Autonomous and Adaptive Systems
Final Project

August Johansen Fors

June 2024

Abstract

This paper is based on DeepMind's paper named *Playing Atari with Deep Reinforcement Learning*[2]. In my implementation I use the *ProcGen*[4] environments to represent Atari-style games. The game is to be played by an reinforcement learning agent where the input is raw RGB-frames of the game. This is to be achieved by using a convolutional neural network in combination with a reinforcement learning algorithm known as Q-learning.

1 Background

In this specific project I have considered environments given by ProcGen. These are environments where the reinforcement learning agent is to play an atari-style game. The environments, $\mathcal{E}$, are set up in a way where the observation at time t of the state, x_t, is a frame of the game as a player would see it on a screen. The agent chooses and action following the policy, π, given the state, s_t. The environment then returns a reward, r_t, for the action taken. The objective is to let the agent interact with the environment and select actions to maximize the expected reward. By always ensuring a finite number of time steps, we end up with a finite Markov decision process.[5] This makes it possible for us to use standard reinforcement learning methods.

One of the biggest challenges in this setup is making sense of the observation. As a human it's easy to interpret a screen in order to play the game, but it's quite a difficult task for a computer. In addition, reinforcement learning algorithms require a way to evaluate the expected reward given a state and an action. If the state space is finite and representable, this can be done by the *Bellman Equation*(1). The Bellman Equation lets us calculate the expected future reward of a given state, given the policy. The function that calculates the expected return is known as the action-value function, $Q^*(s,a)$. The policy for the agent will have the optimal state-action function given Bellman equation. The agent would then greedily select the action with the highest expected return.

$$
\begin{aligned}
Q^*(s,a) &= \max_{\pi} \mathbb{E}[R_t|S_t = s, A_t = a] \\
\Rightarrow Q^*(s,a) &= \mathbb{E}[r + \gamma \max_{a'} Q^*(s',a')|s,a]
\end{aligned} \tag{1}
$$

Where the action $a \in \mathcal{A}(s)$, the next state $s' \in \mathcal{S}$ and $r \in \mathcal{R}$. And $\mathcal{A}(s)$, $\mathcal{S}$ and $\mathcal{R}$ are the action, state and reward spaces respectively. [3]

In most cases however, it is not feasible to calculate the expected returns in this manner, because there are either too many possible states, or the state space is continuous. In our case the state space is discrete, but there are simply too many different states, that is different images the screen can display, to ever be represented directly. What we need is a way to approximate the expected return given a state. There are, of course, several ways to tackle this challenge. For games like the maze game, where the objective is to control a mouse through a maze in order to find a piece of cheese, one could preprocess the observation into a grid telling where the mouse is, where the cheese is, and where the walls are. This would make the observation space a lot smaller. This smaller observation space would make it feasible to use the Bellman equations to calculate the expected future reward from a given state.

This method is effective but is not very general. With different games of different complexity this many not be a feasible solution, and the preprocessing of the observation space would have to be customized for each game. A more general approach is to use a convolutional neural network(CNN) in order to approximate the expected return from a given state. This neural network can be split into two main parts - the convolutional part and the fully connected part. The convolutional part is used to compress the data from the image into features that can define the state. This is then passed into the fully connected layers in order to decide what action to take based on the given features. This gives a method that is very general when solving these types of problems. If a program performs well on one game, it can often just be retrained and perform well on another game.

For this project I will be using a reinforcement learning algorithm called DQN, which stands for Deep Q-Network. This algorithm uses the standard Q-learning algorithm together with a CNN as a value function approximator. We will approximate the state-value function with a neural network this is done by minimizing the loss function $L(\theta)$ given by equation 2.

$$
L_i(\theta_i) = \mathbb{E}[(y_i - Q(s,a;\theta_i))^2] \tag{2}
$$

where $Q(s, a; \theta_i)$ is the predicted Q-value and θ are the parameters of the network. $y_i = \mathbb{E}[r + \gamma \max_{a'}]Q(s', a'; \theta_{i-1}|s, a)$ is the target and is given by the Bellman equation and the value function approximator. Both s and a are chosen form a probability distribution $\rho(s, a)$.

2 Implementation

There are quite a few things to think about when implementing an algorithm like this in code. One thing used to increase stability during training is replay memory. For every step taken the algorithm saves state, action, reward and the next step, (s_t, a_t, r_t, s_{t+1}), into a replay memory. When the algorithm then goes onto optimizing the network, it randomly samples a batch of the replay memory. This ensures independence in the samples, and is proven to increase stability in training by reducing the variances in the updates of the weights. The replay memory is kept at a max size but needs to be large enough to ensure adequate independence.

The DQN algorithm chooses the next action epsilon-greedily. This means that there is a chance of ϵ that the agent will choose a random action from the action space. The way it is implemented here epsilon starts with a value of one, giving a 100% probability of choosing a random action. It stays at one for a given number of steps before it linearly decreases towards a minimum value, shown in figure 1. This is to provide adequate exploration before honing in on an efficient policy. As with all reinforcement learning algorithms there will always be a trade-off of exploration vs. exploitation, that is, a trade-off between exploring new strategies or exploiting the best strategies found so far.

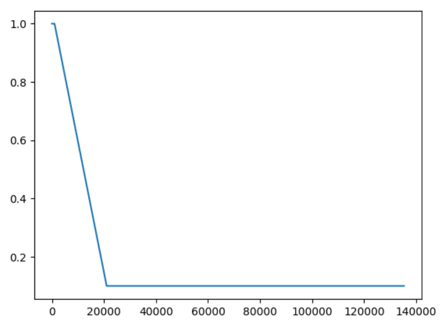


Figure 1: Value of epsilon over episodes

Another technique used to improve the stability of the training is utilizing two separate neural networks. One network, in this project referred to as the training network, is used to predict the Q-values for a given state. During training this network is updated every few steps to minimize the difference between predicted Q-values and target Q-values. The other network, which we'll call the target network, is updated by copying the training network, and is updated far less frequently. This increases stability by decoupling action selection and target calculation. Without this the program could end in a feedback loop where the Q-values choose the action, and the action affects the Q-values and so on. A network updating too frequently can lead to oscillations or divergence in training. Q-learning is an off-policy method, meaning that the actions taken during training doesn't follow the policy. Due to this there is no problem in using two separate networks.

A choice that was made to reduce the scope of the project was not implementing stacking of frames. With this the observation is increased to include several consecutive frames in order to include a time aspect into the observation. This can be crucial for certain games where, for example, the trajectory of an object can be of importance. In this project, however, I've decided to focus on games that don't have this need.

3 Testing

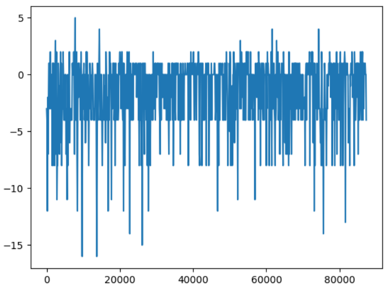
To test the implemented algorithm I mainly focused on two different games. These games were *fruitbot*, where the agent is to collect fruit and avoid crashing in a scrolling world, and *maze*, where the agent is to lead a mouse to the reward through a maze. These were chosen so that frame-stacking wouldn't be need as discussed previously.

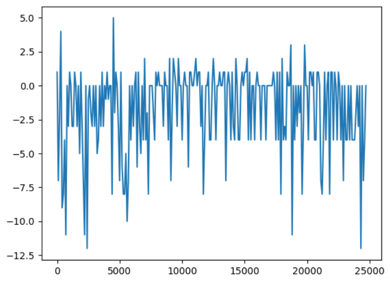
The training was very limited by computing power. In the original Deepmind paper the agent was traning for 50 million steps. During my research I also found others who claimed decent results training for 10 million steps[1]. In my training, while training for around 4 days, I was able to train for 6 million steps at the most. For testing I decided to turn off the background and set distrobution mode to easy, as this should make the problem easier. I also used the option of training only on one level that stays the same throughout the training process. The results will be discussed in the next section.

4 Results

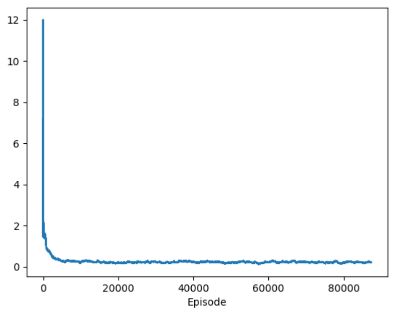
The results attained from this project and the training resources available were quite underwhelming. I was not able to produce any kind of convincing result. It is hard to say why this is the case, but I have some thoughts on what could be the cause, as well as some curiosities I have noticed. I tried training on the *maze* game in addition to *fruitbot*, but it didn't show any change in results, so I left it out of the report.

The main reason I think my project is under preforming is simply not enough training. Or a mix between to little training and not enough exploration. As previously discussed my longest training session was far short of the number of steps used by DeepMind in their paper. This also contributes to difficulties of tuning the system. There are a lot of parameters to tune, like the ones concerning exploration vs. exploitation, batching, learning rate, etc. To tune these becomes difficult when I really have no way to see the results other than training for a very long time. Tuning some of these parameters could probably increase performance and convergence, but as discussed, was not feasible for the scope of my task. In addition, there is really no way of knowing if the convolutional part of the network is working properly. This is the part of the network responsible for taking the raw RGB-frame and turning it into useful information to feed into the rest of the network. If this part isn't performing properly, the decision-making part of the network has little to work with.

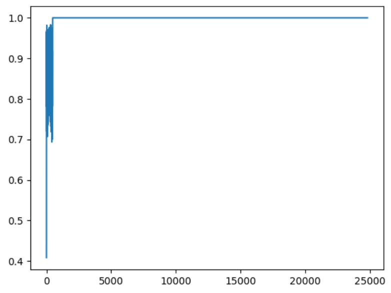

(a) Rewards over episodes


(b) Rewards over episodes for single level

A good way to see the results is to look at the reward throughout the training process. As the network trains and (hopefully) learns a better and better policy, the reward should increase over episodes. Especially as the value of epsilon decreases. As we can see in figures 2a and 2b, there is no apparent increase in rewards, as it seems to randomly jump around throughout the training.



(a) Average Q-value over episodes


(b) Average Q-value over episodes for single level

To discuss some curiosities I noticed, that could give some hints towards the poor performance, I will first discuss the Q-values provided by the network. This is the Q-value for the best action, regardless is the agent took that action or a random one. For each episode I stored the mean of the Q-values of the action chosen for each step. As we can see in figure 3a, the Q-values quickly approach zero, and stay there. The action chosen is still the one with the highest Q-values, but all actions have very low Q-values. As the Q-values are supposed to be an estimate of the expected reward given state and action, I suppose the low Q-values make sense. As the agent is not able to find a good strategy resulting in a high reward, then non of the actions are expected to give a large reward.

Another curiosity is that my Q-values stayed unchanged after a while when training only on one level. The Q-value quickly settled on 1, and stayed unchanged for the rest of the training. In fairness, this wasn't a very long training session and the value might have changed eventually. This constant Q-value point to the network getting stuck on a policy, for example, going right every time.

The last result I want to discuss is how the loss changed throughout the training. The loss is the function defined in equation 2, and is what the network is trying to minimize. As we can see in figure 4 the loss function steadily decreases during the training. This points to the network working properly, and the weights being updated in a correct manner.

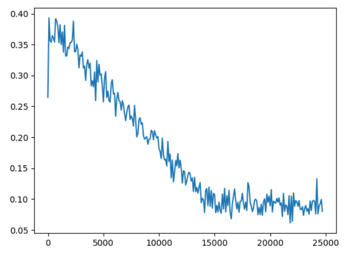


Figure 4: Loss over episodes

References

[1] J. Chapman and M. Lechner. *Deep Q-Learning for Atari Breakout.* 2022. URL: `https://keras.io/examples/rl/deep_q_network_breakout/`.

[2] V. Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: (2013).

[3] M. Musolesi. *Value Function Approximation in Reinforcement Learning.* URL: `https://www.mircomusolesi.org/courses/AAS23-24/AAS23-2main/`.

[4] OpenAI. *ProcGen.* URL: `https://github.com/openai/procgen/tree/master`.

[5] A. Paszke and M. Towers. *Reinforcement Learning (DQN) Tutorial.* URL: `https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html`.