Deep Q-Learning

Matteo Fordiani[1,*]

[1]University of Turin, Physics of Complex Systems MSc
[*]matteo.fordiani@edu.unito.it

ABSTRACT

This report accompanies a working implementation of Deep Q-Learning and Double Q-Learning as described in "Human-level control through deep reinforcement learning" by Mnih et al.[1], and "Deep reinforcement learning with double q-learning" by Hessel et al.[2] respectively. We propose an agent built via the PyTorch framework that replicates the Deep-Q and Double-Q architectures on the Atari 2600 Breakout game. The player agent replicates the results described in the papers, when accounting for the difference in computational means.

The accompanying code can be found on GitHub at matteo4diani/deep-q-learning for the Google Colaboratory notebook and at matteo4diani/deep-q-learning-utils for the model as a Python script and various utilities.

Figure 1. Our agent playing Breakout

Background

Reinforcement learning has its roots in control theory and computer games offer a perfect benchmark for an agent's ability to control its environment. Video-games have the added benefit of a readily available human benchmark. The Atari 2600 environment is an established benchmark for RL agents.
The Atari 2600 provides a 160x192 pixels display and 8-bit graphics: due to these lo-tech specs it's easy to replicate RL achievements on a modern personal computer.
The first use of the Atari 2600 as a benchmark was in 2013 by Bellemare et al.[3]: in 2015 Deep Q-Learning by DeepMind is the first deep learning model to successfully learn control policies for Atari games directly from high-dimensional sensory input using reinforcement learning[1].
The DeepMind model is a convolutional neural network with layers of tiled convolutional filters to mimic the effects of receptive fields. The network is trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards given the current observations.
The distinguishing principle of Q-Learning can be identified in the emphasis on *future rewards*: this idea of "thinking in advance" was introduced by Watkins in his doctoral thesis of 1989[4].

The main question in this line of research could be expressed as: can an agent entirely reliant on sensory perceptions and on a single numerical reward learn to outperform humans on complex tasks in dynamic environments by thinking ahead of itself and replaying its own experience?

Methods

Theory

Let an agent Λ in a possibly stochastic environment ε. Λ can interact with ε at discrete time-steps t and after every interaction Λ receives a reward r_t (which may be negative) and ε may change its state accordingly. Λ has a finite set of interactions $A = \{1, ..., K\}$.
At each time-step Λ observes ε only as an image $x_t \in \mathbb{R}^d$ and chooses $a_t \in A$. We therefore consider sequences of actions and observations, $s_t = x_1, a_1, x_2, ..., a_{t-1}, x_t$, and learn game strategies that depend upon these sequences. The goal of the agent is to interact with the emulator by selecting actions in a way that maximises future rewards. We make the standard assumption that future rewards are discounted by a factor of γ per time-step, and define the future discounted return at time t as $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$ where T is the time-step at which the game terminates.
We define the optimal action-value function $Q^*(s,a)$ as the maximum expected return achievable by following any strategy, after seeing some sequence s and then taking some action a as $Q^*(s,a) = \max_\pi \mathbb{E}[R_t \mid s_t = s, a_t = a, \pi]$ where π is a policy mapping sequences to actions (or distributions over actions).
The optimal action-value function obeys an important identity known as the *Bellman equation*. This is based on the following intuition: if the optimal value $Q^*(s',a')$ of the sequence s' at the next time-step was known for all possible actions a', then the optimal strategy is to select the action a' maximising the expected value of $r + \gamma Q^*(s',a')$, thus $Q^*(s,a) = \mathbb{E}_{s' \sim \mathscr{E}}[r + \gamma \max_{a'} Q^*(s',a') \mid s,a]$. The basic idea behind many reinforcement learning algorithms is to estimate the action-value function, by using the Bellman equation as an iterative update, $Q^*_{i+1}(s,a) = \mathbb{E}[r + \gamma \max_{a'} Q^*_i(s',a') \mid s,a]$.

$$Q^{\text{new}}(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t,a_t)}_{\text{current value}} \Big)}^{\text{temporal difference}} \tag{1}$$

Such value iteration algorithms converge to the optimal action value function, $Q_i \to Q^*$ as $i \to \infty$. In practice, this basic approach is totally impractical, because the action-value function is estimated separately for each sequence, without any generalisation. Instead, it is common to use a function approximator to estimate the action-value function $Q(s,a,\theta) \approx Q^*(s,a)$. The DeepMind model uses a neural network as $Q(s,a,\theta)$. We refer to a neural network function approximator with weights θ as a Q-network. A Q-network can be trained by minimising a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i

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

where $y_i = \mathbb{E}_{s' \sim \mathscr{E}}[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) \mid s,a]$ is the target for iteration i and $\rho(s,a)$ is a probability distribution over sequences s and actions a that we refer to as the behaviour distribution. Differentiating the loss function with respect to the weights we arrive at the following gradient,

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot); s' \sim \mathscr{E}}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right) \nabla_{\theta_i} Q(s,a;\theta_i)\right] \tag{2}$$

Rather than computing the full expectations in the above gradient, it is often computationally expedient to optimise the loss function by stochastic gradient descent. If the weights are updated after every time-step, and the expectations are replaced by single samples from the behaviour distribution ρ and the emulator ε respectively, then we arrive at the familiar *Q-learning* algorithm.
Note that this algorithm is model-free: it solves the reinforcement learning task directly using samples from the emulator ε, without explicitly constructing an estimate of ε. It is also off-policy: it learns about the greedy strategy $max_a Q(s,a;\theta)$, while following a behaviour distribution that ensures adequate exploration of the state space. In practice, the behaviour distribution is often selected by an ε-greedy strategy that follows the greedy strategy with probability $1-\varepsilon$ and selects a random action with probability ε.

Implementation

Preprocessing

1. Convert RGB to Grayscale
2. Rescale images to $84 * 84$
3. Stack $m = 4$ most recent frames to supply as input
4. Clip all positive and negative rewards between 1 and -1
5. Frameskip every $k = 4$ frames and select last action for skipped frames
6. Set a hard maximum (30) on no-op (do nothing) actions at the start of each episode

Training

We implement the Deep Q-Learning algorithm as:

```
» Initialize replay memory $\mathscr{D}$ to capacity $N$
» Initialize action-value function $Q$ with random weights
» for episode $= 1, M$ **do**
»» Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
»» for $t = 1, T$ **do**
»»» With probability $\varepsilon$ select a random action $a_t$ otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
»»» Execute action $a_t$ in 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 $\mathscr{D}$
»»» Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathscr{D}$
»»» Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
»»» Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 2
»» end for
» end for
```

Results

Training was conducted first in the Google Colaboratory managed environment (Python 3.7 at time of writing) for exploratory analysis, then on a virtual machine with 64GB RAM and running Ubuntu Linux 22.04 LTS (Python 3.10) for the final model training. We deem the experiment a success: the agent learns to play with human-like proficiency. The agent is not yet achieving the high-scores reported by DeepMind, but the average reward per life reported in the charts can be misleading: in its best runs the agent achieves rewards around 300 points and more.

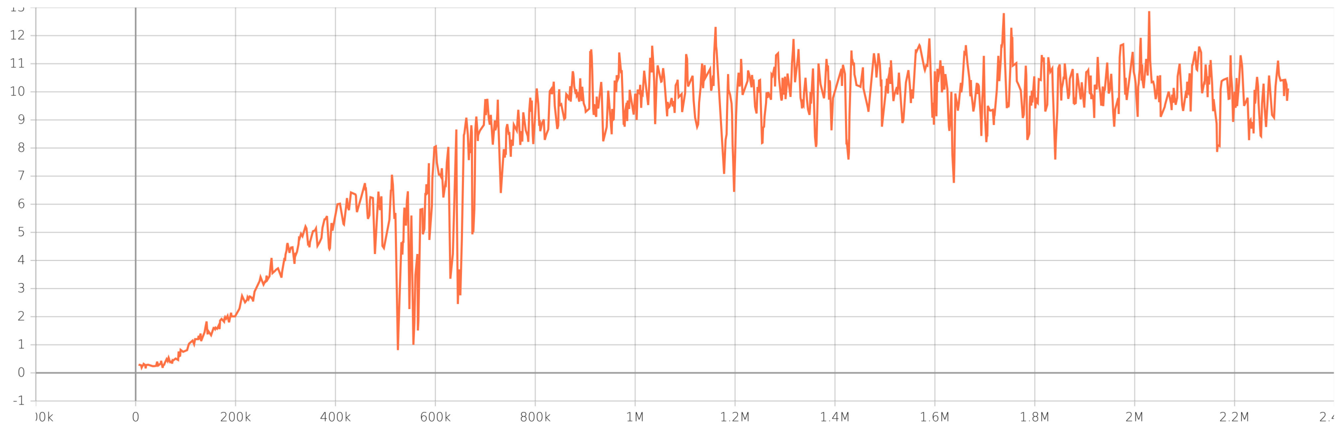

Figure 2. Average reward after X steps

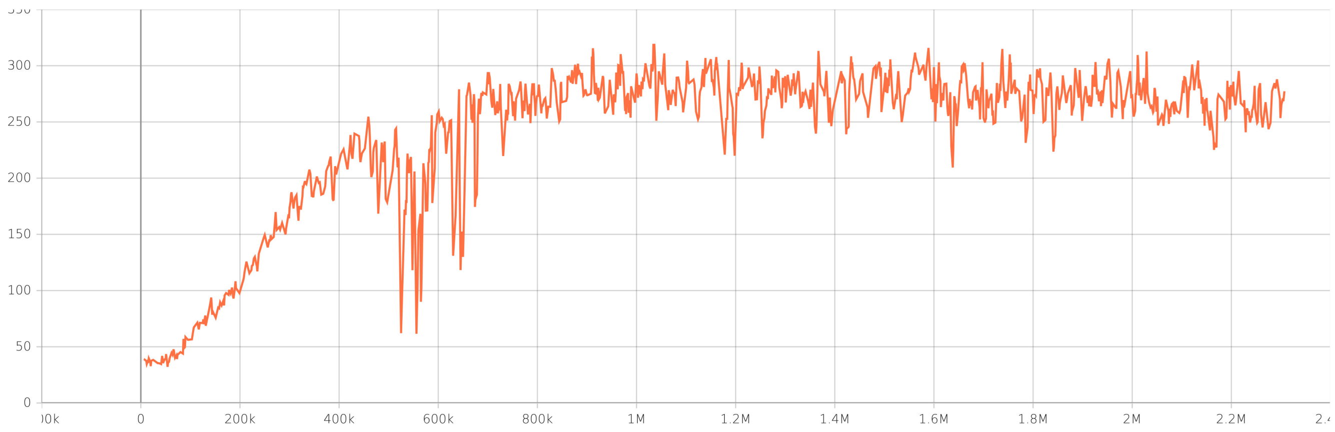

Figure 3. Average episode length after X steps

Conclusions

We are still far from the results presented in the DeepMind papers due to not having a GPU at hand and exhausting Google Colaboratory's RAM, but our agent is indeed learning to play Breakout, albeit at a slow pace. Recent advancements in Deep Q-Learning like the Rainbow Model combine all the flavors of Q-Learning (Double Q-Learning, Dueling Q-Learning, etc.) in one model and outperform the various techniques used in isolation[5].

References

1. Mnih, V. *et al.* Human-level control through deep reinforcement learning. *nature* **518**, 529–533 (2015).
2. Van Hasselt, H., Guez, A. & Silver, D. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, vol. 30 (2016).
3. Bellemare, M. G., Naddaf, Y., Veness, J. & Bowling, M. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.* **47**, 253–279 (2013).
4. Watkins, C. J. C. H. Learning from delayed rewards. (1989).
5. Hessel, M. *et al.* Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence* (2018).