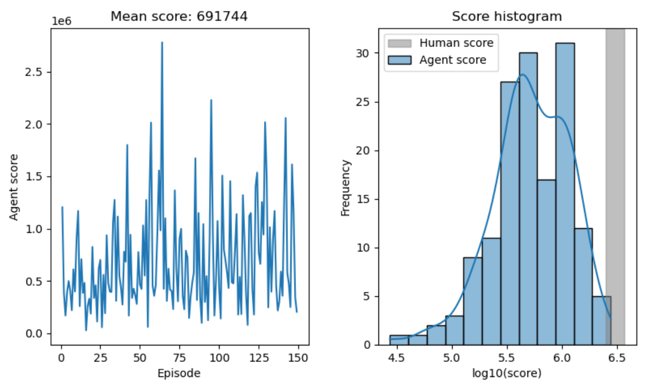

Fig. 4. Evaluation of the agent after finishing training. The human score was obtained by letting an experienced player play the game for 20 minutes. The grey area is covering both the highest and the lowest score obtained.

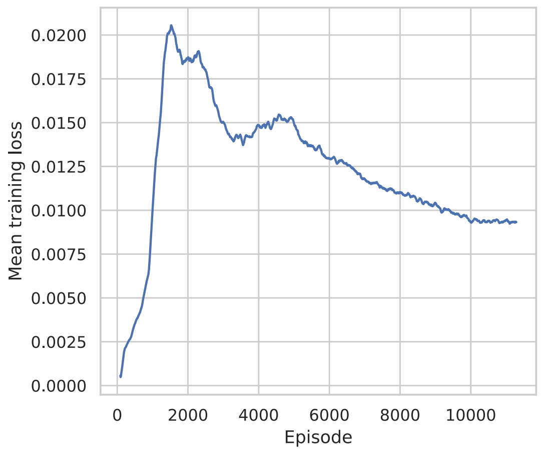

Fig. 5. Mean loss obtained during training. The loss increases as the agent starts to score higher without the Q-values reflecting it. When the Q-network starts to converge towards the optimal one, the loss decreases.

to use the flippers when the ball is high on the board but that it should activate them when the ball rolls down and probably deactivate them after hitting the ball.

Most interesting is perhaps the fact that the agent thinks it is useful to activate the flippers after roll-overs, seen most clearly in the lower left corner. Flipper actions have the secondary property that they interact with the roll overs (moving which ones are active to the left or right), motivating the opportunity of higher score in precisely that area.

VI. Conclusion

Using a relatively simple Q-network we were able to train the agent to achieve reasonable performance in the game of pinball. However, possibly due to the agents lack of access to important information such as mission status it was not able to beat human performance. One possible improvement would be to incorporate more methods from [3], such as distributional DQN which was shown to have a big impact on specifically the game of pinball, Another option is add information to the state by including more game information (roll over status,

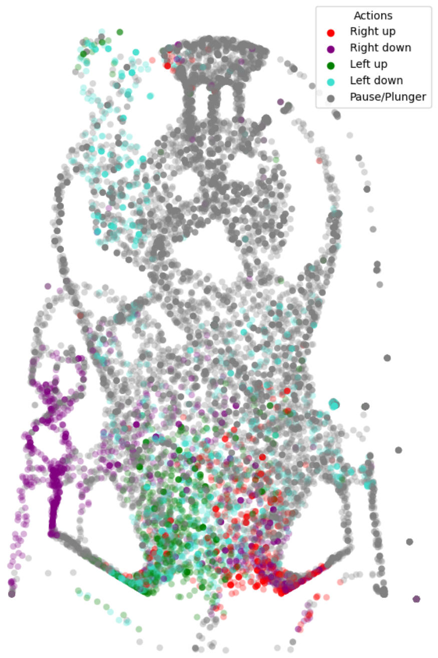

Fig. 6. Preferred action as a function of ball position over the course of 100 games. Note that the Q-values also depends on ball speed as well as flipper- and plunger states, which is not shown in the picture. This explains why different actions might be optimal at the same spatial coordinate.

mission status, etc.) or to use raw pixel information as the state.

VII. Pinball terminology

- **Roll overs** - are spots on the playing field lit when the ball passes, in Space Cadet Pinball, they occur in triplets at the top of the playing field, as well as the left part of the playing field. When all three rollovers are lit, a bonus multiplier (meaning increased score) is upgraded.
- **Cradling** - is a technique where a player traps and holds the pinball using the flippers on the pinball machine.
- **Draining** - is when the ball is lost through the side lanes or between the flippers. Usually means the ball is over.
- **Ball save** - if the player drains within a short amount of time after the game started, a new ball is immediately given by the machine.
- **Plunger** - is the spring loaded rod at the bottom right side of the pinball table. This is pulled and then released to launch the ball and start the game.

References

[1] Christopher John Cornish Hellaby Watkins. “Learning from delayed rewards”. In: (1989).
[2] Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: (2013). ISSN: 0166-4328. URL: http://arxiv.org/abs/1312.5602.
[3] Matteo Hessel et al. “Rainbow: Combining Improvements in Deep Reinforcement Learning”. In: *CoRR* abs/1710.02298 (2017). arXiv: 1710.02298. URL: http://arxiv.org/abs/1710.02298.