Atari, Deep Reinforcement Learning

Matjaz Zupancic Muc
Faculty of Computer and Information Science
Ljubljana, Vecna pot 113
Email: mm1706@student.uni-lj.si

***Abstract*—In this seminar we attempt to train an agent to play two Atari games of varying complexity (Pong & Freeway) using an off policy, model free method, known as Deep Q Learning. We are able to achieve great performance on the simpler game and decent performance on the more complex game. We expose the main advantage & weakness of this approach.**

I. Introduction

The core of this project is based on [1], where Deep Q Learning Algorithm was presented. The novelty of this approach was developing a model free agent, which learns using high dimension observations of the environment, such as pixels and scores to train a reinforcement agent. Method outperformed the best existing reinforcement learning methods on 43 out of 49 Atari games without incorporating any additional prior knowledge about Atari games. The second paper [2] introduced a modification which prevents over-estimations of action values. In this seminar we evaluated both algorithms on the Pong & Freeway Atari game and compared the results.

March 3, 2022

II. MATERIALS AND METHODS

A. Background

Before we proceed to methods and results, we provide an explanation of main concepts. Deep Q Learning Algorithm builds on top of classic Q-learning which is impractical for environments with a large state space. Algorithms uses a deep neural network to learn from the large state space. Neural network is used to approximate the optimal Q-function: $Q(s,a,\theta) \approx Q^{*}(s,a)$. Where θ are the parameters of the neural network. Standard Q-learning is known to diverge when the state space becomes large. To solve this issue the following two concepts were introduced.

First concept known as experience replay is introduced to remove correlations in the sequence of observations, it also prevents the agent to get stuck in a poor local minimum. The concept is based on storing experiences at every time step. Experience is defined as $e_t = \{s_t, a_t, r_t, s_{t+1}\}$, where s_t is the state of the environment at time t, a_t is the action performed by the agent at time t, r_t is the reward obtained by performing a_t in s_t, s_{t+1} is the resulting state in which the agent finds its self after performing a_t at s_t. The agent stores experiences in a data set $D_t = \{e_1, e_2, e_3, ..., e_t\}$. Data-set D is formally called replay memory buffer and is of size N, when its size exceeds N, old experiences are overwritten with new ones. A sub set of experiences is sampled at random from D at every time step t and used to update the online network Q.

Second modification is the use of a separate network (target neural network) for generating the targets, which are defined as: $y_j = r_j + \gamma max_a Q^{'}(\phi_{j+1}, a^{'}, \theta^{-})$. Target network gets updated using online network every C steps. Using two networks makes the algorithm more stable. Generating the target values using an older set of parameters θ^{-} adds a delay between the time an update to Q is made and the time the update affects the targets y_j, making divergence or oscillations less likely. The following is the algorithm introduced in [1].

```
**Algorithm 1: deep Q-learning with experience replay.**
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
**For** episode = 1, $M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  **For** $t$ = 1,T **do**
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
    network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  **End For**
**End For**
```

In [2] they showed that Deep Q Learning tends to suffer from overestimation of action values. They pointed out that since Q-learning selects optimal action by picking the maximal Q value for a given state, it tends to prefer overestimated Q values. In case over-estimation effects all action values equally such problem will not occur. If, however the over estimations are not uniform across states, the policy gets skewed. The max operator in standard Q-learning and Deep Q learning, uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates. To prevent this they they suggest we use one network Q to select the max action and the second network Q^{-} to evaluate the value of that action. We change the definition of target as follows: We no longer use $y_i = r_i + \gamma max_q Q(s_{i+1}, a_{i+1})$, instead we define target as $y_i = r_i + \gamma Q^{-}(s_{i+1}, max_q Q(s_{i+1}, a_{i+1}))$.

B. Methods

1) General: Actions space of both games is composed of three actions, the agent can either move u, down or stay still. The size of the state space is $256^{84\times84\times4}$. The state space is the set of screen pixels (Image size: 84×84, Grayscale: 256 gray levels, 4 consecutive images).

Pixel images are pre-processed before we learn from them (code credit: [5], [6]). Images are downscaled and converted to grayscale, flickering of pixels is removed, the last 4 observations of environment are stacked.

Structure of the neural network is similar to the one presented in [1]. The network has 3 convolutional layers and 2 fully connected layers. Network uses RMSProb optimizer and a MSE Loss function. The agent is trained using epsilon-greedy approach, the value of ε decreased from $\varepsilon_{max}=1$ to $\varepsilon_{min}=0.1$, using steps size of $\varepsilon_d=0.00001$.

2) Pong: We'll start by training a DQN agent on a simpler game. We let the agent train until its score stabilizes (this took around 1 million steps). No additional tuning of network parameters was performed. Average score as a function of number of steps is displayed on Fig.2. The figure clearly shows that agent learns over time and that learning is stable, unlike the regular Q-learning. We can see that most of the learning occurs as soon as the epsilon drops down to its minimum of 0.1. The max score is reached in around 600000 steps. To see if action overestimation plays a role in pong, we will also train a DDQN agent, results are displayed on Fig.2. Results are similar, but the slope of performance seems slightly steeper at the last few episodes, which suggests that agent could pick up more skill, with more training time. DQN & DDQN agent gameplay (with 1% of random actions) [7]. Figure Fig.3 shows the distribution of actions that a trained agent performs. Note that action 0, represents no action, actions 1, represents move up, action 2 represents move down.

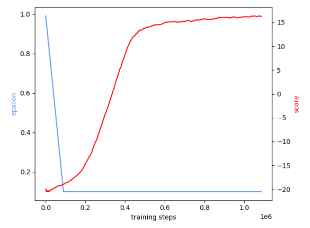
3) Freeway: We'll attempt to train a DQN & DDQN agent to play a game with a different setting. As mentioned the actions space of Freeway is identical to the one in Pong. The environment is more dynamic & complex. The goal of the game is to safely cross ten lanes of freeway traffic as many times as possible in a set amount of time ($1.09min$). When the player is hit by a vehicle the game does not restart, but rather the player is set back 2 lanes & and spends a fraction of time recovering before it can move again. Initially we trained both models for 1000 episodes, using a replay buffer of 70000.

Fig.2 shows DDQN agent performance. We can see that DDQN agent starts to learn when epsilon drops to $\varepsilon=0.1$. Learning appears to be relatively quick. We should also mention that a sub optimal agent which always picks forward action, achieves an average score of 21. This may explain why agent learns so fast, it quickly learns that choosing forward action in most states delivers lots of reward. For this reason we also plot the distribution of actions a trained agent selects. We can see that agent pick forward action most of the time, but also learns that sometimes performing action 0 (staying still instead of just running for the prize) or even performing action 2 (stepping back) makes sense. DQN DDQN Agent game play [7].

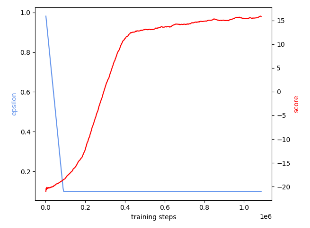
Finally we let the DDQN agent play another 3000 episodes, to see if it can pick up more skill. The learning curve is shown on Figure 4. It seems like the agents performance does not increase much from 2 to 5 million steps, but it seems like it does increase from 5 to 8 million steps. If we take a look at action distribution, agent seems to pick action 2 more frequently than action 0. DDQN Agent game play [7].

Table 1. shows average score over 10 episodes for all agents, (trained agents use $\varepsilon=0.01$). *DQN* denotes a DQN agent trained for 1 million steps in case of pong & 2 million steps in case of freeway. *DDQN* denotes a DDQN agent trained for 2million steps in case of pong & 2 million steps in case of freeway. $DDQN_8$ denotes a DDQN agent trained for 8 million steps. $NAIVE$ denotes agent which always selects forward action in freeway.

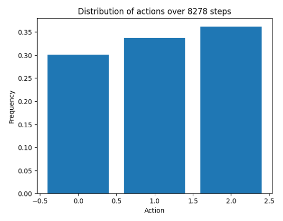
C. Results



(a) DQN

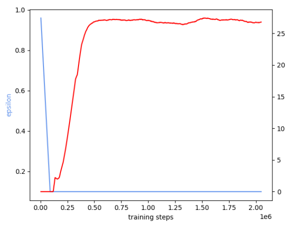


(b) DDQN

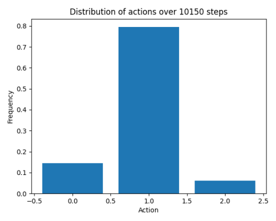


(c) DQN, Action distribution

Fig. 1: Pong, performance after 500 episodes

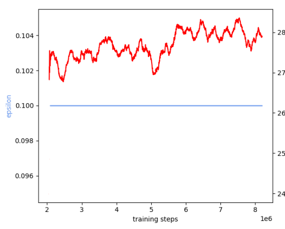


(a) DDQN

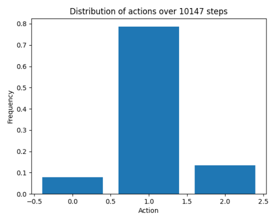


(b) DDQN, Action distribution

Fig. 2: Freeway, performance after 1000 episodes



(a) DDQN



(b) DDQN, Action distribution

Fig. 3: Freeway, performance after 4000 episodes

Game	*DQN*	*DDQN*	*DDQN$_8$*	*NAIVE*	*RANDOM*
Pong	20.8	20.8	/	/	-21.0
Freeway	todo	31.5	32.5	21.2	0

TABLE I: Average score over 10 episodes

III. Conclusion

We evaluated DQN and DDQN agent on two games of varying complexity. We showed that both methods are capable of delivering decent results. We displayed one of main benefits of these method, that being is its re-usability. We expose the most noticeable weakness which is sample inefficiency. As soon the complexity of environment increased the training time necessary to achieve decent results increased substantially. Further research could include kick staring a model with a model trained on a different game, to see if training time decreases.

References

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

[2] Van Hasselt, H., Guez, A. Silver, D. (2015). Deep Reinforcement Learning with Double Q-learning.

[3] http://karpathy.github.io/2016/05/31/rl/

[4] https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On/blob/master/Chapter06/lib/wrappers.py

[5] https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On

[6] https://github.com/lexfridman/mit-deep-learning

[7] https://drive.google.com/drive/u/0/folders/1DmdHEUcrO6_F2zk0ytMzL3K9vcyw9rPA