Human-level control through deep reinforcement learning

Marius Stanescu

March 27, 2017

In this document I will give a brief outline of the 2015 Nature paper mentioned in the title [1]. These notes are taken for the Bucharest Computer Vision meet-up organized on the 16th of March, 2017. Since this is the first reinforcement learning paper discussed with the group, I focus more on the basic RL required to understand the paper and the intuition of why the DQN algorithm works. More implementation details or technical elements can be found in the paper or explored in a later meeting.

[1] Volodymyr Mnih and et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533, 26 February 2015

Mnih et al. first introduced the Deep Q-Network (DQN) algorithm in 2013 [2], and the result received a lot of attention as it came when the deep learning craze was starting to take off. Their work is based on Q-value iteration and it was applied to ATARI games helping establish ALE(Arcade Learning Environment) as an RL benchmark [3]. A single architecture (same algorithm, network architecture and hyperparameters) can be used to play 47 ATARI games, more than half at or above human level. The screen is taken as input (pixel level) and Q-values are returned as output (think goodness of all actions). It outperforms classic RL methods[4] that use hand-crafted features without any game-specific knowledge, just by processing the screen pixels.

[2] Volodymyr Mnih and et al. Playing atari with deep reinforcement learning. 19 December 2013

There were some breakthrough publications in computer vision and speech recognition such as AlexNet (2012), ImageNet competition etc.

[3] Marc G Bellemare and et al. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.*, 47:253–279, 2013

[4] Note that these are quite weak baselines, if we look at what we have now after tuning/improving for a couple years on this dataset

This document continues with an introduction to RL, building up from basics up to Q-value iteration and some of its modifications required to understand DQN. This is followed by a short explanation of DQN and what improvements in makes, a brief overview of the paper's results and a few conclusions and tips. But first, let's talk about the core concepts of reinforcement learning.

What is Reinforcement Learning (RL)?

Reinforcement Learning is an exciting area of machine learning. Informally, it's about learning an efficient strategy (sequences of actions) in a given environment you may or may not know the physics of. Rewards are assigned for a given behavior and over time, the agents learns to reproduce that behavior in order to receive more rewards.

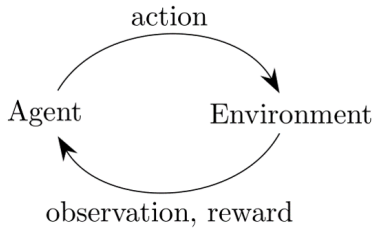


Figure 1: RL - an iterative trial and error process.

Compared to supervised learning we usually don't have full access to function to be optimized and need to query it through interaction

MDPs and terminology

Formally, an environment is defined as a Markov Decision Process (MDP) or Partially Observed MDP (POMDP):

- **S** - set of states of the environment
- **A** - set of actions, which the agent selects from at each time step
- $\mathbf{P}(\mathbf{r}, \mathbf{s}'|\mathbf{s}, \mathbf{a})$ - a transition probability distribution of receiving a reward r and transition to s'
- γ - a discount factor $\gamma \in (0, 1)$, indicating preference for present rewards compared to future rewards

In a deterministic environment (chess) this will look like a delta function (1 for s' and 0 everywhere else). Also, P can also be provided as $P(s'|s, a)$ and one of $R(s), R(s, a)$ or $R(s, a, s')$.

Other terminology:

Episode - a complete play from one of the initial states to a final state $(s_0, a_0, r_0), (s_1, a_1, r_2), \ldots, (s_n)$.

Cumulative (discounted) reward - also called return, it is the (discounted) sum of reward accumulated throughout an episode $R = r_0 + r_1 + \ldots + r_{T-1} + V(s_T)$ or $R = r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots$

Policy π - the agent's strategy to choose an action at each state, can be stochastic $\pi(a|s)$ or deterministic $\pi(s) = a$.

The goal is to find the optimal policy π^* that maximizes the expected return: $\max_\pi E[r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots]$. This is usually intractable, so we need to train an agent such that his policy converges to the theoretical optimal policy.

Sate value function $V^\pi(s)$ - the expected cumulative reward from a state s following that policy

$$\begin{aligned} V^\pi(s_0) &= E_\pi[r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots] \\ &= E_\pi[r_0 + \gamma V(s_1)] \\ &= \max_a Q^\pi(s_0, a) \end{aligned}$$

Say we have V^* the value function of the optimal policy π^*. If we want to retrieve π^* we can do so greedily by choosing $a = arg\max_a[E_\pi(r_0 + \gamma V(s_1))]$. However, in the model-free setting we don't know s_1, so that's why we need Q functions. We can now retrieve π^* by $a = arg\max_a[Q^{\pi^*}(s_0, a)]$. No more uncomputable expectations, cool! We moved the expectation inside Q, which we now hope is smooth enough to be reasonably approximated with NNs.

State-action value function - also called **Q function**

$$\begin{aligned} Q^\pi(s_0, a_0) &= E_{s_1 \sim P(s_1|s_0, a_0)}[r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots] \\ &= E_{s_1 \sim P(s_1|s_0, a_0)}[r_0 + \gamma V^\pi(s_1)] \\ &= E_{s_1 \sim P(s_1|s_0, a_0)}[r_0 + \gamma E_{a_1 \sim \pi}[Q^\pi(s_1, a_1)]] \end{aligned}$$

Bellman equation and backups

The equation

$$Q^\pi(s_0, a_0) = E_{s_1 \sim P(s_1|s_0, a_0)}[r_0 + \gamma E_{a_1 \sim \pi}[Q^\pi(s_1, a_1)]]$$

is called the Bellman equation. For an optimal policy π^* we can define the optimal $Q^* = Q^{\pi^*}$ which satisfies $Q^*(s, a) = \max_\pi Q^\pi(s, a)$.

As hinted to in the sidenotes, $\pi^*(s) = arg\max_a Q^*(s,a)$. We can re-write the Bellman equation for Q^* as:

$$Q^*(s_0,a_0) = E_{s_1\sim P(s_1|s_0,a_0)}[r_0 + \gamma \max_{a_1} Q^*(s_1,a_1)]$$

The optimal strategy is to select a_1 that maximizes the expected value. Next, the Bellman backup operator $\mathcal{T}$ applied to a function Q is

$$[\mathcal{T}Q](s_0,a_0) = E_{s_1\sim P(s_1|s_0,a_0)}[r_0 + \gamma \max_{a_1} Q(s_1,a_1)].$$

It can be proved that Q^* is a fixed point of this operator, meaning that $\mathcal{T}Q^* = Q^*$. Moreover, if we apply this operator to any initial Q, the series converges to Q^*: $Q, \mathcal{T}Q, \mathcal{T}^2Q, \ldots \to Q^*$.

Q-Value iteration

We now have all the ingredients for the basic version of Q-value iteration: start with a random Q and iteratively apply $\mathcal{T}$ until convergence. This works because we can greedily compute $\max_a Q(s,a)$ inside $\mathcal{T}$, even without knowing the transition probabilities (model-free RL).

Algorithm 1 Q-Value Iteration

Initialize $Q^{(0)}$
for $n = 0, 1, 2, \ldots$ until termination condition **do**
  $Q^{(n+1)} = \mathcal{T}Q^{(n)}$
end for

Figure 2: Q-Value iteration

We can compute an unbiased estimator of the RHS in the Bellman equation using a single sample – a single transition (s,a,r,s') –, and it does not matter what policy was used to select actions (off-policy):

$$[\widehat{\mathcal{T}Q}](s_0,a_0) = r_0 + \gamma \max_{a_1} Q(s_1,a_1)$$

Backups still converge to Q^* even with this noise! [5]

[5] Tommi Jaakkola, Michael I Jordan, and Satinder P Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Comput.*, 6(6):1185–1201, 1994

Then, in the first version of Q-Value we replace $Q^{(n+1)} = \mathcal{T}Q^{(n)}$ by

$$Q^{(n+1)}(s,a) = mean\{\widehat{\mathcal{T}Q}_t, \forall t \text{ such that } (s_t,a_t) = (s,a)\}$$
$$\text{and } \widehat{\mathcal{T}Q}_t = r_t + \gamma \max_{a_{t+1}} Q^{(n)}(s_{t+1},a_{t+1}).$$

The mean is computed after interacting with the environment for a fixed number of timesteps K, and for all (s,a) pairs. This is known as sampling-based Q-value iteration.

Next, we want to parameterize Q with a network Q_θ, instead of using a tabular form. For it to work, we need to rephrase this as an optimization problem [6]:

$$Q^{(n+1)}(s,a) = arg\min_Q \sum_{t=1}^{K} \|\widehat{\mathcal{T}Q}_t - Q(s_t,a_t)\|^2$$

[6] Recall that we can express $mean\{\hat{x}_i\} = arg\min_x \sum_i \|x_i - x\|^2$

The resulting batch algorithm is called Neural-Fitted Q-Iteration [7].

Algorithm 2 Neural-Fitted Q-Iteration (NFQ)[1]

Initialize $\theta^{(0)}$.
for $n = 0, 1, 2, \textit{dots}$ **do**
  Run policy for K timesteps using some policy $\pi^{(n)}$.
  $\theta^{(n+1)} = \text{minimize}_\theta \sum_t \left(\widehat{\mathcal{T}Q_{\theta^{(n)}}}_t - Q_\theta(s_t,a_t)\right)^2$.
end for

Figure 3: Neural-Fitted Q-Iteration

[7] Martin Riedmiller. Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328, 2005

Finally, we can adapt this batch method into an online/incremental algorithm in which we can simply plug SGD, for example. Instead of the full backups $Q = \widehat{\mathcal{T}Q}_t$ we were doing before, we can do a partial backup:

$$
\begin{aligned}
Q &= \epsilon\widehat{\mathcal{T}Q_t} + (1-\epsilon)Q \\
&= Q - \epsilon(Q - \widehat{\mathcal{T}Q_t}) \\
&= Q - \epsilon\nabla_Q \|Q - \widehat{\mathcal{T}Q_t}\|^2/2
\end{aligned}
$$

which is equivalent to a gradient step on the squared error.

Algorithm 3 Watkins' Q-learning / Incremental Q-Value Iteration

Initialize $\theta^{(0)}$.
for $n = 0, 1, 2,$ *dots* **do**
  Run policy for K timesteps using some policy $\pi^{(n)}$.
  $g^{(n)} = \nabla_\theta \sum_t \left(\widehat{\mathcal{T}Q_t} - Q_\theta(s_t, a_t)\right)^2$
  $\theta^{(n+1)} = \theta^{(n)} - \alpha g^{(n)}$ (SGD update)
end for

Figure 4: Incremental Q-Value Iteration

Q-Value Iteration requires some boundary conditions such as the reward received at the final state of the episode being 0. In Go or Chess, a $+1$ reward is usually assigned to the transitions that lead to a final winning board (respectively -1 for a loosing board) and 0 for everything else. Hence, the states close to the terminal states are the first to converge because they are close to the *true* label, victory or defeat. The algorithm will diffuse the Q-values for the more distant states between the two extremes $[-1;1]$, and a transition with Q value near to o will lead to a balanced board, while the closer to 1 the closer to a certain victory. As long as we see enough transitions near the terminal states, Q-learning will work reasonably well. Using a function approximator means it will be able to generalize its learning from visited states to states never seen while training. The network should be able to abstract patterns and understand the strength of an action based on previously seen patterns.

DQN

The algorithm, shown below (as copied from the paper) is a hybrid of online and batch Q-value iteration, interleaving optimization with data collection.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and 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 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 $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in D
    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

ϕ_t is the processed state, we can just think of it as s_t to maintain consistency with our notation so far.

The major improvements introduced or additions to the vanilla Q-value iteration are described in the following subsections, roughly in order of importance.

Experience Replay

One of the issues with the vanilla Q-Value iteration algorithm is that the transitions are very correlated. After all, they are extracted from the same episode! The experience replay is a buffer of the last N transition (a million in original paper), from which you randomly sample a minibatch (say 32 transitions) and use this to compute the update (instead of the last transition only).

Why is this a valid approach? Because, as seen before, the backup $\mathcal{T}Q^{(n)}$ can be performed using off-policy data - so these can be transitions generated by past iterations of the policy we're optimizing.

Moreover, now the algorithm sees each transition (s, a, r, s') more than once, leading to better data efficiency and reward propagation [8]. Also, there is increased stability as this history contains data from many past policies derived from $Q^{(n)}, Q^{(n-1)}, Q^{(n-2)}, \ldots$ and because it changes rather slowly.

[8] Remember, the reward has to be propagated from the states where you actually receive it, backwards. You actually score in Pong many frames after the one in which you actually hit the ball correctly!

Target network

The second major addition is the use of a second network during the training procedure:

Q is used to select actions and to compute the gradient descent step on at every time step

$\hat{\mathbf{Q}}$ is the *target* network, a second Q used to compute the Bellman backups, the target Q-values you are trying to get closer to (the further away, the bigger the loss in SGD).

Why not just use the current Q as backup target? The issue is that at every training step, the Q-network's values shift, and if we are using a constantly shifting set of values to adjust our network values, then the value estimations can easily spiral out of control. The network can become destabilized by falling into feedback loops between the target and estimated Q-values. To mitigate that risk, the target network's weights are fixed, and only periodically (every C steps) updated to the primary Q-networks values. Chasing a fixed target is easier than a moving one, and thus training will be more stable.

Instead of updating the target network periodically and all at once, it is better to update it frequently, but slowly. This technique was introduced in another DeepMind paper, where they found that it stabilized the training process.

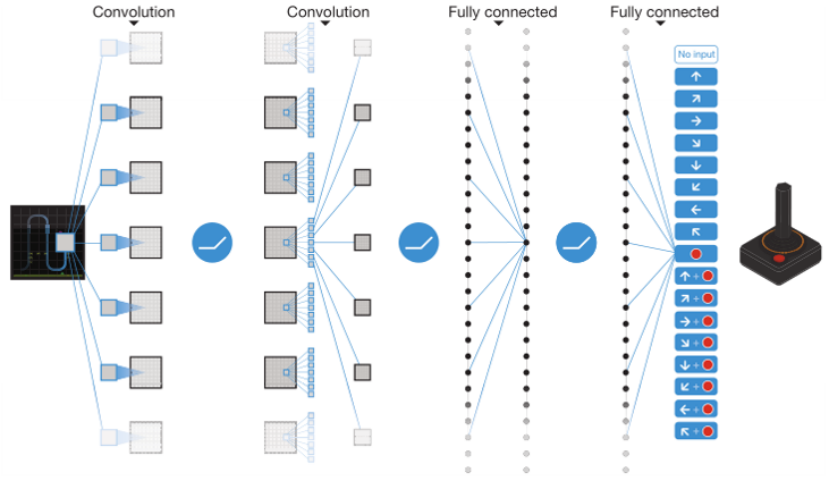
Timothy P Lillicrap and et al. Continuous control with deep reinforcement learning. 2015

Exploration vs. exploitation

Exploring randomly will eventually converge to the optimal policy ... but only after almost infinite time. Usually the state space is too big to visit every possible trajectory, and thus we should try to focus the exploration on the more relevant parts of the state space. A common way (also used here) is $\epsilon-$greedy exploration: choose an action at random with a small probability ϵ or the best action according to the current policy otherwise [9].

[9] ϵ is usually annealed over time to transition from exploration to exploitation

Network architecture



The two convolutional layers have 32 8x8 filters with stride 4 and 64 3x3 filters with stride 1. Afterwards there are two fully connected layers of 512 and 18 units, respectively. Each hidden layer is followed by a rectifier nonlinearity.

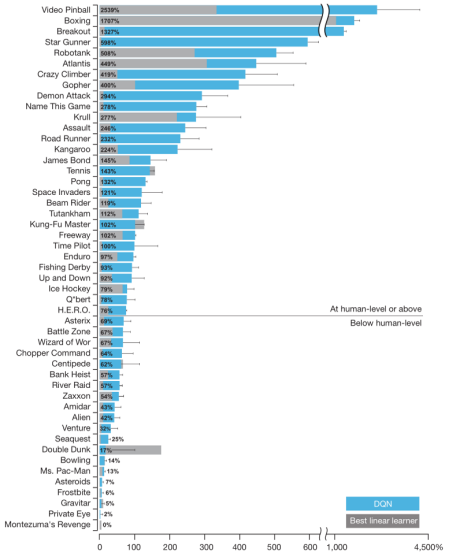
The input images are resized to 84×84 grayscale, and only 1 in 4 frames is actually processed. For the following 3 images, the last action is repeated. To give information to the network about the current momentum, the last 4 frames (with skip frame, you pick 1 every 4) are stacked into 4 channels. Those 4 frames represent a history we need to construct[10], as we are not using Recurrent Neural Networks (RNNs). So we need 16 game frames to construct our $84 \times 84 \times 4$ input. The output contains the Q value of every action possible (in different games some of these will be disabled, the minimum and maximum number of actions are 4 and 18).

[10] A screen snapshot is a partial observation of the full state, since we do not have access to speed vectors for example. Partially observed MDPs (POMDPs) can be reduced to fully observed MDPs by accumulating all previous states into a history. We make do with the last few observations.

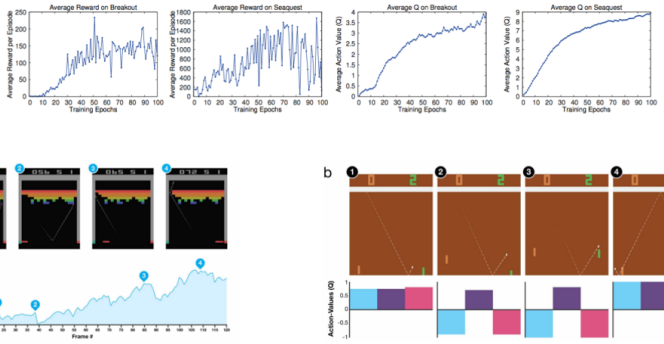
Results

Game	With replay, with target Q	With replay, without target Q	Without replay, with target Q	Without replay, without target Q
Breakout	316.8	240.7	10.2	3.2
Enduro	1006.3	831.4	141.9	29.1
River Raid	7446.6	4102.8	2867.7	1453.0
Seaquest	2894.4	822.6	1003.0	275.8
Space Invaders	1088.9	826.3	373.2	302.0

TODO: more text explaining results. Better ($\geq$ 75%) than humans in more than half of the games. Result on the right are scaled from 0%– random play to 100%– professional human games tester.



Are Q-Values meaningful?

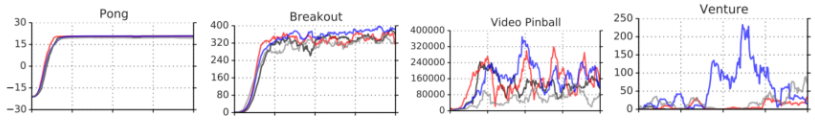


Yes, but usually Q ends up severely overestimating the actual returns. Also, it can shift a lot without significant impact to the loss. Double DQN addresses this. However, DQN works decently in practice because the differences between the Q-value of different actions is more important than their absolute value.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. 2015

Conclusions

- Single architecture (same algorithm, network architecture and hyperparameters) used to learn control policies for 49 different environments.
- Minimal prior knowledge, only pixels and score as input.
- Not equally reliable on all tasks.



- Improvements from recent work (to be discussed later)
  - Double DQN: closer DQN estimates to true values
  - Dueling nets: separately estimate V and A, $Q(s,a) = V(s) + A(s,a))$
  - Prioritized replay: favor important time steps - large gradients

  Some other tips:
- Large replay buffers (1 M) improve robustness.
- Converges slowly: ATARI often needs 10-40M frames (hours on a GPU) to get better than a random policy.
- Run at least 2-3 different seeds when experimenting.
- Learning rate scheduling is important (high rates in initial exploration, decrease over time).

References

Marc G Bellemare and et al. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.*, 47:253–279, 2013.

Tommi Jaakkola, Michael I Jordan, and Satinder P Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Comput.*, 6(6):1185–1201, 1994.

Timothy P Lillicrap and et al. Continuous control with deep reinforcement learning. 2015.

Volodymyr Mnih and et al. Playing atari with deep reinforcement learning. 19 December 2013.

Volodymyr Mnih and et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 26 February 2015.

Martin Riedmiller. Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328, 2005.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. 2015.