Learning Space Invaders with Deep Q-Learning Networks

Davide Lanzoni
Computer Engineering
University of Bologna
davide.lanzoni@studio.unibo.it

Abstract

In this report the implementation of a reinforcement learning algorithm capable of learning how to play the game Space Invaders emulated with The Arcade Learning Environment is covered. Here the last version of The Arcade Learning Environment is used, with some improvements that are discussed. An evaluation is done, and compared to some previous works. In the end a discussion about further techniques that could improve the process is considered.

1 Introduction

Space Invaders is an arcade video game by Atari in which the goal is to defeat wave after wave of descending aliens with a horizontally moving laser to earn as many points as possible. The invaders must be defeated before they reach the bottom of the screen or before they hit the cannon three times with their laser. Points are scored each time a Space Invader gets hit. They are worth different amounts of points, depending on their initial position on the screen (the Space Invaders are worth 5, 10, 15, 20, 25, 30 points in the first through sixth rows respectively). If all 36 Space Invaders are defeated before they reach the earth, a new set of invaders will appear on the screen. The list of the possible actions are in Table 1

Table 1: Space Invaders action space

Num	Action
0	NOOP
1	FIRE
2	RIGHT
3	LEFT
4	RIGHTFIRE
5	LEFTFIRE

2 Arcade Learning Environment

The Arcade Learning Environment (ALE) is a framework for developing AI agents for Atari 2600 games like Space Invaders. ALE provides a game-handling layer which transforms each game into a standard reinforcement learning problem by identifying the accumulated score and whether the game has ended. By default, each observation consists of a single frame: a 2D array of 7-bit pixels, 160 pixels wide by 210 pixels high (Bellemare et al. [1]). There are different version for each game, and the newest with the last version of the framework is the -v5, the one used in this implementation . The new environments follow the best practices outlined in Machado et al. [2]. Before the last version, the ALE hasn't been competely deterministic, but now the emulator is 100% deterministic.

3 Implementation

The implementation of the Deep Q-Learning Newtwork started using Tensorflow and Keras, but getting a lot of memory issues and being extremely slow in the training, I decided to move to Pytorch and to use some methods in OpenAI Baselines, following the principles of Mnih et al. [3]. Here the pseudocode of the algorithm:

```
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 $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the
    network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  **End For**
**End For**
```

Figure 1: Deep Q-learning with experience replay pseudocode.

The idea is to create parallel environments of the game, in a way to explit the GPU and get more frames of the game in less time. To do this I used the SubprocVecEnv from OpenAI Baselines, a vectorized environment for stacking multiple independent environments into a single environment. Following Mnih et al. [3], the agent should interact with the environment 4 steps in between gradient steps, so I set the number of parrarel environments to be 4. SubprocVecEnv also resets automatically the environment when it's done.

$$L_1(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{if } |y - f(x)| \leq 1 \\ |y - f(x)| - \frac{1}{2} & \text{otherwise} \end{cases} \tag{1}$$

Always following Mnih et al. [3], the error term should be clipped between -1 and 1, but here there is a little bit of confusion in literature. In the implementation of DeepMind, they clipped the error term, while in newer implementations like OpenAI Baselines, they use the Huber Loss, which changes from the squared error to the absolute error when loss is smaller or greater than 1. I decided to use the Huber Loss and the formula used is in Equation 1.

3.1 Preprocessing

Preprocessing is needed to feed the network with "lighter" images, and to follow some principles underlined by Mnih et al. [3]. The steps are:

1) Frame skipping
2) Grayscale and rescale (84x84)
3) Reward clipping [-1,1]
4) Frame Stacking (k=4)
5) No-Op Actions at start (max=30)

The first step is done by the ALE v5 as the frame skip is set to 4 by default. The second step is done through OpenAI Baselines. Here the wrapper class WarpFrame does the job. The clipping is done through a wrapper in the OpenAI Baselines called ClipRewardEnv. The stacking is done with a modified version of the class FrameStack and the No-Op actions are performed by the class NoopResetEnv, always from OpenAI Baselines.
In addition to that, 2 different crop have been tested. In the cropV1 the "earth" portion of the screen and some black lateral and top borders are removed, while in the cropV2 the score has been cropped out.

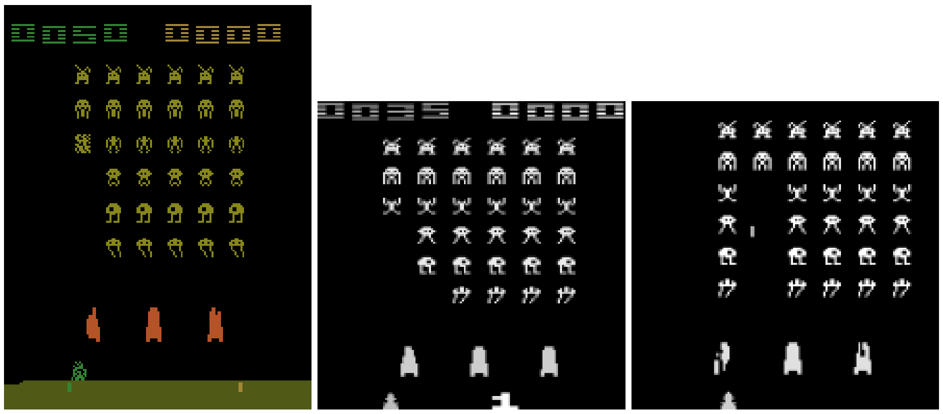
Figure 2: Steps from original to grayscale and cropped frames. From the left: original image, grayscale with cropV1 and grayscale with cropV2.

3.2 Convolutional Neural Network

The input to the neural network consists of an 4 x 84 x 84 image produced by the preprocessing. Since I use Pytorch, the order needed is Channel-Height-Width (C, H, W), and not Height-Width-Channel as the output of the environment. To convert the order, I made an ObservationWrapper which transposes the channels.The first hidden layer convolves 32 filters of 8 x 8 with stride 4 with the input image and applies a rectifier nonlinearity . The second hidden layer convolves 64 filters of 4 x 4 with stride 2, again followed by a rectifier nonlinearity. This is followed by a third convolutional layer that convolves 64 filters of 3 x 3 with stride 1 followed by a rectifier.The final hidden layer is fully-connected and consists of 512 rectifier unit. With a first pass through, I calculate the exact number of units in input for this layer. The output layer is a fully-connected linear layer with a single output for each valid action. The number of valid actions are 6 in Space Invaders.

3.3 Deep Q-Learning Networks (DQN) and Double Deep Q-Learning Networks (DDQN)

Deep Q-Learning Networks use two networks, the online network and the target network. The target network, with weights θ^- in Equation 2 taken from Mnih et al. [3], is used to compute the q values and the highest q value to select the action, while the gradient descent step is done on the online network. Every certain number of steps, the target network is updated with the weights of the online network.

$$y_j = \begin{cases} r_j & \text{if episode terminates at step j + 1} \\ r_j + \gamma max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases} \tag{2}$$

In Double Deep Q-Learning Networks, the online network does the action selection (online weights θ_t in Equation 3 taken from Van Hasselt et al. [4]), while the target net is used to compute the q values of that action (target weights θ'_t).

$$Y_t^{DoubleQ} \equiv R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta_t); \theta'_t) \tag{3}$$

Both DQN and DDQN are implemented in this work, with possible selection of one of the two by using the Double parameter of the agent class.

3.4 Training

The training is done by a gradient descent step every 4 steps (taken by the 4 different parallel environments). The longer training lasted one day and 22 hours for a total of 13.85 million steps. The hyperparameters used are the following:

NUM_ENVS = 4
GAMMA = 0.99
BATCH_SIZE = 32
BUFFER_SIZE = 300000
MIN_REPLAY_SIZE = 50000
EPSILON_START = 1.0
EPSILON_END = 0.1
EPSILON_DECAY = 300000
TARGET_UPDATE_FREQ = 10000 // NUM_ENVS

BUFFER_SIZE has been set to 300000 and not 1000000 as in Mnih et al. [3], due to memory limitation. In fact, even if I used the LazyFrames, which save memory keeping pointers to the frames and not the frames themselves (if the same frame is encountered, the pointer points to the same memory area), during the training the program used 9.6 GB of memory. EPSILON_DECAY should be 1000000 as stated Mnih et al. [3], but since in the paper it was equal to the size of the replay memory, I set it to be 300000. TARGET_UPDATE_FREQ has been set to 10000 divided by the number of environments, since in the DeepMind paper is stated out that it's the number of environment steps and not the number of parameter updates. Since I have 4 environments taking a step in between each parameter update, I need to divide the update frequency by the number of environments.
To normalize the values, and so speed up learning and convergence, I used the ScaledFloatFrame wrapper included in the OpenAI baselines, scaling the observation values between 0 and 1 rather than 0 and 255.

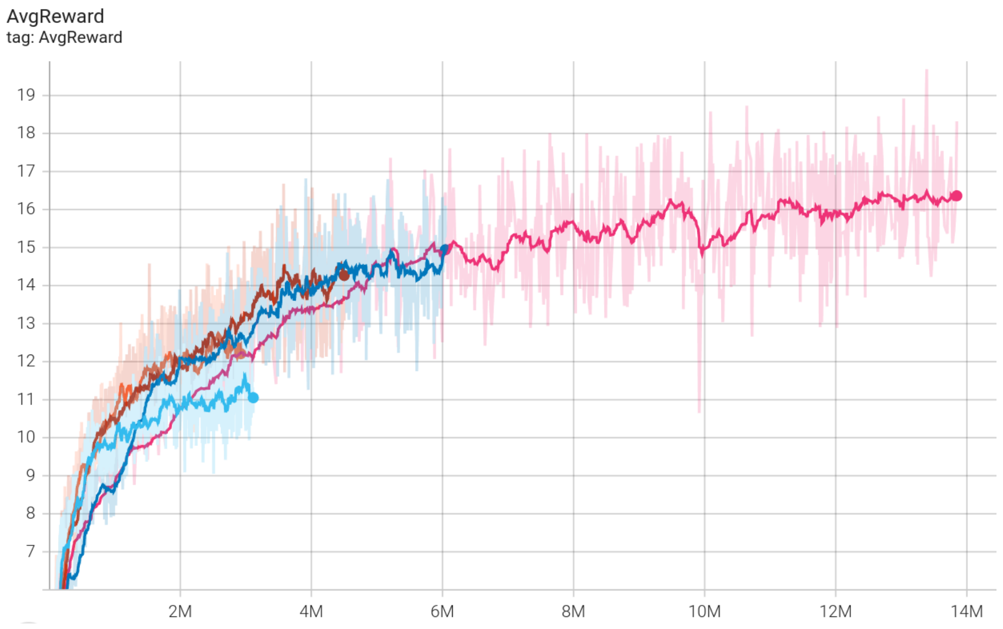

Figure 3: Graph of the average clipped reward. Curve smoothed by 95%.

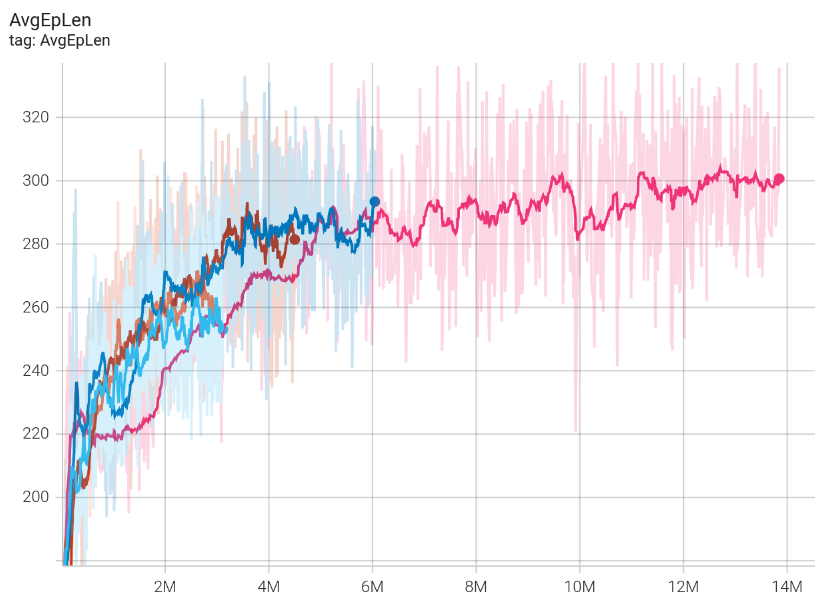

Figure 4: Graph of the average length of each episode. Curve smoothed by 95%.

In Figure 3 can be seen the rise of the average reward and in Figure 4 the average length of episodes. Colors refer to different configurations and attempts:

🟧 DQN with learning rate at 5e-5 and no crop
🟦 DQN with learning rate at 2.5e-5 and no crop
🟥 DQN with learning rate at 2.5e-5 and cropV1
🟦 DDQN with learning rate at 5e-5 and cropV1
🟪 DQN with learning rate at 2.5e-5 and cropV2

All configurations follow almost the same growth pattern in the reward graph, and some have been stopped earlier when they weren't good enough.

4 Evaluation of the agent

The agent has been evaluated playing the game for 30 episodes. The results are the following:

Rewards: [1110,1230,1160,1080,1200,1205,750,1175,985,1170,1320,550,960,1375,
1430,1140,695,1150,960,1440,495,1230,1730,1525,1230,1595,1400,1200,1025,1200]

Avg reward: 1157.1666
Standard deviation: 276.07877

The result is lower than the one reached by DeepMind, which have an average reward of 1976 and a standard deviation of 893.

5 Conclusions and further improvements

As can be seen in the Figure 3, all the configuration have almost the same reward curve, with only the DDQN (the cyan line) which asymptotized at lower values. With the cropV1 and cropV2 the network should have reached convergence faster, due to the smaller state space, but looking at the graphs, this is not the case. A further improvement would be to try a dueling DQN and check if it can reach even higher rewards. I think one of the main lack in this work, is the bigger 1000000 size replay memory, due to hardware limitations. Another test, in fact, would be to try the same code on a better performing machine with one million replay memory. However the results obtained are satisfactory, with a relatively fast training time.

References

[1] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[2] M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, and M. Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, 2018.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[4] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.