MALS Project: Bootstrapped DQN

Jens Nevens
500093
jens.nevens@vub.be

Mathieu Reymond
502002
mathieu.reymond@vub.be

I. Introduction

In this project, we extend the immensely popular Deep Q-Learning (DQN) algorithm. Since its introduction, there have been countless variations and extensions to the basic principles it introduced to use a deep neural network in a reinforcement learning setting, using raw pixel values as inputs. Many of these have been evaluated using Atari games. We have chosen to implement Bootstrapped DQN (BDQN), a technique aimed at improving the exploration/exploitation trade-off.

In this report, we briefly discuss the ideas behind DQN and BDQN. Afterwards, we elaborate on the changes we needed to make to implement BDQN. Next, we compare the results of our BDQN implementation to the standard DQN approach in two Atari game environments and we discuss our findings. Finally, next to the technical aspect, we will also discuss our practical findings, since running these algorithms is not always an easy task.

II. Deep Q-learning

Consider an agent, situated in an environment, that can perform a set of actions in order to interact with it. This agent needs to reach a certain objective, and thus will try to learn a mapping of state to actions as to optimise this objective. This learning process is guided by means of rewards that the agent receives, typically when the goal is reached.

In Q-learning, the agent will assess which action to take by means of Q-values, a measure that predicts the positive impact of this action to the final objective. More concretely, $q_\pi(s,a)$ is defined as the expected discounted reward of taking action a in state s, and thereafter following policy π:

$$q_\pi(s,a) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}\,\middle|\, s_t = s, a_t = a\right]$$

where γ is the *discount factor*.

Optimising this function will lead to the highest possible cumulated reward for any given state:

$$Q^*(s,a) = \max_\pi q(s,a)$$

Deep Q-learning (DQN) uses a deep convolutional neural network to approximate this function. In order to cope with the unstability and divergance produced by a non-linear approximator to represent the Q-value function, DQN introduces two ideas.

First, it uses *experience replay*. Instead of directly training on the learned experience (s_t, a_t, r_t, s_{t+1}), each tuple is saved into memory. The learning is then applied on a set of samples taken uniformly at random from the memory. The main benefit of this technique is the removal of correlation between the experience sequences.

Secondly, the correlation between the current action-values and the target values is reduced by only periodically updating the target values. Thus, the weights θ at step i of the neural network are updated according to the following loss function:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right]$$

where U is a uniform random distribution, D the experience replay memory, θ^- the target weight (only updated after a fixed number of steps).

These two ideas allowed DQN to achieve super-human performance in a range of challenging domains using only the raw pixel-values as input. Since the introduction of the DQN algorithm, there has been a tremendous amount of variations and extensions using the same ideas as presented above. One of these extensions is Bootstrapped DQN.

III. Bootstrapped DQN

When using Reinforcement Learning techniques, one key idea to keep in mind is the balance between exploration and exploitation. Without the former, one may easily find itself stuck into a poor local optimum, without any means to overcome it. However, depend too much on it, and your agent will keep wandering off, never to follow its optimal path. Nonetheless, even with a balanced strategy, exploration often takes place sporadically.

Bootstrapped DQN (BDQN) aims to tackle this shortcoming by encouraging *deep exploration* on a DQN agent. Instead of exploring over a limited (single) amount of steps, the agent will avoid the optimal path over a longer duration of time, increasing the quality of its search.

The algorithm applies bootstrapping, a technique that, given a dataset D, will generate k datasets $\tilde{D}$ of the same size by uniformly sampling D with replacement. To implement this efficiently, a mask $m_i, i \in \{1,\dots,k\}$ will be added to each recorded experience, that states in which distribution it belongs.

Using a modified architecture of the deep convolutional network in order to accomodate for those datasets, BDQN adds k heads that replace the original DQN output layer. Each head is trained only on its own sampling distribution $\widetilde{D}$. At the start of each episode, the agent selects one head, from which it will record its experience. The followed policy is the one from that head, until the end of the episode.

This followed policy π_k may be optimal for the current head, but that is not necessarily the case for the other ones. Those heads will, for the larger part of the episode, follow an exploration strategy.

The Q-update is applied to all heads, each one drawing experience from the experience replay memory, according to its mask. It is important to mention that each Q-function $Q_k(s, a; \theta)$ is trained against its own target functino $Q_k(s, a; \theta^-)$.

Algorithm 1 The DQN algorithm

```
Input: Value function network $Q$.
1: Let $B$ be a replay buffer storing experience for training
2: for each episode do
3:     Obtain initial state from environment $s_0$
4:     for step $t = 1$ to end of episode do
5:         Pick an action according to $a_t \in \text{argmax}_a Q(s_t, a)$
6:         Receive state $s_{t+1}$ and reward $r_t$ from environment,
           having taken action $a_t$
7:         Add $(s_t, a_t, r_t, s_{t+1})$ to replay buffer B
8:     end for
9: end for
```

Algorithm 2 The Bootstrapped DQN algorithm

```
Input: Value function networks $Q$ with $K$ outputs $\{Q_k\}_{k=1}^{K}$.
    Masking distribution $M$.
1: Let $B$ be a replay buffer storing experience for training
2: for each episode do
3:     Obtain initial state from environment $s_0$
4:     Pick a value-function to act using $k \sim U(1, \ldots, k)$
5:     for step $t = 1$ to end of episode do
6:         Pick an action according to $a_t \in \text{argmax}_a Q_k(s_t, a)$
7:         Receive state $s_{t+1}$ and reward $r_t$ from environment,
           having taken action $a_t$
8:         Sample bootstrap mask $m_t \sim M$
9:         Add $(s_t, a_t, r_t, s_{t+1}, m_t)$ to replay buffer B
10:    end for
11: end for
```

IV. Experimental Setup

Our setup is based on an off-the-shelf implementation of Deep Q-Learning[1]. Before starting the development of our extension of DQN to Bootstrapped DQN, we experimented with different parameter settings. An overview of the parameter values used in these experiments can be seen in Table I.

First of all, we reduced the replay memory size from 1000000 samples to 250000 samples. The main reason for this was to reduce the runtime overhead and memory requirement, while

[1]https://github.com/devsisters/DQN-tensorflow

Parameter	Value
Replay Memory Size	$250,000$
Epsilon	$1 \xrightarrow[1,000,000]{} 0.1$
Learning Rate	$0.0025 \xrightarrow[50000]{0.96} 0.00025$
Screen Width/Height	42
History Length	8

TABLE I
DQN Parameter Tuning

still maintaining enough samples to learn from. We saw both in Pong and Space Invaders that a smaller replay memory was still sufficient for the algorithm to learn.

We also experimented with ϵ, controlling the exploration/exploitation trade-off during training. In standard DQN, this parameter is annealed from 1 to 0.01 over the first 10^6 interactions. In our experiment, we set the lowest value for ϵ to 0.1 to increase the exploration of the agent. Our goal was to slightly increase the exploration of the agent such that it would find an optimal solution faster. However, we saw no significant changes in learning speed or success and opted for the standard annealing scheme from 1 to 0.01.

A similar experiment was ran for the learning rate α. In standard DQN, this is fixed at 0.00025. We experimented with an annealing scheme of 0.0025 to 0.00025 over the first 50000 interactions, with the learning rate decay set to 0.96. While the reward of the agent appeared to go up much faster using this annealing scheme, this was only true at the start of the experiment. In the longer run, there was no difference in reward. Consequently, the standard learning rate of 0.00025 was kept.

In another attempt to reduce the memory requirement of the resource-intensive DQN algorithm, we reduce the width and height of the input pixels from 84×84 to 42×42. However, an often occurring issue when downsizing the input images is too much of the image details are lost. We saw this effect in Pong, where the ball was probably no longer detected. Again, the standard parameters were kept.

As a final experiment, we doubled the history length from 4 to 8. The history length determines the number of input frames that will be considered by the agent as a single learning step, since it is impossible to detect features of the input (e.g. movement) using a single frame. While we did see an increase in the reward obtained by the agent (in Pong), the larger history length also caused a significant increase in runtime – almost doubling runtime. We could not afford this increase in runtime and opted for the standard history length of 4 frames.

The parameter values discussed above are also used in the BDQN extension. The other parameters remained unchanged from the off-the-shelf DQN implementation.

As mentioned earlier, BDQN makes use of a *mask* to determine which heads are trained on what samples from the replay memory. There are several possibly ways of creating these masks. We opted for a binary mask of the same length as the number of heads, generated using a binomial distribution $B(1, 0.5)$. In the original BDQN implementation, this is

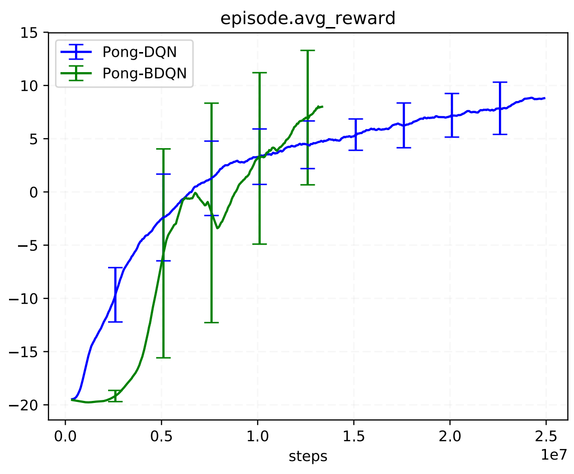

Fig. 1. Average reward per episode for Pong

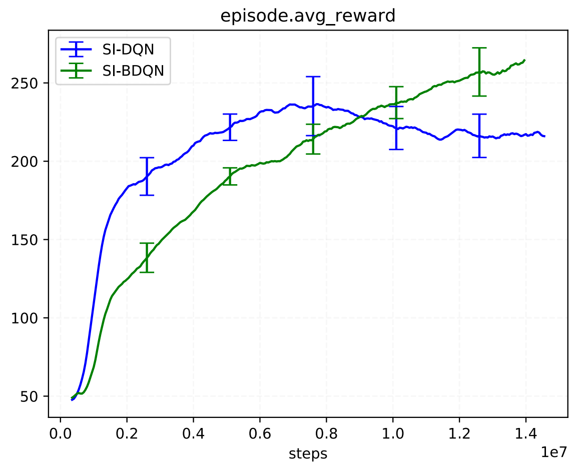

Fig. 2. Average reward per episode for Space Invaders

referred to as the double-or-nothing bootstrap. This mask is however only used during training. When evaluating, we use an ensemble voting policy. In other words, each head votes for a given action and the action with the most votes is executed.

Finally, we should also note that the original BDQN implementation makes use of the DDQN update rule and that the weights in the heads are normalized by a factor $1/K$. These are aspects of the original work that we did not include in our extension.

V. Experimental Results

We chose to evaluate our implementation on 2 Atari game environments: Pong and Space Invaders. We chose Pong because it is often quite easy to train and many DQN variants achieve the same reward on it. This can serve as a benchmark. Space Invaders, on the other hand, is more challenging. However, random behaviour might lead to good scores.

For each algorithm, we performed 10 runs and plotted the average reward per episode. Each line represents the average of those 10 runs, using a running average of 5×10^5. The results are shown for Pong and Space Invaders in Figure 1 and Figure 2, respectively.

As we can see from those plots, DQN learns faster from the start than BDQN. This is due to the multiple heads BDQN is using. Even though the Q-update is applied to all of them, the vast majority is not only following a policy that is sub-optimal for them, but all heads are also trained on a smaller experience replay dataset. Indeed, the number of experience samples received is the same as with DQN, but all those samples now have a mask, meaning they will not be used by all the heads.

Eventually, despite its slower start, BDQN performs better than DQN. For both games, the BDQN variant achieves a higher reward than DQN after 1×10^6 steps. The deep exploration enforced by following another head's policy thus seems to pay off. This result is, however, more clear for Space Invaders than it is for Pong. Due to the large variance in Pong, we again would need more training to completely validate this hypothesis.

When looking specifically at the performance of Space Invaders, we notice a dip in performance for DQN after 8×10^6 steps. We try to explain this decline by analysing the average Q-values over time (as shown in Figure 3). DQN's Q-values stabilise a bit after 2×10^6 steps but, right when the reward decline is about to start, we see the Q-values increasing again. We think that the diverging of those values lead to the decrease in performance.

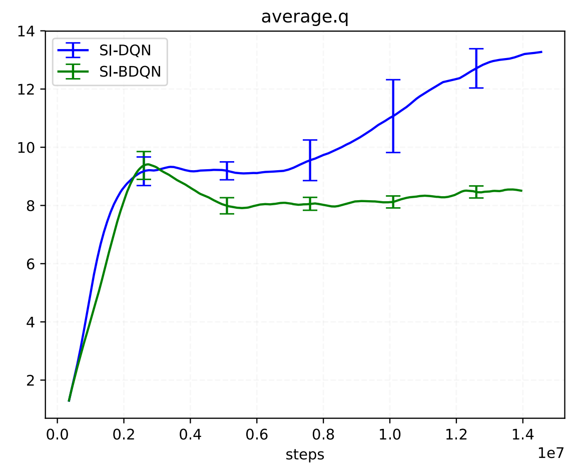

Fig. 3. The average Q values for Space Invaders

We can note that the BDQN algorithm has a very large variance for Pong (Figure 1) compared to DQN. For Space Invaders (Figure 2) there is no such difference in variance. We argue that this is due to the nature of the two games used for evaluation. In Space Invaders, the agent can reach decent scores by performing random actions, randomly shooting for example. In Pong, this will not work. The agent has to stay alive by keeping the ball in the game. Thus, Pong is more strict when it comes to finding a good policy. When one of the 10 runs has some issues finding a good policy, it will perform bad. On the other hand, when a run has found a decent policy, it performs better. We can also note that the variance of BDQN Pong decreases as the number of steps surpasses 1×10^6. However, we would need more training to further validate this hypothesis.

In general, we were unable to perform all 50×10^6 training steps due to time/CPU power constraints. We should note that multiple runs of Pong DQN suddenly jumped from an average reward of 5 to 15 after 30×10^6 steps. This is not entirely clear on Figure 1 due to the averaging and sliding window. An average score of 15 is what multiple DQN variants in the literature consistently obtain. For Space Invaders, we could only complete approximately 15×10^6 training steps. Perhaps the DQN variant would have picked up again after more training.

VI. Conclusion

In the games we used to evaluate our DQN extension, we found that BDQN does indeed improve upon standard DQN. As was noted by the original authors, BDQN is slower to learn. This is due to the larger network architecture and the reduced experience for each head. However, the 20% decrease in learning time mentioned by the original authors is not observed in our experiments. This can be dependant on many factors, e.g. choice of game environment, hardware specifications, etc.

Furthermore, as we have noted in Section IV, the original BDQN implementation is further optimized by using the DDQN update rule and gradient normalization. We doubt these changes would make any difference to the time required to learn, however, they could give a small boost to learning performance.

Appendix

The source code of this project is available on GitHub: https://github.com/JensNevens/AtariDominator. The DQN implementation is located on the *master* branch. The BDQN implementation is present on the *bdqn* branch.