Playing Atari2600 Pong with Reinforcement Learning and Deep Q-Network

Federico Zanini
Department of Computer Science and Engineering (DISI)
University of Bologna, Italy
federico.zanini@studio.unibo.it

Abstract

This work is about creating a DQN agent, able to learn and well perform in the Atari2600 Pong, a classic Atari game. The implementation is inspired by two different Deep Mind's papers about DQN representing the "*state-of-the-art*". The code is developed on a jupyter notebook in python while the game environment is provided by OpenAi called *"OpenAi Gym"* [2].

1 Introduction

In recent years, a lot of artificial intelligence studies have been conducted on games like chess or Go. Games are particularly good for these kind of studies because sets a close and controlled environment that allows researchers to explore, study and understand better AI applications.
Among the others, also reinforcement learning has found his application in games environment having particularly good results: one example of this is the success of AlphaGo (by DeepMind) [1], that was able to beat the world champion of the board game Go; one of the most difficult board game. This exiting result pushed the developer to implement two new different versions of AlphaGo: AlphaGo Zero [1] that was a completely self-taught algorithm and Alpha Zero, that was also able to play other games like shogi or chess.
In this work we won't see anything as sophisticated as the AlphaZero algorithm, but we will take in account DQN as presented in two different papers from Deep Mind: "Human-level control through deep reinforcement learning" [3] and "Playing Atari with deep Reinforcement Learning"[4].

2 Background

2.1 OpenAI Gym

To achieve our goal, we will use a free toolkit designed to develop and test reinforcement learning algorithms named OpenAI Gym [2].
OpenAI Gym gives us a very large number of different environments such as the well known Atari 2600 games, providing a development kit that makes extremely easy to set a lot of different preferences such as frame skip, RAM or pixels input and more, also giving a standardised interface that makes the develop easier.

2.2 Deep Q-Network

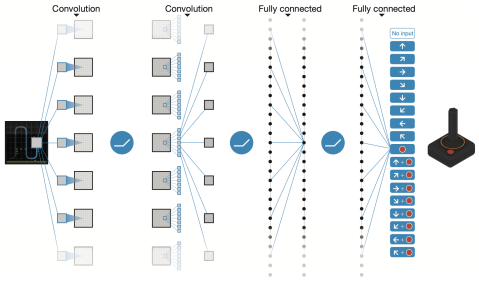


Figure 1: DQN network's architecture [3]

Deep Q-Network combines Q-learning with a deep convolutional neural network able to process an input matrix of 84×84 pixels representing the observation of the environment. To give the agent the idea of movement, that cannot be inferred from a single image, the network is fed with a stack of 4 frames, resulting into an input of $4 \times 84 \times 84$.

The goal is to learn the optimal policy in order to be able to play the game in the best possible way. To do so, the agent needs to explore a bit first and then must play in a non random way. This is why the hyperparameter *epsilon*, of the $\epsilon - greedy$ policy, is set to decrease linearly during the progress of the training, granting a good exploration.

The network is trained on a mini-batch of images, to be specific, 32 images made of 4 frames each, using RMSprop as optimizer. The mini-batch is generated from the Experience Replay buffer.

The first big difference from standard Q-learning is the usage of Experience replay, that is a "biologically inspired" mechanism of collecting the experience learned from the agent interacting with the environment at each time-step and storing it into a buffer or *replay memory buffer*:

$$e_t = (S_t, A_t, R_{t+1}, S_{t+1})$$

Q-learning updates are applied by sampling at random a mini-batch of sequences from the replay memory buffer. This approach presents several advantages like:

- Data efficiency: each sample of experience stored in the replay memory can be used for several updates.
- Pulling random batches of experience from the replay memory breaks correlations between samples reducing the variance of the updates.

Note that this is possible because Q-learning is an off-policy method and, because of that, we can update the policy (our Q-network) by training on the observation data collected in the replay buffer following another policy ($\epsilon - greedy$).

A second big modification to standard Q-learning is presented in the second of the two papers: "DQN - Human level control through deep reinforcement learning" [3], where a separate network is introduced, aiming at further improving the stability of the algorithm and used to generate the target Q-values for the Q-value update. This target Q-network has the same architecture of our Q-network and it is not trained directly but the Q-network is cloned into it every *"C"* steps.
Generating the targets with this approach makes divergence and oscillations more unlikely because an old set of parameter is used and this create a delay between the time an update Q is made and the time the update affects the targets.

Here follows a pseudo-code of the DQN algorithm presented [3]:

Algorithm 1: DQN, Deep Q-learning with experience replay

Initialise replay memory D to capacity N
Initialise action-value function Q with random weights θ
Initialise target action value function $\hat{Q}$ with weights $\theta^- = \theta$
for $episode = 1, M$ **do**
  Initialise sequence $s_1 = x_1$
  Initialise preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \text{argmax}_a \, Q(\phi(s_t), a; \theta)$
    Execute action a_t in the 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 parameter θ
    Every C steps reset $\hat{Q} = Q$

Note: In our project, we decided to implement the more complete version of the algorithm, that is exactly the one presented in the box above as **Algorithm 1** and can be found in paper [3].

3 Implementation

3.1 Design choices

Since we were having some struggles during the training of our agent, that was more disposed to learn a random policy, we decided to change some of the hyperparameters' values from the ones suggested in the paper:

- In the original implementation of DQN, RMSprop is used as optimizer with a learning rate of 0.00025. In order to have an improvement in the training speed, we opted for using Adam as optimizer, with a learning rate of 0.0001.
- The update frequency of the target network has been set to 1000 frames.
- The memory buffer of the Experience replay has been set to 10000.
- Epsilon decays from 1.0 to 0.01 in the first 100000 frames (exploration frames). In the original implementation epsilon decays from 1.0 to 0.1 in the first million frames.

These choices were made accordingly to the hyperparameters setting found in the introduction of an article published on Medium, about speeding up the DQN learning using PyTorch [8] and were well suited also for our specific case.

3.2 Neural network architecture

Since in the two Deep Mind's papers we have two different neural network architectures, we decided to implement the simpler of the two, in particular the one presented in paper "Playing Atari with Deep Reinforcement Learning" [4]:

1. An input layer with input shape $4 \times 84 \times 84$.
2. A convolutional layer having 16 filters with size 8×8 and stride 4.
3. A convolutional layer having 32 filters with size 4×4 and stride 2.
4. A fully connected layer with 256 hidden units.
5. A fully connected layer with *n* nodes, with *n* equals to the number of actions and linear activation as output of the network.

Note: both Q-network and target Q-network share the same architecture.

3.3 Loss function and Optimizer

3.3.1 Huber

The loss function used is Huber Loss: $tf.keras.losses.Huber()$ [5]

Huber is a loss function that mixes Mean Square Error (MSE) and Mean Absolute Error (MAE) loss functions. MSE loss functions makes the network care more about the large errors instead of the small ones due to the squaring operator. MAE, on the other hand, is more sensitive to small errors due to the absolute value operator.
Huber loss uses MSE for large values and MAE for small values and can be synthesized as:

$$Huber(a) = \begin{cases} \frac{1}{2}a^2 & if\ |a| <= 1 \\ (|a| - \frac{1}{2}) & otherwise \end{cases}$$

3.3.2 Adam

Adam is an optimizer designed specifically for training deep neural networks, showing huge performance gains in terms of training speed. As already said in section 3.1, this was the key point to pick Adam over RMSprop for our work.
Adam uses *Momentum* and *Adaptive Learning Rates* to converge faster:

- **Momentum**: "the momentum algorithm, accelerates stochastic gradient descent in the relevant direction, as well as dampening oscillations". [6]
- **Adaptive Learning Rate**: "Is the adjustments to the learning rate in the training phase by reducing the learning rate to a pre-defined schedule". [6]

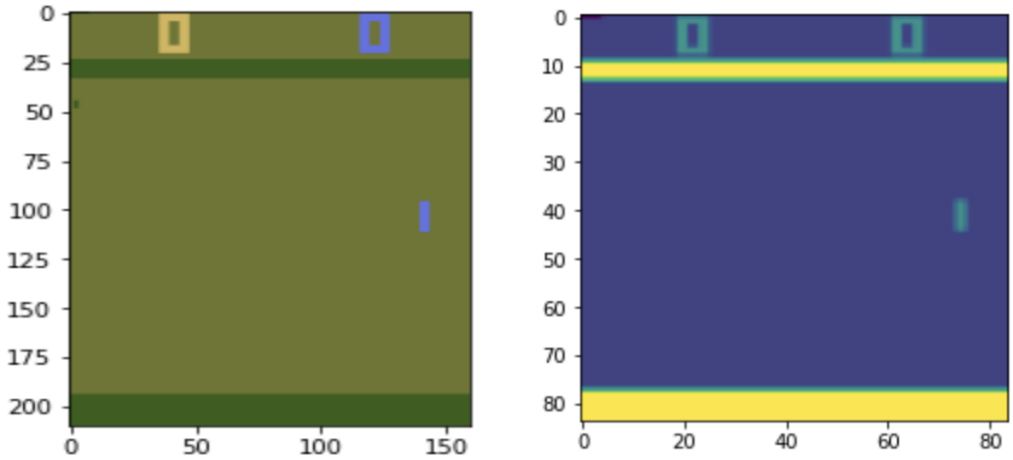
In practice Adam consists in an RMSprop optimizer that also has Momentum. [6]

3.4 Preprocessing and Environment details

The environment chosen from the Open Ai Gym's environments is *PongNoFrameskip-v4*. The environment has been altered with the help of two wrappers, *gym.wrappers.AtariPreprocessing()* and *gym.wrappers.FrameStack()* [7], with the goal of:

- Set the Frame skip to 4 to make the agent only experience the environment every 4 frames.
- Convert the image into grey scale.
- Resize the image to change it's original shape to 84×84.
- Create a frame stack of 4 frames.

The new shape of the observation after the preprocessing process, and the input to the neural networks, is $4 \times 84 \times 84$.



(a) Original observation

(b) Preprocessed observation

Figure 2: Preprocessing of the original game screen

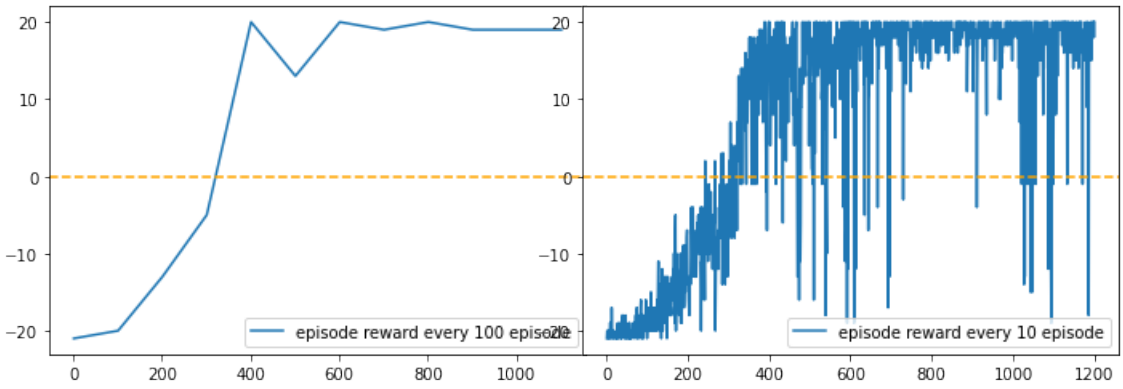
4 Conclusions

The training of the agent has been carried out on a Virtual Private Server (VPS) with 4 Ampere Altra CPU cores (based on the ARM Neoverse N1 architecture) and 24GB RAM on Oracle Cloud for 3 days and for a total number of 1200 episodes and then stopped manually.

The result of the training is very satisfying and see the agent reach a peak reward of 20 in an episode and also an average reward of 18.59 for last 100 episode as his best.

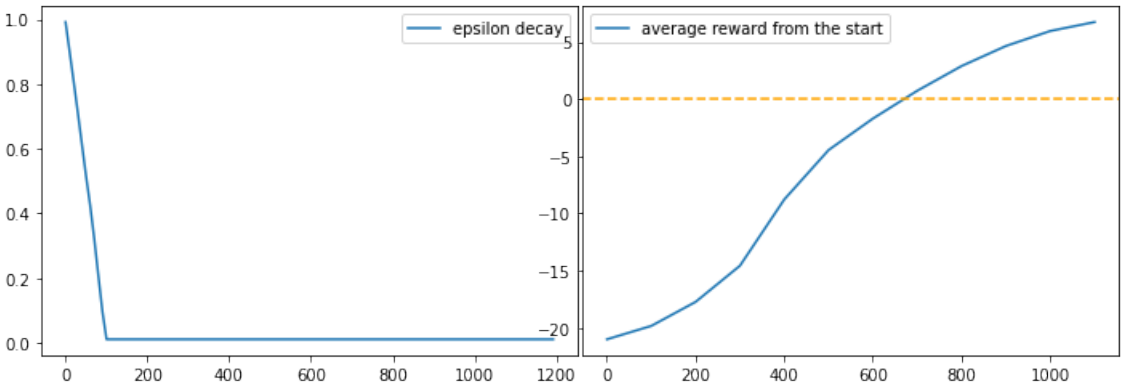
As shown in figures 3a and 3b, the agent was never been able to reach the maximum reward of 21 in an episode, not even while exploring. Analysing the videos recorded, we can see that the agent was consistently conceding the first point of the game, probably because the first ball served in a game was served in a different way compared to the ball served after the other player scored a goal. Exploiting this situation, made simpler for the agent to maximise the reward and eventually win the game, always repeating the same pattern of actions from that moment until the end of the episode. This strategy emerged in the later stages of the training. In earlier stages though, we can see situations in which the agent was still conceding the first point and still reaching the reward of 20 but playing in a different and more various way without repeating the same pattern of action over and over.

Further improvement to the work can be done by reconsidering the way rewards are assigned in order to help the agent understand how to reach the maximum reward in a game and also win the game.



(a) Reward per episode, every 100 episode

(b) Reward per episode



(c) Epsilon decay

(d) Average reward from start

References

[1] Alpha Go - https://en.wikipedia.org/wiki/AlphaGo

[2] OpenAI Gym - https://gym.openai.com

[3] DQN - Human level control through deep reinforcement learning - https://deepmind.com/research/publications/human-level-control-through-deep-reinforcement-learning

[4] DQN - Playing Atari with deep Reinforcement learning - https://arxiv.org/pdf/1312.5602.pdf

[5] Ronald Moore (May 2, 2019) Loss Functions for Neural Network https://en.wikipedia.org/wiki/Huber_loss

[6] Adam - https://towardsdatascience.com/adam-optimization-algorithm-1cdc9b12724a

[7] OpenAi Gym wrappers - https://github.com/openai/gym/tree/master/gym/wrappers

[8] Speeding up DQN on PyTorch - https://shmuma.medium.com/speeding-up-dqn-on-pytorch-solving-pong-in-30-minutes-81a1bd2dff55

[9] Reinforcement Learning Explained Visually - https://towardsdatascience.com/reinforcement-learning-explained-visually-part-5-deep-q-networks-step-by-step-5a5317197f4b