Comparison between DQN and Double DQN algorithms on Atari Breakout

Pierpasquale Colagrande
Alma Mater Studiorum - University of Bologna
`pierpasqu.colagrande@studio.unibo.it`

Abstract

The following project is a Python implementation of DQN and Double DQN algorithms and a comparison between the performances of the twos in the videogame Breakout for Atari 2600. The scope of this project is thus to compare the two algorithms showing the differences in performance and value estimates, along with the effects of the improvements that Double DQN introduced over DQN.

1 Introduction

1.1 Problem

The problem we are trying to solve is training an agent to make it play the videogame "Breakout" for Atari 2600 console. In particular, we will focus on the usage of the algorithms DQN and the improved version Double DQN. The scope is thus to compare the two algorithms and show what changes in terms of performances between the two algorithms when applied to a reinforcement learning task. At the same time, we are aiming to implement the two algorithms from scratch, exploiting the already available libraries just for the simulation environment.

The codebase will also be organized in a modular structure and by following an object-oriented approach, in order to produce source code as much readable and reusable as possible.

1.2 Environment

In order to test our implemented algorithms, we need a task to learn. In this case, the task we used was the game Breakout for Atari 2600. OpenAI's Gym library [1], which is a library that offers tools for reinforcement learning tasks, also offers a bunch of environment of Atari 2600 games, including Breakout. We're thus using Gym's Breakout environment.

The game consists in moving a bar left and right in order to hit a bouncing ball and destroy colored bricks. Depending on the color of the destroyed brick, we get a higher or a lower score.

1.2.1 Observation space

While using Gym's Atari environments, at each timestep of the training/testing procedure, we obtain an observation in the shape of a RGB image that is the frame of the game at that timestep. More particularly, the observation space has a shape 210x160x3, where 210 is the height of the frame, 160 is the width and 3 is the number of channels (the RGB channels). Each pixel of the three channels has an integer value from 0 to 255.

1.2.2 Action space

The action space, namely the space of all the possible actions, is a discrete space and depends on the game. Some games may have more available actions while other games have very few. The action

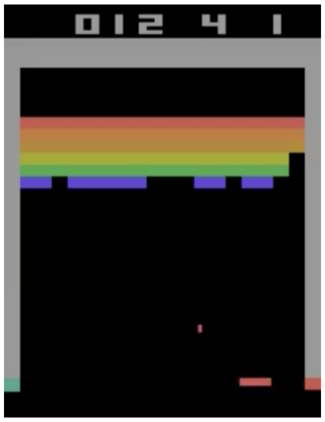

Figure 1: A frame of Breakout game for Atari 2600.

space of each game is a subset of a much larger space composed of 18 possible actions in total but, depending on the game, the possible actions may be less. Each of these actions is associated with an integer value that indicates that and only that action. In case of Atari Breakout, the possible actions are 4:

- NOOP: means "no action", namely do nothing, associated with integer 0
- FIRE: means "start the game by firing the ball", associated with integer 1
- RIGHT: means "move the bar right", associated with integer 2
- LEFT, means "move the bar left", associated with integer 3

1.3 Rewards

Intuitively, the rewards correspond to the score of the brick destroyed. More details regarding the environment are available in the dedicated page of the official Gym's documentation [2]. In this page we also find more information related to the rewards.

1.4 Algorithms

The algorithms implemented and tested are DQN and Double DQN.

Deep Q Network algorithm, DQN for short, is a deep reinforcement learning algorithm from Google DeepMind which comes from the combination of Q-learning and deep neural networks. More particularly, it is a Q-learning based algorithm in which the function approximator for values is a neural network in order to adapt the Q-learning algorithm to complex tasks like videogames which usually have a high dimensionality. DQN was introduced in two different versions by the same authors at Google DeepMind, a version by Minh et al. [3] from 2013 and a version by the same authors et al. [4] from 2015. The latter version is the same as the former but with some small changes, mostly in the training process and the neural network. We implemented DQN by following the 2015 version.

We also implemented an improvement of DQN called Double DQN. Double DQN was introduced by Van Hasselt et al. [5], researchers at Google DeepMind as an improvement to DQN. DQN computes value estimates as the maximum of the values estimated by the target network and this produces an overestimation bias in the value estimation. Max operator, in fact, tends to overestimate values under the noise of a single network. By using only one network for the estimation we are forcing a link between the network noise and the value estimates. Double DQN uses two networks to estimate values, a target network to get the action whose estimated value is the highest, (using argmax instead of max), and an online network to estimate values again by re-estimating the values for actions that were the ones with the highest estimates when using the target network. We basically use the online network to evaluate the estimates of the target network.

In order to implement the algorithms, we used PyTorch [6] library that provides all the tools to set up Deep Neural Networks and training/testing loops.

2 Implementation

We implemented DQN and Double DQN by entirely following the original papers. More specifically, we implemented the 2015 version of DQN [4] and the original version of Double DQN [5].

2.1 Environment improvements

We implemented different improvements to make training faster and reduce the dimensionality of the observation space. All these improvements were implemented as wrappers around the original Gym environment. Moreover, most of these wrappers were already available in Gym's Baselines library, we thus used these already available wrappers and changed a bit of their code to adapt them to some of our needs and fix some bugs related to libraries versions.

2.1.1 Preprocessing

The original DQN paper by Minh et al. [3] introduced a way to preprocess the observation frames in order to reduce the dimensionality of the input space. More particularly, they used:

- frame skipping: a certain number of frames is skipped at every timestep
  - max between frames: when skipping frames, the observation is built by taking the pixel-wise maximum between the two most recent frames among the skipped ones
- frame stacking: the network is given not just one observation frame but rather a stack of a certain number observation frames
- dimensionality reduction: to reduce the dimensionality of the observation frames, they are transformed into grayscale frames and rescaled to a lower size
- pixel value scale: the pixel values are scaled from range 0-255 to range 0-1

2.1.2 Reward clipping

All negative rewards are clipped to -1 while all positive rewards are clipped to +1, leading to rewards all being equal either to -1, 0 or 1.

2.1.3 Noop reset

The agent performs a certain number of noop actions when being reset, up to a certain maximum number of noop actions.

2.1.4 "FIRE" action at the beginning of an episode

In the game Breakout, the player has to press the "FIRE" button at the beginning of the game to make it start (to launch the ball), but this is something that the agent might find difficult to learn, so performing "FIRE" action automatically at the beginning of the training will speed up the training, even though a good agent should be able to learn this behaviour by itself.

2.1.5 Time limit

To avoid the agent being stuck, especially at the beginning, we also limited the maximum number of frames per agent. If the agent reaches this frame limit, the current episode is ended and a new episode is started. This avoids the agent being stuck but also speeds up the training because the agent has also to learn to play and win faster.

2.1.6 Episode as a life

At the beginning of the game, we have 5 lives. Everytime the player misses the ball, we lose one life and when we get to 0 the game is over. We implemented a wrapper around the environment such that we consider each life as a single episode (a single game) in order to speed up the training, so when we lose one life, the episode ends and a new episode starts with one life less. When we get to 0 lives, the next episode starts from 5 lives again.

2.2 Training

2.2.1 Experience replay

Our agent, following DQN paper, uses experience replay, namely we use a buffer to hold state transitions and we sample a minibatch of these state transitions to feed to the network at each training step. At the beginning of the training, this buffer is initialized with a certain number of samples that are collected by only selecting the action to perform randomly.

2.2.2 ϵ-greedy training policy with ϵ decay

We used the ϵ -greedy training policy, namely at each training step the action to perform is selected based on a probability ϵ of it being a random action. Consequently, at each training step, the agent will follow a random policy with a probability of ϵ while it will follow the learnt policy with a probability of $1 - \epsilon$. This ϵ value is decayed based on the number of training steps, namely the higher the current training step, the lower the ϵ value is. ϵ is decayed from a max value to a min value over a certain number of training step, after this number of steps it is kept to the minimum value.

2.2.3 Separated training and testing environments

We used two separate training and testing environments. The testing environment differs from the training environment only in its initialization. In fact, each time the testing environment is reset, the same seed is used for the reset in order to have exactly the same environment when testing, thus having comparable testing results between various training steps. In fact, we want to have different environments when training while the same environment when testing.

2.2.4 Vectorized training environment

The training environment was also vectorized, so instead of having one environment and performing just one environment step for each training step, we have a stack of a certain number of environments, ending up in multiple environment step for each training step. This speeds up the collection of new state transitions.

2.2.5 Training/testing configurations

To configure the training/testing phase, we used .yaml files in combination with Hydra [7] library in order to create configuration files that are automatically loaded at the beginning of the training/testing. These configuration files are used to define the hyperparameters for the training experiment and other parameters like, for example, the seeds for reproducibility or the names of the various output files.

2.3 Model

For both DQN and Double DQN algorithms, we used two networks, an (online) q network that we used to select the action at each training step, which is also the network on whose parameters we perform gradient descent step, and a target (offline) q network, which is the network that we use to estimate values. This network is exactly the same as the q network, the only difference is that its parameters are updated every C steps by copying the parameters from the online q network. We implemented the network described in the 2015 version of DQN [4], which is composed of:

- a convolutional layer with 32 filters of size 8x8 and stride 4, followed by a ReLU activation function
- a second convolutional layer with 64 filters of size 4xr4 and stride 2, again followed by a ReLU layer
- a third convolutional layer with 64 filters of size 3x3 and stride 1, followed again by a ReLU layer

- a linear layer consisting of 512 dense units, followed again by a ReLU layer
- a final output layer consisting of as many dense units as the game possible actions, in case of Breakout these are 4

2.4 Algorithms

We implemented the two algorithms using an object-oriented approach. Since we were using PyTorch to implement the deep network, we used PyTorch tensors and operations between tensors to represent the observation frames and the various batches of state transitions, in order to make it easier to pass this data to the network and make everything more readable and consistent across the codebase. We used Numpy arrays only for support in certain situations.

3 Results

3.1 Experimental setup

We did two experiments, the first consisting in training an agent for Atari Breakout with DQN and the second consisting in training the same agent with Double DQN. In order to compare the two experiments, we kept all the hyperparameters the same between the two experiments, only changing the algorithm. The training, preprocessing and additional hyperparameters we used are available in the Appendix.

We also used Wandb [8] to log each metric at each training step.

3.1.1 Reproducibility

To make the results even more comparable, we set all random seeds the same across the two experiments, so we ended up setting the seed for Numpy, PyTorch and Python random operations. Moreover, at the beginning of the training, we set once the training environment seed and, since the training agent is a vectorized agent, we set a different seed for each environment in the stack. As said in the previous section, we also set the seed for the testing environment, but differently from the training environment, we set the testing environment seed each time we reset the environment, in order to have exactly the same testing environment at each test step, while we don't want this behaviour with training. Moreover, the seed for testing was different from the seed for training, to have separate environments. The training environment was still changing at each reset, because we set up the seed once, but the way it changed was depending on the seed set at the beginning of the training, so the training environment changed the same way across all the experiments.

3.2 Quantitative results

We used different metrics to evaluate the agents. While training, we also evaluated the agent after a certain number of training steps, by doing an on-policy testing episode.

3.2.1 Value estimates

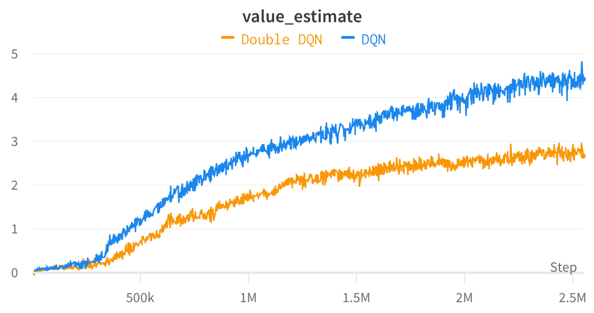

Figure 2: Comparison of value estimates per training step between Double DQN and DQN.

The value estimates are used to compare not the two agents performances but to effectively see how much DQN overestimates values compared to Double DQN. In fact, while training, Double DQN should keep the value estimates on a lower magnitude compared to DQN. Since this is purely a training metric, there isn't a testing version of this metric.

As we can see from Fig. 2, Double DQN keeps the value estimates on a lower magnitude compared to DQN, indicating that DQN is overestimating the value estimates, as expected. Moreover, we can see how, over time, DQN value estimates tend to grow faster compared to Double DQN value estimates, indicating again that Double DQN handles overestimation better than DQN.

3.2.2 Average reward

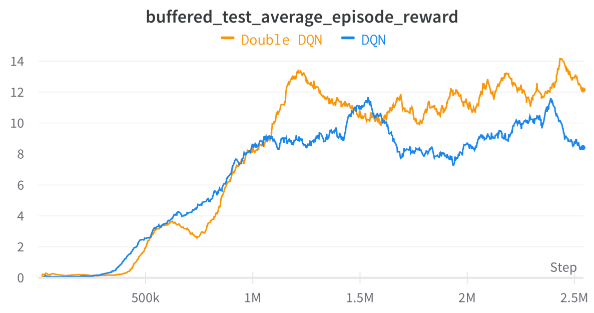

Figure 3: Comparison of testing average episode reward per training step between Double DQN and DQN.

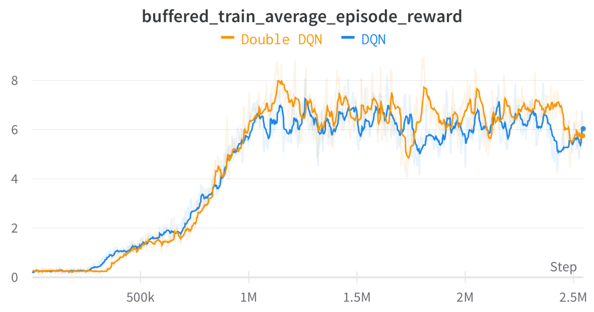

Figure 4: Comparison of training average episode reward per training step between Double DQN and DQN.

The average reward consists in averaging the total reward of each episode. In doing this, we decided to take two average rewards metrics, one for training and one for testing. We did this because while training, the agent selects an action to perform on the training environment accordingly to the ϵ-greedy policy, so it will not act by fully following the learnt policy because, while training, we want to also keep some exploration even after a certain number of training step. On the other hand, when testing, the agent selects actions only following the learnt policy, so the average reward for testing is entirely obtained by following the learnt policy and this gives a good measure of how well the agent is doing and how much it is improving over time. Regardless this difference, both metrics should improve over time.

We computed the average reward as a buffered average reward, namely an average reward computed over a certain number of most recent training/testing episodes. This gives us an indication of how much the agent has improved recently, because we limit the computation of the average reward to some recent episodes.

As we can see from Fig. 3, Double DQN achieves a higher testing average reward in the long term compared to DQN, even though this starts to happen only after 1 million training steps. This confirms the hypothesis that DQN's overestimation also influences the performances of the agent in the long term. In fact, as we can see, before 1 million steps the performances between the two agents are more or less the same.

Overestimation also influences the training average reward. In fact, as we can see from Fig. 4, Double DQN training average reward stays on average above DQN training average reward, especially after 1 million training steps. This is less visible, however, because we still have some random action selection happening in the training even after 1 million frames because of ϵ-greedy policy, so the graph is more noisy and volatile compared to the testing graph. In fact, we also had to apply a little bit of smoothing to the graph to make the results more comprehensible. The original graph is still visible in the same figure and is the one more transparent. However, it is still possible to see that both in training and testing, Double DQN achieves a higher average reward peak.

3.2.3 Episode total reward

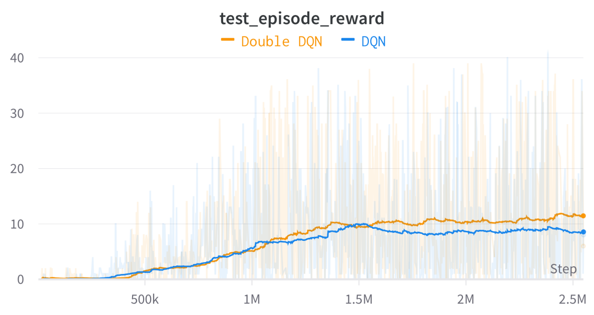

Figure 5: Comparison of testing episode reward per training step between Double DQN and DQN.

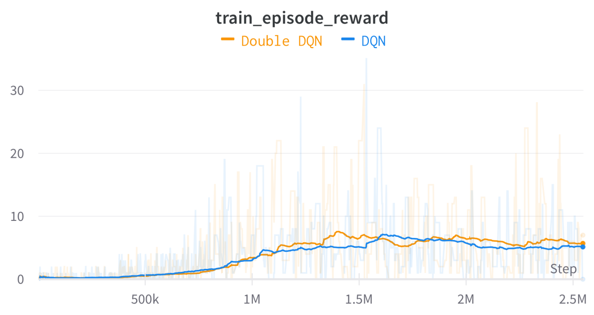

Figure 6: Comparison of training episode reward per training step between Double DQN and DQN.

We also considered the total reward of the last episode as a metric of evaluation, because ideally the more the agent learns, the higher reward it will obtain in each episode. We also divided between training and testing episode total reward, resulting in two different total episode reward metrics. Ideally, both of these metrics should improve over time.

As we can see from Fig. 5, the total reward per episode is higher when training the model with Double DQN compared to a model trained with DQN, especially after 1 million steps. Since during training each life is considered as an episode, the original graph was very noisy and was nearly impossible to determine the actual improvement of the two agents, so we had to apply a lot of smoothing to the graph to actually see and evaluate the performances of the two agents. As we can see, however, this metric grows significantly slower compared to average reward in both cases and this is due to the fact that the agent needs to be trained longer in order to achieve a higher performance per episode. Double DQN is still performing slightly better compared to DQN, especially after 1 million steps.

Regarding training episode reward, instead, we can see from Fig. 6 that even with smoothing, it is nearly impossible to determine which one of the two agents is performing better, and this is again because of the presence of a random component in the training that makes everything more noisy.

Our conclusion is that more training is needed to clearly see a difference between the two agents in this metric, also without smoothing. In fact, with smoothing, this metric basically collapses to average reward metric because of the averaging nature of the smoothing operation, so we can observe a behaviour similar to the one of the average reward metric.

What is important, however, is the fact that if we look at the original non-smoothed graph, both metrics are reaching higher values over time, especially compared to the beginning of the training, indicating the fact that both algorithms are actually learning to play and maximize the score in each episode, and it seems that Double DQN reaches higher scores in the various episodes on average compared to DQN, however as said before, this metric alone is not enough to compare the two agents, it it just another way of checking if the agents are actually learning to play.

3.3 Qualitative results

After a certain number of testing episodes, we also recorded and logged the testing episode, in order to visually see how much the agent is improving in playing. It is possible to see these videos and the comparison between the two agents during the training in the Wandb page [9] of this project. Moreover, it is possible to see a comparison of the two agents at the end of training in the folder "test_videos" of the GitHub repository [10] of this project.

4 Future developments

In order to improve the results, hyperparameter tuning could be deployed to select better hyperparameters. However, improving the scores of the agents was not the scope of this project, so we left the default hyperparameters in order to compare the two agents as fairly as possible.

Another improvement could be to add some deep learning regularization techniques, like dropout layers in the deep network or gradient clipping. These techniques will also allow to train longer with a reduced risk of overfitting. Another improvement could be to deploy a learning rate scheduler.

Moreover, a simple but effective improvement could be to train these agents using parallel vector environments, to speed up the collection of samples even more, and to train on better hardware to train faster and longer and get even a better comparison between the two agents. In fact, as it is possible to see in Double DQN paper [5], noticeable differences can be seen between Double DQN and DQN both in average reward and value estimates around and after 50 million training steps.

4.1 Conclusion

As it is possible to see, both algorithms are improving and getting better the longer we train them, but we can also see how Double DQN keeps the value estimates to a lower magnitude compared to DQN and tends to reach higher scores. Double DQN paper [5] claims that the overestimation bias becomes more visible the more actions we have in the action space, so in their paper it is possible to see a clear difference in value estimates and performances in games with a higher number of possible actions. Moreover, they also show that the longer the agents are trained, the more the overestimation bias increases and the more the differences between the two algorithms become clear. In our experiments, we didn't train for too long because of the lack of hardware to do that and we didn't chose a more complex game but we still had a good measure of the differences and improvements that Double DQN introduced over DQN.

References

[1] Openai gym library. URL: `https://www.gymlibrary.ml`.

[2] Atari breakout environment from openai gym. URL: `https://www.gymlibrary.ml/environments/atari/breakout/`.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, 2013. DOI: `10.48550/arxiv.1312.5602`. URL: `https://doi.org/10.48550/arxiv.1312.5602`.

[4] 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*, 518(7540):529–533, February 2015. ISSN: 1476-4687. DOI: `10.1038/nature14236`. URL: `https://doi.org/10.1038/nature14236`.

[5] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning, 2015. DOI: `10.48550/ARXIV.1509.06461`. URL: `https://doi.org/10.48550/arxiv.1509.06461`.

[6] Pytorch library. URL: `https://pytorch.org`.

[7] Hydra. URL: `https://hydra.cc`.

[8] Wandb. URL: `https://wandb.ai/`.

[9] Wandb project page. URL: `https://wandb.ai/pierclgr/AAS_project`.

[10] Github repository. URL: `https://github.com/pierclgr/Atari-Deep-RL/`.

Appendix

To train the two agents, we used Google Colab. Colab limits the available RAM capacity, the GPU and, most importantly, the notebook execution time. We were able to train the networks only for 24 hours, which correspond roughly to 2.5 million training steps which, because of frame stacking (at each training step, we stacked 4 frames), correspond to roughly 10 million frames . If we also consider frame skipping (4 frames at each training step), the number of total frames is roughly 40 millions, but only 10 millions of these were processed, because of frame skipping. If we also think that while frameskipping, we build an observation as the pixel-wise maximum between the two most recent skipped frames, this means that in reality the frames that were used for the observation building and training procedure were 20 millions.

We used the same hyperparameters in the two experiments and these were the hyperparameters that were listed in both DQN paper from 2015 [4] and Double DQN paper [5]. The only thing we changed, because of Colab's limits, is the capacity of the replay buffer that was reduced from 1M to 100K samples, and the number of total training steps, that was reduced from 50M to roughly 2.5M.

Moreover, we didn't enable the automatic fire selection at reset because a good agent should be able to learn to press "FIRE" at the beginning of the game to start playing. Both of our agents were in fact able to learn this behaviour by themselves. Table 1 contains all the used hyperparameters.

Environments hyperparameters	
Number of vectorized training environments	4
Max steps per episode	1000
Num. frames to skip	4
Num. frames to stack	4
Resized observation size	84
Grayscale	True
Num. max noop	30
Scale observations	True
Life as an episode	True
Reward clipping	True
Fire action at reset	False
Experience replay hyperparameters	
Buffer capacity	100K
Num. initial samples in the buffer	50K
Training hyperparameters	
Num. of training steps	2.7M
Batch size	32
Max epsilon value	1
Min epsilon value	0.1
Decay epsilon every	1M steps
Reward discount factor	0.99
Update target network every (C)	10K steps
Learning rate	0.00025
Optimizer	RMSprop
Momentum	0.95
Squared momentum	0.95
Min. squared gradient	0.01
Loss function	Smooth L1 loss
Testing hyperparameters	
Episode reward buffer capacity	100
Test model every	2K steps
Save testing episode video every	5 testing episodes
Reproducibility seeds	
Random seeds	1507
Training environments seeds	1507, 1507+1, ..., 1507+num_vector_environments
Testing environment seed	2307

Table 1: Hyperparameters used for the training.