Sapienza University of Rome

Artificial Intelligence
Solving Pong using Deep-Q Learning

Michele Cipriano

March 13, 2018

1 Introduction

In the last years, reinforcement learning has seen an increasing interest thanks, in particular, to GPUs and the diffusion of deep learning. The aim of this project is to study the Deep-Q Learning with Experience Replay algorithm, introduced in [1], and train the Deep-Q Network on the game of Pong, using OpenAI Gym environments[2].

In particular, different experiments have been performed comparing the algorithm of [1] and its successor, introduced in the famous Nature paper [3]. The networks have been trained on two different Pong environments, which differs in the way the agent interacts with the them.

The project has been developed entirely in Python, using TensorFlow to define and train the neural networks and Gym to interact with the Atari Pong environments. The training phase has been performed on Google Compute Engine, which provides an easy access to GPU computational power.

The experiments have been done on two slightly different Pong environments. The best network obtained a maximum average score (averaged over 25 episodes) of 16.92 on `PongNoFrameskip-v0` analizing 2.2M frames learning a good strategy just after 600,000 frames.

2 Gym

OpenAI Gym is a toolkit for reinforcement learning research which includes a collection of environments. The environments share a common interface with which the agent can interact. In particular, at each timestep, the agent performs an action receiving an observation, a reward and a boolean value telling whether the observation is final with respect to the episode or not.

Part of this environments are classic Atari games and make use Arcade Learning Environment. This project makes use of two similar Pong environments,

`Pong-v0` and `PongNoFrameskip-v0`. Both returns as observation an RGB image of shape $(210, 160, 3)$. The difference is within the number of times the action is repeated: while in `Pong-v0` each action is repeatedly performed for a duration of k frames, where k is uniformely sampled from $\{2, 3, 4\}$, in `PongNoFrameskip-v0` each action is performed for a single frame.

Another interesting tool used in this project are OpenAI Atari wrappers[4], which simplify the management of the frames, providing preprocessing and different levels of abstraction on the observations. The main wrappers used are:

- **WarpFrame**: warp frame to 84×84 as done in [3].
- **FrameStack**: stack k last frames, this wrapper is used to obtain tensors of size $84 \times 84 \times 4$ that can be easily fed to the DQN.
- **MaxAndSkipEnv**: return an observation only every k-th frame. This wrapper has been used only with `PongNoFrameskip-v0` to speed up training by repeating the same action $k = 4$ times, reducing stochasticity w.r.t. `Pong-v0`.

3 Deep-Q Learning

Let's consider a task in which an agent interacts with an environment $\mathcal{E}$, in this case an Atari emulator with the Pong game, by taking at each timestep t an action a_t from a set of action $\mathcal{A} = \{1, \dots, K\}$. After the action has been performed, the agent observes an image $x_t \in \mathbb{R}^d$ and receives a reward r_t, representing the change in the game score.

Let's consider finite sequences of actions and observations $s_t = x_1, a_1, x_2, \dots, a_{t-1}, x_t$ for each timestep t and define a Markov Decision Process (MDP) where each sequence is a distinct state. The future discounted reward at time t, defined as $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, where γ is the discount factor can be used to define the optimal action-value function:

$$Q^*(s, a) = \max_{\pi} \mathbb{E}\left[R_t \mid s_t = s, a_t = a, \pi\right]$$

where π is a policy mapping sequences to actions. Since this function can also be defined by using the *Bellman optimality equation*:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s', a') \middle| s, a\right]$$

it's possible to estimate it by using an iterative update:

$$Q_{i+1}(s, a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q_i(s', a') \middle| s, a\right]$$

converging to the optimal action-value function $Q_i \to Q^*$ as $i \to \infty$. Of course, when the state space is too big, like in the case of Pong, it's impossible to store all the values in a single table. Moreover, the iterative update would take too much time to converge since no generalization is done on the problem. This problem can be easily handled by approximating the Q function with a function $Q(s, a; \theta) \approx Q^*(s, a)$ which depends on the weights θ. In this project $Q(s, a; \theta)$ has been defined by using the deep neural networks (Q-network) described in [1] and [3].

In the first experiment the neural network has been trained by minimizing a sequence of loss functions that changes at each iteration i:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right] \tag{3.1}$$

where $y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})\right]$, $\rho(s,a)$ is the probability distribution over sequences s and actions a and θ_{i-1} are the parameters of the network at iteration $i-1$ which are held fixed during the optimization step.

A simple strategy to replace the expectations and to simplify the computation of the gradient of the loss function (in order to perform the optimization) is to sample data from the distribution ρ and the emulator $\mathcal{E}$. This is done via a technique called *experience replay*, which consists in storing in a dataset $\mathcal{D} = e_1, e_2, \ldots, e_N$ the experience $e_t = (s_t, a_t, r_t, s_{t+1})$ of the agent at each timestep t. This allows to perform each gradient descent step using a minibatch of experiences $e \sim \mathcal{D}$ sampled from the experience replay dataset.

Let's now describe the Deep Q-learning with Experience Replay algorithm (algorithm 1) introduced in [1]. The main cycle executes M episodes interacting with the Gym environment and updating the weights θ of the neural network. First, the sequence is initialized (line 4) with the first frame returned by the environment, which is itself preprocessed by a function ϕ which makes it feedable to the neural network. The function ϕ, as described in [1], takes as input a RGB frame of shape $(210, 160, 3)$ and return a frame of shape $(84, 84, 1)$ after converting the input to a greyscale image, downsampling it to a $(110, 84, 1)$ image and cropping an $(84, 84, 1)$ region that captures the playing area. To maximize the performances of the program, the preprocessing has been done with the OpenAI Atari Wrappers[4], described above, which also stack the last four observations (frames) returned by the environment. The agent selects an action a_t using an epsilon-greedy strategy (lines 6-7), where ϵ is linearly descreased for the first 100,000 frames from 1.0 to 0.02, and executes it in the environment, receiving a reward r_t and observing an image x_{t+1} (line 8). The sequence is preprocessed as described above and the new transition is stored in the experience replay $\mathcal{D}$ (lines 9-10). At this point a minibatch of transitions is sampled from $\mathcal{D}$ and it is used to performed a gradient descent step on the loss function (lines 11-13). Note that the target y_j changes depending on the last observed frame[1] (if it's final or not). To have a comparison with a fast convergence implementation (see [5]) the optimization algorithm used has been Adam (with learning rate $\eta = 0.0001$), instead of RMSProp as originally proposed in the paper, which has faster convergence properties w.r.t RMSProp[6] even if it is less stable in this scenario. The training has been started after 10,000 observations.

As said before, the function Q is parametrized by a neural network. Its architecture is pretty simple, it consists of a hidden layer which convolves 16 8×8 filters with stride 4 applying nonlinearity through the rectified linear unit ($ReLU(x) = \max(0, x)$), followed by another convolutional layer, which convolves 32 4×4 with stride 2 applying $ReLU$ nonlinearity, followed by a fully connected layer of 512 $ReLU$ units[2] and an output layer with a single output for each valid action of the selected environment (6 in the case of Pong).

The network has been trained on the environment `Pong-v0` for 1,000 episodes (around 3M frames) using a batchsize of 32 elements and a discount factor

[1] The experience replay implementation also stores the boolean `done` which denotes the end of the episode.

[2] In the original paper the final hidden layer has size 256.

Algorithm 1 Deep Q-learning with Experience Replay[1]

1 Initialize replay memory $\mathcal{D}$ to capacity N
2 Initialize action-value function Q with random weights θ
3 **for** episode $= 1, M$ **do**
4 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
5 **for** $t = 1, T$ **do**
6 With probability ϵ select a random action a_t
7 otherwise select $a_t = \max_a Q(\phi(s_t), a; \theta)$
8 Execute action a_t in emulator and observe reward r_t and image x_{t+1}
9 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
10 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
11 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
12 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a', \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
13 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
14 **end for**
15 **end for**

$\gamma = 0.99$ reaching an average reward per episode (computed over 25 episodes) of -4.84 as it is possible to see from figure 3.1.

Let's now improve the algorithm by adding a target network and changing the loss function and the architecture of the network as done in [3]. The idea is to improve the training by improving the stability of the algorithm.

Let's change the definition of the loss function in 3.1 by:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim U(\mathcal{D})} \left[L_\delta \left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right) \right]$$

where $U(\mathcal{D})$ is the uniform distribution describing the experience replay, θ_i^- are the parameters of the target network, updated only every C steps and hence kept fixed during the optimization steps, θ_i are the parameters describing the action-value function Q (note that in algorithm 2 the target network is denoted as $\hat{Q}$) and L_δ is the Huber loss function, defined as:

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \leq \delta \\ \delta \left(|a| - \frac{1}{2}\delta \right) & \text{otherwise} \end{cases}$$

which behaves quadratically for small values of a and linearly for larger values. The architecture of the action-value function Q is similar to the one described above. It consists of a convolutional layer which convolves 32 8×8 filters with stride 4 applying $ReLU$ nonlinearity, followed by another convolutional layer which convolves 64 4×4 filters with stride 2 applying $ReLU$, followed by a third convolutional layer which convolves 64 3×3 filters with stride 1 applying again $ReLU$. The three convolutional layers are followed by a fully connected layer of 512 $ReLU$ units and an output layer of the size corresponding to the number of valid action of the environment.

The resulting algorithm (algorithm 2), described below, is similar to the previous one, except for the target action-value function $\hat{Q}$, initialized on line 3, used to compute the target value y_j on line 13 and updated every C steps on

line 15, the weights θ^-, that parametrize the function $\hat{Q}$, and the loss function (line 14), which is the Huber loss instead of the squared loss.

The network has been trained on the environment `Pong-v0` for 1,000 episodes (around 3.6M frames) using a batchsize of 32 elements and a discount factor $\gamma = 0.99$ reaching an average reward per episode (computed over 25 episodes) of 0.2 as it is possible to see from figure 3.1. The Huber loss function used a value of $\delta = 1$. Algorithm 2 slightly improves algorithm 1 performing much better in particular in the initial phase, where the agent explores the environment more frequently.

Algorithm 2 Deep Q-learning with Experience Replay[3]

1 Initialize replay memory $\mathcal{D}$ to capacity N
2 Initialize action-value function Q with random weights θ
3 Initialize target action-value function $\hat{Q}$ with random weights $\theta^- = \theta$
4 **for** episode $= 1, M$ **do**
5 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
6 **for** $t = 1, T$ **do**
7 With probability ϵ select a random action a_t
8 otherwise select $a_t = \max_a Q(\phi(s_t), a; \theta)$
9 Execute action a_t in emulator and observe reward r_t and image x_{t+1}
10 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
11 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
12 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
13 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a', \theta^-) & \text{for non-terminal } \phi_{j+1} \end{cases}$
14 Perform a gradient descent step on $L_\delta(y_j - Q(\phi_j, a_j; \theta))$ w.r.t. the parameters θ
15 Every C steps reset $\hat{Q} = Q$
16 **end for**
17 **end for**

The same network has been trained with the same hyperparameters on a similar Pong environment (`PongNoFrameskip-v0`), with the `MaxAndSkipEnv` wrapper, resulting in a less stochastic environment than `Pong-v0`. As said before, while in `Pong-v0` each action is performed randomly for 2, 3 or 4 frames, in `PongNoFrameskip-v0` each action is performed only once for each frame. Combining this environment with `MaxAndSkipEnv` it has been possible recreate the environment used in the Nature paper, executing the same action for k frames (in the experiment $k = 4$) and making the agent observe only each k-th frame, exactly as done in [1] and [3]. The network obtained an average reward per episode of 16.92 (again, average over 25 episodes) analizing 2.2M frames (1,000 episodes). It is interesting to notice that the network learns a good action-value function in the first 600,000 frames, slightly improving until the end of the training process.

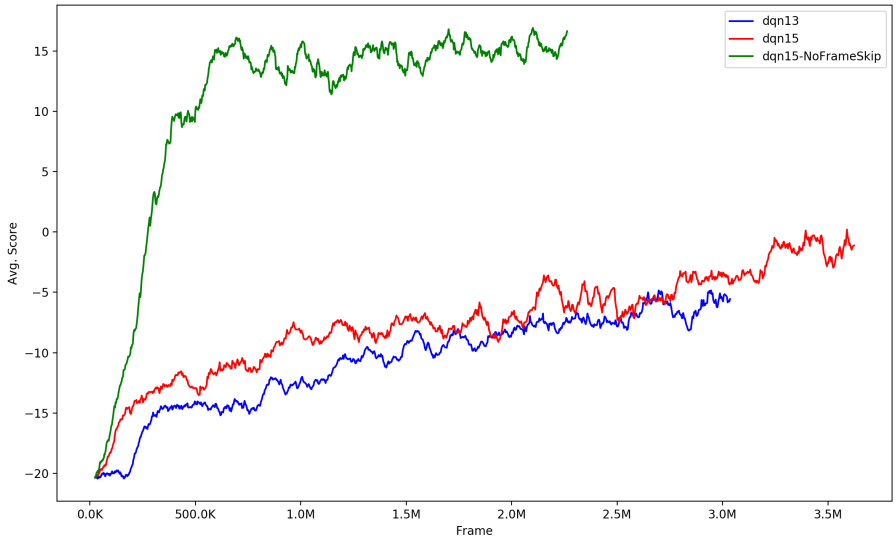

Figure 3.1: The figure shows the average score (averaged over 25 episodes) of the three described implementations. In blue the DQN of [1] trained on `Pong-v0`. In red the DQN of [3], trained on `Pong-v0`. In green the DQN of [3] trained on `PongNoFrameskip-v0`.

4 Conclusion

Deep-Q learning is a simple and powerful method to solve reinforcement learning problems. The experiments of Pong have shown not only how much important is to improve learning algorithms, in this case by introducing a target network, using a more stable loss function and redefining the convolutional neural network, but also how stochasticity can affect the performances of the network.

The experience replay memory, which provides data to the network, plays an important role in the training phase. Improving the way the data are sampled from this memory (for example by prioritizing samples from which the network can learn more[7]) could make the training more efficient, making the same network learn better action-value functions in less time.

Reinforcement learning is a hot research topic and more and more methods are introduced every day. Parallelizing the computation is essential in order to speed up training and explore the environment quickly. Exploiting the structure of the network by combining value-based and policy-based methods, like in actor-critic methods[8], seems to improve algorithms like Deep-Q learning, even without the use of GPUs. For sure, the combination of both GPUs and actor-critic methods will further improve current algorithms, making them converge faster and with more stability[9, 10].

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

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

[4] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Openai baselines." `https://github.com/openai/baselines`, 2017.

[5] "Speeding up DQN on PyTorch: how to solve Pong in 30 minutes." `https://medium.com/mlreview/speeding-up-dqn-on-pytorch-solving-pong-in-30-minutes-81a1bd2dff55`.

[6] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization.," *CoRR*, vol. abs/1412.6980, 2014.

[7] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *CoRR*, vol. abs/1511.05952, 2015.

[8] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, pp. 1928–1937, 2016.

[9] M. Babaeizadeh, I. Frosio, S. Tyree, J. Clemons, and J. Kautz, "GA3C: GPU-based A3C for deep reinforcement learning," *NIPS Workshop*, 2016.

[10] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, and K. Kavukcuoglu, "IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures," *CoRR*, vol. abs/1802.01561, 2018.