Deep Learning Lab
Università della Svizzera Italiana
Faculty of Informatics
December 20, 2019

Assignment 4: Deep Q-Network

Giorgia Adorni (giorgia.adorni@usi.ch)

1 Introduction

The goal of this project is to implement and train a Deep Q-Network agent, based on *Mnih et al., 2015*.

All the models were implemented using TensorFlow and trained on an NVIDIA Tesla V100-PCIE-16GB GPU.

2 Environment, Agent and Training

`OpenAI Gym` has been used to create the *BreakoutNoFrameskip-v4* environment.

The DQN agent has three main components: an online Q-network, a target Q-network and replay buffer. The two networks are used to improve the stability of this method, in particular, every C steps, the target network is updated with the online network parameters.

The replay buffer of capacity 10 000 is composed of state, action, reward, next state, and termination flag. It will be used during the training in order to create batches by sampling them from the buffer.

In Table 1 is summarised the architecture of both the networks.

conv1	**conv2**	**conv3**	**fc1**	**fc2**
8×8, 32	4×4, 64	3×3, 64	512	k
s. 4×4	s. 2×2	s. 1×1		
p. same	p. same	p. same		
ReLU	ReLU	ReLU	ReLU	ReLU

Table 1: Network architecture

3 Tasks

Wrappers

- The `FrameStack` wrappers is used to stack the last k frames returning a lazy array, which is a structure much more memory efficient.

- The `ScaledFloatFrame` wrapper rescales the value of the pixels initially comprised between 0 and 255 between 0-1.
- The `MaxAndSkipEnv` wrapper is used to reduce the number of frame, hence the quantity of data to process. In order to do this, firstly it skips some frames of the game play. Then, for all the skipped frames, it continually repeats the same action, sums the rewards in order not to lose information and at the end, for each pixel of the frame, the maximum pixel value is chosen among all the frames skipped.
- The `ClipRewardEnv` wrapper classifies the reward as $+1$, 0 or -1 according its sign.

Online Q-network and Target Q-network

As mention in Section 2, the addition of the target network, the this used to generate targets in the Q-learning update, improves the stability of the presented method. In particular, every C steps, the target network is updated with the online network parameters. This procedure adds a delay between the time between the time the network is updated and the time when the update affects the target, making divergence or oscillations much more unlikely.

ϵ-greedy policy

Acting according to an ϵ-greedy policy ensure an adequate exploration of the environment in order to learn about potentially (could be better or worse) new sources of reward, instead of exploit the well-known sources of reward.

For a given state s, the ϵ-greedy policy with respect to Q chooses a random action with probability ϵ, and an action $\arg\max_a Q(s, a)$ with probability $1 - \epsilon$.

4 Experiments

Experiment 1

The first experiment includes a training phase in which the agent interact with the environment for a total of 2 000 000 steps.

The replay buffer is initially empty, and the networks are not updated until it is populated with 10 000 transitions.

Every 4 steps, a batch composed 32 transitions is sampled from the replay buffer and used to update the parameters of the online Q-network. Root mean square prop (`RMSPropOptimizer`) is used as optimiser to minimise the temporal-difference error, with learning rate of 0.0001 and a decay of 0.99.

The parameters of the online network are copied to the target network every 10 000 steps.

Figure 1 shows the number of steps elapsed in each episode averaged over the last 100 episodes.

It is clearly visible how the number of steps for episode increase over time. Since each episode corresponds to a "life of play", a greater number of steps per episode can be interpreted as an improvement in the network's ability to play the game without losing a life quickly.

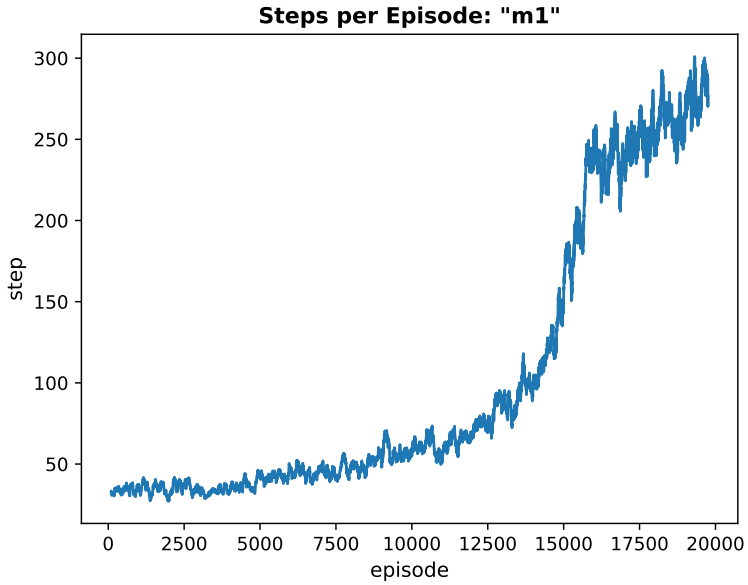

Figure 1: Steps per episode of the first experiment

Figures 2 shows the return per episode, averaged over the last 30 episodes (moving average) to reduce noise, while Figure 3 shows scores across 30 independent plays, that correspond to the sum of the return obtained across a sequence of 5 different episodes.

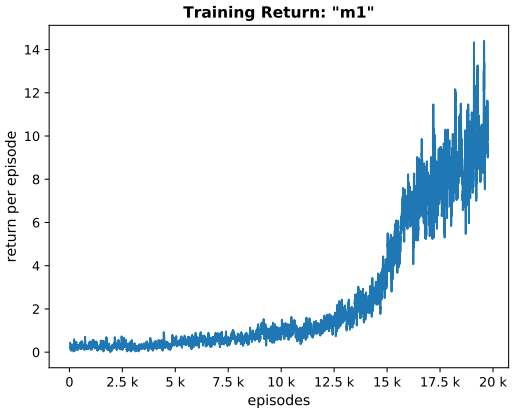

Figure 2: Return per episode

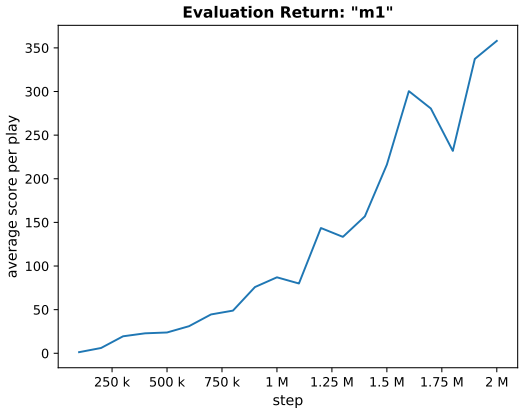

Figure 3: Average score per play

The game score obtained is 358.0 that is, as expected, a little lower compared to the one achieved in literature, that is 401.2 ($\pm$26.9), since the model presented in the paper is much more complex. Moreover, the trend of the curve is growing, which means that further increasing the number of steps, the score should continue to improve.

In Figure 4 is shown a subsample of the temporal-difference error $L(\theta)$ averaged over the last 50 steps in order to reduce noise.

The training process takes 5 hours and 18 minutes.

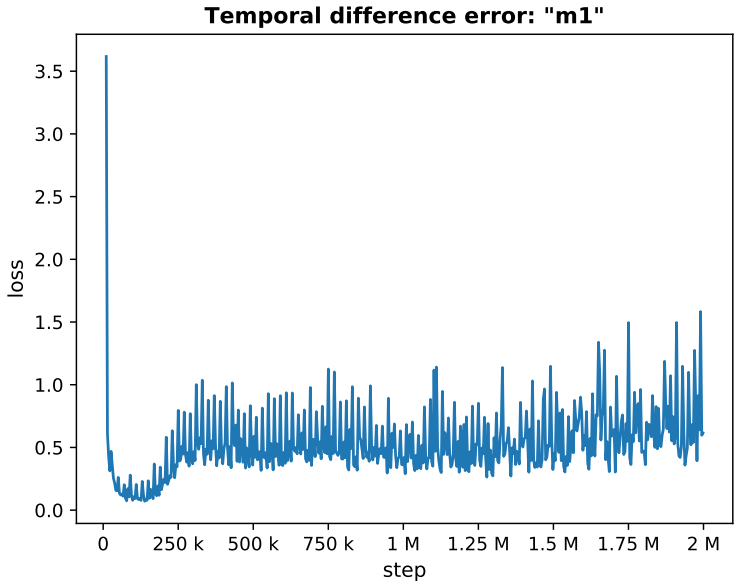

Figure 4: Loss of the first experiment

Experiment 2

The second experiment performed modifies only the value of the parameter C with respect to the previous experiment. In particular, the target network is updated every 50 000 instead of 10 000.

The training process takes 4 hours. Figure 5 shows a comparison among the evaluation scores of the experiments.

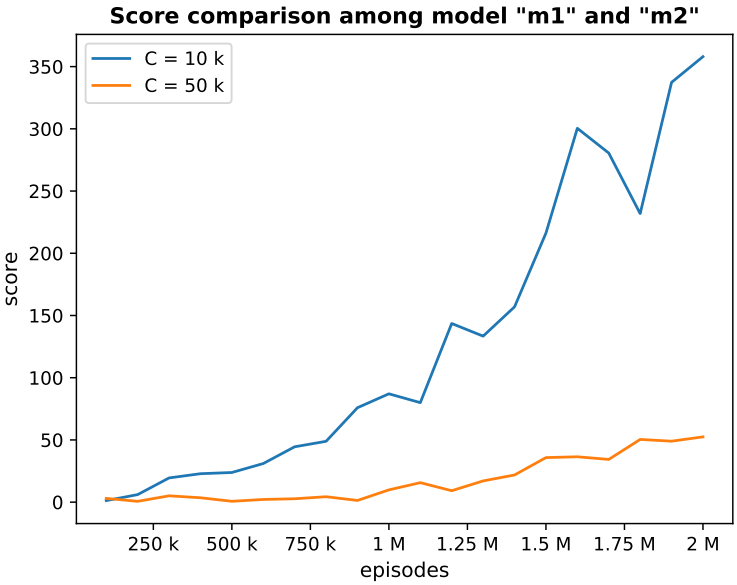

Figure 5: Score comparison

It is clearly visible how, by reducing the frequency of the updates, the network slows down the learning.

Experiment 3

The third experiment repeats the same training procedure as the first experiment using a different environment. The Atari game tried in this case is *StarGunner*. The training process takes 4 hours and 11 minutes.

Figures 6 and 7 show the return per episode (moving average) and scores across 30 independent plays.

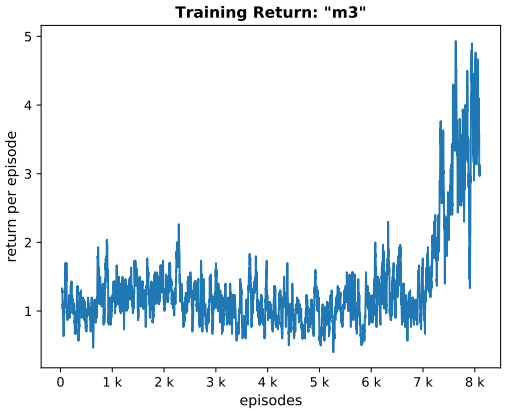

Figure 6: Return per episode

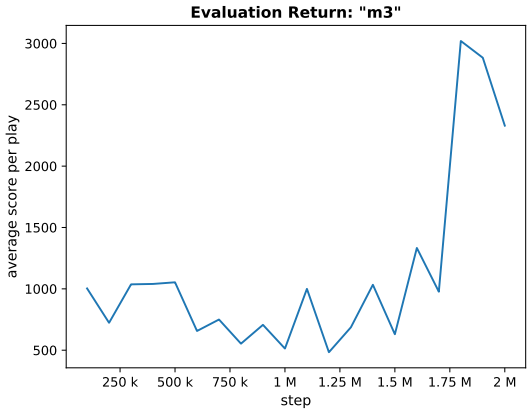

Figure 7: Average score per play

Also in this case, the return curves have a growing trend. After $1.5M$ steps the score increases very quickly, with a peak at 3020.0 and then slightly decreases around $2M$ steps.

In Figure 8 is shown a subsample of the temporal-difference error $\mathrm{L}(\theta)$ averaged over the last 50 steps.

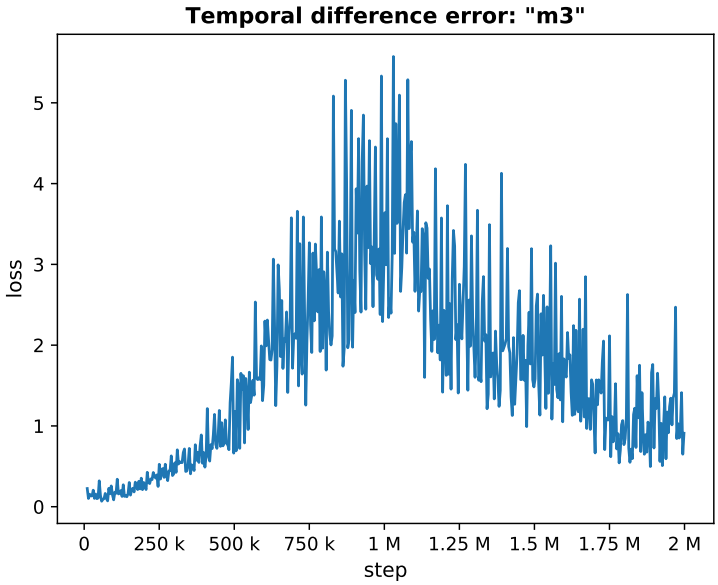

Figure 8: Loss of the third experiment

In this case, the loss does not have a low value nor a decreasing trend as for the first experiment.

Experiment 4

The last experiment tries to improve the results for `Breakout` obtained in the first experiment, using the data recorded from 10 000 gameplay steps to populate the replay buffer. In this case the model is trained for 300 000 steps and the process takes 31 minutes.

Figures 9 and 10 show the return per episode (moving average) and scores across 30 independent plays

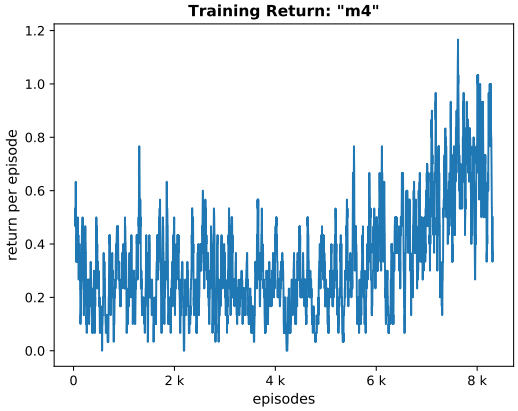

Figure 9: Return per episode

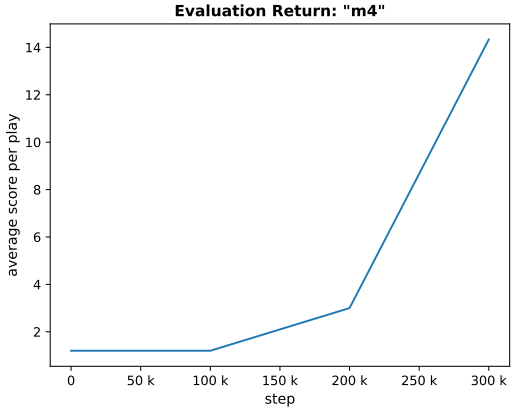

Figure 10: Average score per play

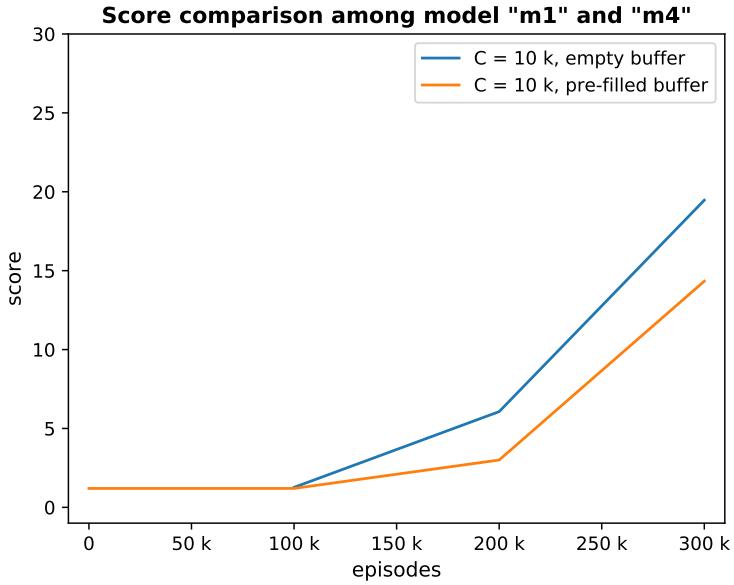

Figure 11: Score comparison

The evaluation return obtained during this experiment is 14.33, that is similar to the one obtained in the first experiment at the same time step, that is 15.03.

The similar results are probably due the limited capacity of the replay buffer. In fact, after only 10 000 steps, the past data experiences are replaced.