

Activity 8

Deep Learning Lab

December 6, 2019

1 Assignment 4

In this assignment, you will implement and train a Deep Q-Network (Mnih et al., 2015) agent. You should read and understand the paper before you start working on the assignment.

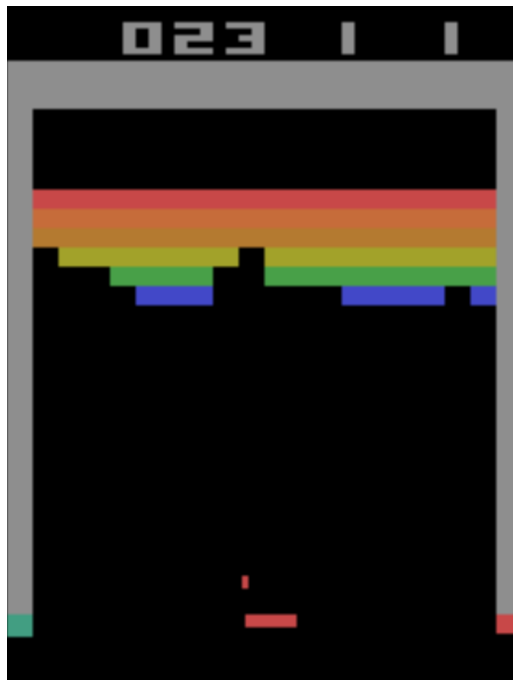


Figure 1: Atari Breakout gameplay.


1.1 Environment

You will use OpenAI Gym (Brockman et al., 2016) to create the environment. You may install gym and gym[atari] (needed for the next steps) with “`pip3 install gym gym[atari]`”.

A good introduction to the Gym can be found [here](#). You should become familiar with the main Gym environment methods, such as `reset`, `step`, and `render`. As an exercise, try to create a `CartPole-v0` environment, and render the results of executing a couple of actions.

For the remainder of this assignment, you will use the `BreakoutNoFrameskip-v4` environment. The observations provided by this environment are RGB images (frames) containing 210×160 pixels. An example frame is shown in Figure 1.

In order to preprocess the frames provided by the environment, you will need to use [Gym wrappers](#), which are used to modify the interface of an environment. The following steps allow you to implement an interface consistent with the paper:

- Copy [this](#) module to your project directory.
- Create an additional function in this [module](#) called `wrap_atari_deepmind`: 
 - Your new function should receive the environment name (in this assignment, `BreakoutNoFrameskip-v4`), and a boolean variable that indicates whether the reward should be clipped. The reward should only be clipped during training, not evaluation. The distinction will become clear later.

- Your new function should use *make_atari* to create the environment by name, and wrap the resulting environment with the *wrap_deepmind* function. You will use *episode_life=True*, *frame_stack=True*, *scale=True*, and *clip_rewards* as specified from the input to your new function.
- Your new function should return the wrapped environment.

Note that *make_atari* and *wrap_deepmind* use several wrappers internally, such as the frame stack wrapper, image scaling wrapper, and clip reward wrapper. Make sure you understand the role of each of these wrappers.

1.2 Agent

Your DQN agent will have 3 main components: online *Q*-network, **target** *Q*-network, and replay buffer.

- Implement the online *Q*-network with the following architecture:
 - Convolutional layer 1: 32 filters, 8×8 , stride 4, padding “*SAME*”, rectified linear activations.
 - Convolutional layer 2: 64 filters, 4×4 , stride 2, padding “*SAME*”, rectified linear activations.
 - Convolutional layer 3: 64 filters, 3×3 , stride 1, padding “*SAME*”, rectified linear activations.
 - Fully connected layer 1: 512 rectified linear units.
 - Fully connected layer 2: k linear units, where k is the number of actions.

Initialize the weights with *tensorflow.python.ops.init_ops.VarianceScaling* and the biases with *tf.zeros_initializer*. You can accomplish that by creating an object of each of these classes, and calling the object as if it were a function to create a tensor of a given shape. This tensor can then be used to initialize a variable.

- Implement the target *Q*-network with the same architecture as the online *Q*-network. The weights of the target network will be updated only every C steps, where C is a hyperparameter (Algorithm 1). In order to make the parameters of the two networks more easily accessible, you may optionally use a **tf.variable_scope**. Otherwise, you should create two lists that separately store variables for each network (see next step).
- Create an operation that is able to copy the parameters of the online *Q*-network to the target *Q*-network. This operation can be represented by a list of *tf.assign* operations. Each assign operation assigns one (tensor) variable of the online *Q*-network to the corresponding (tensor) variable of the target *Q*-network.
- Implement a *replay buffer* that stores the last 10,000 transitions, each of which is composed of state, action, reward, next state, and termination flag. During training, you will create batches by sampling from the replay buffer. Hint: when new elements are added to a **deque** with a maximum number of elements, a corresponding number of elements is discarded from its opposite end.

1.3 Training

As a next step, you will implement the training algorithm (Algorithm 1). Note the following:

- The discount factor should be $\gamma = 0.99$.
- The agent should interact with the environment for a total of $N = 2,000,000$ steps.
- The exploration rate should start at $\epsilon = 1$, and should be linearly reduced down to $\epsilon = 0.1$ during the first 1,000,000 steps. Afterwards, the exploration rate should be kept at $\epsilon = 0.1$. This process is not shown in Algorithm 1 for simplicity.
- The networks are not updated until the replay buffer is populated with $M = 10,000$ transitions.
- Once the replay buffer is populated:
 - Every $n = 4$ steps, sample a batch composed of $B = 32$ transitions from the replay buffer. Using this batch, update the parameters of the online Q -network to minimize the loss $L(\theta)$ defined in Algorithm 1. Use **RMSprop** (*tf.train.RMSPropOptimizer*) with a learning rate of $\alpha = 0.0001$ and a decay of 0.99. This process is simplified in Algorithm 1.
 - Every $C = 10,000$ steps, copy the parameters of the online network to the target network.

Algorithm 1 Deep Q-learning with experience replay

Input: number of steps N , replay buffer size M , probability of random action ϵ , discount factor γ , batch size B , learning rate α , number of steps between online Q -network updates n , number of steps between target Q -network updates C .

Output: estimate $Q(\cdot; \theta)$ of the optimal action-value function Q^*

```

1: Initialize replay buffer  $\mathcal{D}$ , which stores at most  $M$  tuples
2: Initialize network parameters  $\theta$  randomly
3:  $\theta' \leftarrow \theta$ 
4:  $i \leftarrow 0$ 
5: while  $i < N$  do
6:    $s_0 \leftarrow$  initial state for a new episode
7:   for each  $t$  in  $\{0, 1, 2, \dots\}$  do
8:     if  $\text{random}() < 1 - \epsilon$  then  $a_t \leftarrow \arg \max_a Q(s_t, a; \theta)$  else  $a_t \leftarrow$  random action
9:     Obtain the next state  $s_{t+1}$  and reward  $r_{t+1}$  by taking action  $a_t$ 
10:    if the episode ends at step  $t + 1$  then  $\Omega_{t+1} \leftarrow 1$  else  $\Omega_{t+1} \leftarrow 0$ 
11:    Store the tuple  $(s_t, a_t, r_{t+1}, s_{t+1}, \Omega_{t+1})$  in the replay buffer  $\mathcal{D}$ 
12:     $i \leftarrow i + 1$ 
13:    if  $i \geq M$  then
14:      if  $i \bmod n = 0$  then
15:        Sample a subset  $\mathcal{D}' \subset \mathcal{D}$  composed of  $B$  tuples
16:        Let  $L(\theta) = \sum_{(s,a,r,s',\Omega') \in \mathcal{D}'} (y - Q(s,a;\theta))^2$ 
17:        In the equation above, let  $y = r + \gamma \max_{a'} Q(s', a'; \theta')$  if  $\Omega' = 0$ , and  $y = r$  if  $\Omega' = 1$ 
18:         $\theta \leftarrow \theta - \alpha \nabla_{\theta} L(\theta)$ , noting that  $\theta'$  is considered a constant with respect to  $\theta$ 
19:      end if
20:      if  $i \bmod C = 0$  then
21:         $\theta' \leftarrow \theta$ 
22:      end if
23:    end if
24:    if  $\Omega_{t+1} = 1$  then
25:      break (end of episode)
26:    end if
27:  end for
28: end while


```

1.4 Tasks

1. (10 points) Several Gym wrappers were required to preprocess the data. These wrappers include the *FrameStack* wrapper, the *ScaledFloatFrame* wrapper, the *MaxAndSkipEnv* wrapper, and the *ClipRewardEnv* wrapper. Briefly explain the role of each of these wrappers (in one or two sentences).
2. (10 points) How do Mnih et al. (2015) explain the need for a target Q -network in addition to an online Q -network?
3. (10 points) Why is it necessary to act according to an ϵ -greedy policy instead of a greedy policy (with respect to Q)?
4. (50 points) Train your agent as detailed in Section 1.3. This is a computationally expensive process, and requires a GPU. If you plan on using Google Colab, see Section 3.

During training, you should monitor the number of steps elapsed, the number of episodes elapsed, and the reward obtained at each step. It is also useful to estimate the remaining training time based on the average time that each step requires.

You should plot the following metrics in your report:

- (a) Return during training: return per episode, averaged over the last 30 episodes to reduce noise (moving average).
- (b) Return during evaluation: every 100,000 steps (20 times in total), evaluate an ϵ -greedy policy based on your learned Q -function with $\epsilon = 0.001$. The rewards of the environment should *not* be clipped for this purpose. You should sum the return obtained across 5 different episodes so that you can compare your results to those listed by Mnih et al. (2015). We will call such a sequence of 5 episodes a *play*, and the corresponding sum of returns (without clipped rewards) a *score*. You should average the scores across 30 independent plays before plotting the value for each of the 20 evaluations.
- (c) Temporal-difference error $L(\theta)$ at each update step (see Algorithm 1) 

Important: After 200,000 steps, your agent should obtain an average score of 10 (approximately). If your results are much worse at this stage, contact us as soon as possible. After 2,000,000 steps, your agent should obtain an average score of 150 (approximately).

5. (10 points) After training, render one episode of interaction between your agent and the environment. For this purpose, you may wrap your environment using a `gym.wrappers.Monitor`. If your environment object is called *env*, interacting with the wrapped environment `gym.wrappers.Monitor(env, path, force=True)` will cause the corresponding video to be saved to *path*.
6. (10 points) Instead of updating the target network every $C = 10,000$ steps, experiment with $C = 50,000$. Compare in a single plot the average score across evaluations obtained by these two alternatives. How do you explain the differences?
7. (10 points) **Bonus:** Repeat Steps 4-5 for a different Atari game.
8. (10 points) **Bonus:** Write your own wrapper for an Atari game. This wrapper should transform observations or rewards in order to make it much easier for Algorithm 1 to find a high-scoring policy.
9. (10 points) **Bonus:** Using `record_agent.py` as a guide (see iCorsi3), record your *personal* gameplay for 10,000 steps. Use this data to populate the replay buffer, and train your agent for 300,000 steps. Compute the average score obtained by the resulting agent.

2 Submission

You should deliver the following by the deadline stipulated on iCorsi3:

- Report: a single *pdf* file that clearly and concisely provides evidence that you have accomplished each of the tasks listed above. The report should not contain source code (not even snippets). Instead, when necessary, briefly mention which functions were used to accomplish a task.
- Source code: Python script(s) that could be easily adapted to accomplish each of the tasks listed above. The source code will be read superficially and checked for plagiarism. Unless this reveals that your code is suspicious, your grade will be based entirely on the report. Therefore, if a task is accomplished but not documented in the report, it will be considered missing. Note: Jupyter notebook files are not acceptable.
- Video: a video representing one episode of interaction between your trained agent and the environment.

3 Google Colab tips

Here are some tips for using Google Colab to complete this assignment:

- Make sure to **enable** GPU usage.
- It is easy to **mount** your Google Drive in order to transfer files from/to your personal computer. For example, you can use this to save models and evaluate them locally.
- The training process may take from 2 to 6 hours, depending on your implementation. Google Colab notebooks have an idle timeout of 90 minutes and an absolute timeout of 12 hours. In practice, it is possible to **bypass** these limitations.

References

- Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). OpenAI gym.
- Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529.