Playing Doom with Reinforcement Learning

Federico Cichetti
Alma Mater Studiorum - University of Bologna
`federico.cichetti@studio.unibo.it`

Abstract

The goal of this project is to analyse how different modern Deep Reinforcement Learning (RL) algorithms allow agents to learn to solve the problem of navigating and surviving in a 3D environment based on the famous first-person shooter video game Doom. To this end, we built a simple but extensible RL framework and implemented two classic algorithms from scratch: DQN, an off-policy value-function approximation method, and A2C, a modern policy gradient method based on Actor-Critic. We compare their architectures and their performances on a simple task, as well as highlighting our implementation decisions.

1 Introduction

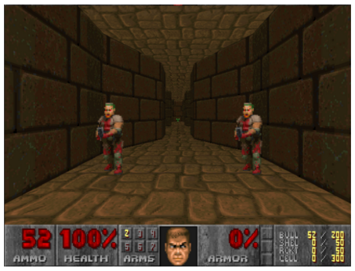
The field of Reinforcement Learning (RL) has achieved some exciting results in the past few years, thanks to Deep RL techniques allowing effective and efficient solutions for problems that have always been considered impossible to solve by machines due to their high number of states and complexity. The main reason behind these achievements is the passage from tabular RL algorithms to value approximation methods like Deep RL. Many recently proposed algorithms based on neural networks have deeply impacted the RL community and are now being considered baselines for all future research. Among the large amount of available methods, we chose to implement a classic DQN [3] [4] and compare it with a recent Actor Critic-based method: A2C (the synchronous version of the A3C algorithm presented in [5]).

The problem we chose to tackle with these algorithms is to solve a simple scenario based on the famous FPS video game Doom (1993). Doom is a fitting proving ground for RL algorithms and has been used in influential works, such as the study on Curiosity-driven Exploration by Pathak et al. [6]. The reasons are plenty. Firstly. the game environment is fully 3D, which makes it more complex than simple Atari games. At the same time, Doom is extremely lightweight for modern computers, with a minimal rendering overhead and almost no impact on RAM/GPU, leaving more space for DNN computations. The first-person perspective of the game can also be a desirable feature, because the navigational skills required for an agent in this environment will probably be closer to those of environments in the real world. Finally, Doom has a very active community and there are projects like ViZDoom [2] that provide all the necessary tools to integrate bots and agents with the game. ViZDoom also has an OpenAI Gym [1] wrapper, which is what we use in our implementation.

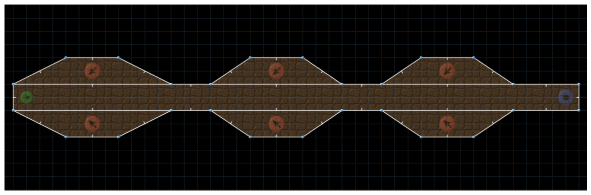
2 Problem

The agent is positioned at the beginning of a short corridor. At the end of the corridor there is a vest that represents the objective: reaching it will positively conclude the episode. The reward for each step is given proportionally to how closer (positive reward) or farther (negative reward) the agent ends up being from the vest after its action with respect to its previous position. In the corridor are also placed 6 enemies, 3 per side, that will shoot at the agent as he walks to the end. The agent can be hit multiple times, but will eventually die as its health runs out. Dying always receives a large negative

Preprint. Under review.



(a) A frame of the game in the problem scenario.



(b) The game map. The green marker corresponds to the player spawn point, the red markers are enemies and the blue marker represents the objective.

Figure 1: Environment representation.

reward. It's almost impossible to reach the end of the corridor without taking care of the enemies first, and to do that the agent is armed with a gun that can kill them in a single shot. Successfully completing the task therefore involves multiple skills: the agent should not only learn to walk towards the objective, but also to defend itself in its advance, learn to aim towards the enemies and shoot them as well as carefully avoiding their shots. The agent has 7 available actions: it can move forward, left, right and backwards, turn left or right and shoot. The environment representation can be seen in Figure 1.

Pre-processing To play the game, we used the official Gym wrapper for ViZDoom. The `step` function of the environment returns 320x240 RGB frames. Following the pre-processing algorithm from [4], each frame obtained by the environment is sent to an instance of the `StateManager`, a utility class that facilitates the production of valid states. The RGB frame is resized to 84x84 and converted into grayscale. Then, we convert the image into an array and normalise its pixel values, dividing them by 255. The pre-processed frame is finally added into a buffer that holds the previous 4 frames (initialised with black images). The 4 frames are stacked into a single 84x84x4 tensor, which is returned and used as input for the models.

3 Algorithms

We tried to solve the problem by means of two powerful and general algorithms: DQN, which was originally built to solve a large amount of Atari 2600 games [4] and A2C, a policy gradient method based on Actor Critic [5]. Figure 2 describes the neural models that have been implemented.

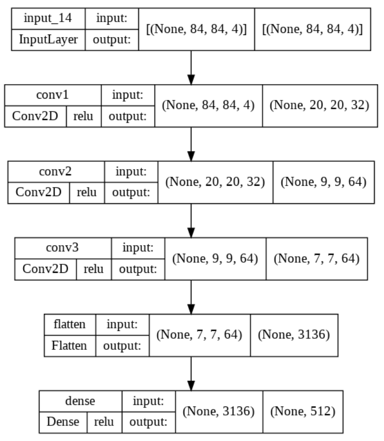
3.1 DQN

DQN (Deep Q-Network) is an action-value function approximation method. The core idea is to use a parameterised function (a neural network) to approximate the optimal Q-value function Q^*:

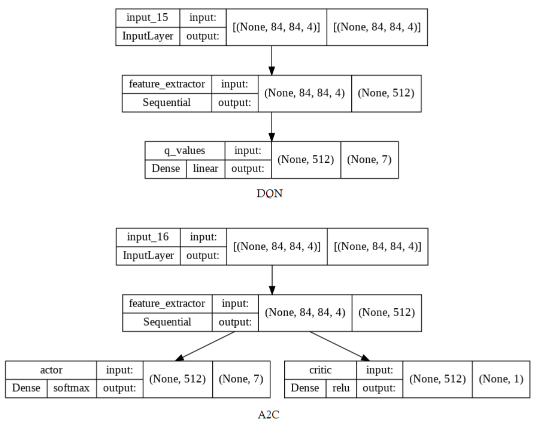
$$\hat{Q}(s,a,\theta) \sim Q^*(s,a) = \max_{\pi} \mathbb{E} \left\{ r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = s, a_t = a, \pi \right\} \tag{1}$$

As in traditional Temporal Differerence (TD) methods, we don't need to have a full sequence of rewards in order to compute updates. At each time step we can sample an action a_t from an ϵ-greedy policy and apply it on the environment to obtain full knowledge of a single transition (s_t, a_t, r_t, s_{t+1}). We can immediately update our Q values based on this transition, using the error between our estimate of the value function at the current state $\hat{Q}(s_t, a_t, \theta)$ and the discounted estimate at the next state summed with the received reward: $r_t + \gamma \max_a \hat{Q}(s_{t+1}, a, \theta)$. Notice that we don't actually compute a_{t+1} and instead pick the maximum among the Q values for state s_{t+1}: that's the main difference between Q-learning and SARSA, the first being an off-policy update method where the exploration policy is a fully greedy policy, while the second is an on-policy method and requires the computation of a_{t+1} with the same ϵ-greedy policy used for a_t. The parameters are then updated by stochastic gradient descent, where the loss function is simply the squared difference between the two quantities:

$$loss = \left(r_t + \gamma \max_a \hat{Q}(s_{t+1}, a, \theta_t) - \hat{Q}(s_t, a_t, \theta_t) \right)^2 \tag{2}$$


input_14 InputLayer input: output: [(None, 84, 84, 4)] [(None, 84, 84, 4)]
conv1 Conv2D relu input: output: (None, 84, 84, 4) (None, 20, 20, 32)
conv2 Conv2D relu input: output: (None, 20, 20, 32) (None, 9, 9, 64)
conv3 Conv2D relu input: output: (None, 9, 9, 64) (None, 7, 7, 64)
flatten Flatten input: output: (None, 7, 7, 64) (None, 3136)
dense Dense relu input: output: (None, 3136) (None, 512)

(a) Architecture of the feature extractor.


input_15 InputLayer input: output: [(None, 84, 84, 4)] [(None, 84, 84, 4)]
feature_extractor Sequential input: output: (None, 84, 84, 4) (None, 512)
q_values Dense linear input: output: (None, 512) (None, 7)
DQN
input_16 InputLayer input: output: [(None, 84, 84, 4)] [(None, 84, 84, 4)]
feature_extractor Sequential input: output: (None, 84, 84, 4) (None, 512)
actor Dense softmax input: output: (None, 512) (None, 7)
critic Dense relu input: output: (None, 512) (None, 1)
A2C

(b) Architectures for the two algorithms (DQN and A2C).

Figure 2: Model architectures.

Therefore, the update rule is:

$$\theta_{t+1} = \theta_t + \eta \left[r_t + \gamma \max_a \hat{Q}(s_{t+1}, a, \theta_t) - \hat{Q}(s_t, a_t, \theta_t) \right] \nabla \hat{Q}(s_t, a_t, \theta_t) \tag{3}$$

Mnih et al. [4] actually go a step further in designing DQNs, trying to address the instability that comes from using non-linear functions approximators. The main sources of instability are the correlations in sequential observations and the correlations between the estimate of action value $Q(s_t, a_t, \theta_t)$ and the target of the update $r_t + \gamma \max_a \hat{Q}(s_{t+1}, a, \theta_t)$, due to the use of the same set of weights. They propose to use an *experience replay buffer* to remove the correlations of the first kind and to instantiate a different *target network* that is used to compute the Q-values in the target with a different set of weights from the original Q network to reduce correlations of the second kind.

Experience replay To remove the correlations in the updates caused by the sequential nature of the experience, we need to compute updates using non-consecutive experiences. This can simply be implemented using a replay buffer. At each step we don't let the agent immediately perform an update, but instead store the transition (s_t, a_t, r_t, s_{t+1}) into the buffer for later use. When we decide to perform an update, we randomly sample a mini-batch of experience from the buffer and use it to update the weights. After a while the new updates will replace the old ones, but the same experience sample can be used multiple times in its lifespan. The use of the experience buffer is justifiable because we are learning off-policy: in on-policy algorithms we cannot use it because the experience sample may be generated with a different policy than the one we are currently using.

Target network If we follow the update step in equation 3, we can see that an increase of $Q(s_t, a_t, \theta)$ also leads to an increase of $Q(s_{t+1}, a, \theta)$ for all a, due to the same weights being used between the two networks. This correlation between predicted and target Q-values harms the convergence of the network. To solve this, the authors propose to use two separate Q-networks: one being the *policy network* with parameters θ, the other being the *target network* with parameters θ^-. At each training step we only update the policy network computing $Q(s_t, a_t)$, while we keep computing the target $Q(s_{t+1}, a)$ with old weights. Every C steps we copy the weights of the policy network into the target. This mechanism adds a delay between the time an update is made and the time that update influences the target, stabilising the training.

Implementation details Our implementation of the DQN is directly taken from the paper by Mnih et al. [4]. We call the first part of the network the *feature extractor*: it's composed by three different convolutional layers, a flattening operator and a final dense layer, each with a rectifier activation function (ReLU). This sub-network extracts the features that are needed for the computation of the Q-values and reduces the $(84 \times 84 \times 4))$ input tensor to a 512-dimensional tensor. Then, we process the features with a fully connected (FC) layer with no activation function, outputting a tensor with 7 values, one for each action, representing $\hat{Q}(s_t, a)$ for each a. For training we use an Adam optimiser with a learning rate of 0.001. We train on mini-batches of 32 samples once every 50 frames. The buffer holds up to 8,000 experience samples. Every 1,000 frames we copy the weights of the policy network into the target network. Both the rewards seen in training and the error term $r_t + \gamma \max_a \hat{Q}(s_{t+1}, a, \theta_t^-) - \hat{Q}(s_t, a_t, \theta_t)$ are clipped in the range $[-1, 1]$. For training, ϵ is set to 1 and linearly decreased up until frame 800,000, where it reaches the minimum value of 0.01.

3.2 A2C

A2C is an Actor Critic method, and as such it aims to learn an approximation of both the policy function (learnt by the actor sub-network) and the value function (learnt by the critic sub-network) which is then used to give a feedback to the actor. The actor provides a mapping from the input state to a probability distribution on the actions (policy, $\pi(A|s_t, \theta_t)$), while the critic predicts the value of a state ($V(s_t, \omega_t)$) according to the current policy:

$$V(s_t, \omega_t) \sim V^{\pi}(s) = \mathop{\mathbb{E}}_{\pi}\{R_t | s_t = s\} = \mathop{\mathbb{E}}_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s\right\} \tag{4}$$

The actor is trained to maximise the total cumulative reward, usually by gradient ascent updates where the quantity to improve is $J(\theta)$, a function of the parameters of the model that expresses its performance. The Policy Gradient Theorem provides an analytic expression that is proportional to the real gradient of $J(\theta)$ and does not depend on the unknown effects of the policy on the state distribution (which, in a naive computation, would have to be considered). A direct derivation from the Policy Gradient Theorem leads to the REINFORCE update rule [7], which is the basis of all policy gradient methods:

$$\theta_{t+1} := \theta_t + \alpha R_t \nabla ln\pi(a_t | s_t, \theta_t) \tag{5}$$

REINFORCE is a Monte Carlo method and as such is unbiased because we have the full sequence of rewards obtained during an episode and thus R_t can be computed exactly. The problem is that it suffers from high variance, because in each episode the trajectories (and by consequence the cumulative reward R_t) can vary greatly, leading to unstable learning. One way to reduce variance is to subtract the learned estimate of the value function (provided by the critic) from the return. This quantity is also known as baseline: $b(s_t) = V(s_t, \omega_t) \sim V^{\pi}(s_t)$, with the difference $R_t - V(s_t, \omega_t)$ being called advantage. The resulting gradient becomes:

$$\nabla ln\pi(A_t | S_t, \theta_t)(R_t - V(s_t, \omega_t)) \tag{6}$$

Differently from REINFORCE, Actor Critic is a TD-based method. We don't only bootstrap from the critic to provide a baseline for the update rule, but we also bootstrap the advantage without waiting for the episode to end. This can be achieved by approximating the full return R_t with an n-step return, leading to the loss for the actor becoming:

$$actor\_loss = ln\pi(A_t | S_t, \theta_t)\left(\sum_{i=0}^{k-1} \gamma^i r_{t+1} + \gamma^k V(s_{t+k}, \omega_t) - V(s_t, \omega_t)\right) \tag{7}$$

k being the number of steps that were played in an episode before a terminal state was reached (upper-bounded by t_{max}). If $k = 1$ we update at each step and the advantage formula becomes the classic TD(0)-error, but we found that updating using mini-batches leads to a more stable training, so we update only at the end of the episode (k = number of episode steps).

As proposed by Mnih et al. [5], we also add the entropy of the policy π to the objective function. The entropy of a probability distribution ($\pi(A|s_t, \theta)$) is defined as:

$$H(\pi(A|s_t, \theta_t)) = -\sum_{a \in A}^{n} \pi(a|s_t, \theta_t) ln\pi(a|s_t, \theta_t) \tag{8}$$

If the probability distribution approaches a deterministic choice (eg. $[0.99, 0.006, 0.004]$), then the entropy is close to 0 (-0.063 for the presented case), while for close-to-uniform probability distributions we have high entropy values. Having the entropy introduced in the minimisation problem forces the actor to produce more even distributions, improving exploration by discouraging convergence to deterministic policies.

The critic is also updated at the end of each episode. Its loss function is simply the squared difference between the approximated return and the predicted value of the current state:

$$critic\_loss = \frac{1}{2}\left(\sum_{i=0}^{k-1}\gamma^i r_{t+1} + \gamma^k V(s_{t+k}, \omega_t) - V(s_t, \omega_t)\right)^2 \tag{9}$$

The update has many similarities to that of DQN because the goal of approximating a value function is the same.

Implementation details. The actor and critic were actually implemented as two separate output layers following a shared feature extractor, whose architecture is equal to the one previously described in Paragraph 3.1. The actor was implemented as a FC layer with 7 output neurons, followed by a softmax activation function that is used to generate probability distributions. The critic is a FC layer with a single neuron and no activation function that directly outputs the value of the input state. The returns are computed at the end of the episode, starting from $R = V(s_k, \omega)$ (or 0 in case of terminal state) and iterating backwards though all time steps t in the episode: $R_t \leftarrow r_t + \gamma R_{t+1}$. The gradient is not tracked for this operation: the target is considered a fixed value and the parameters for the critic update come from the second term of the advantage: $V(s_t, \omega_t)$. The final loss is computed as:

$$total\_loss = actor\_loss - \sigma H(\pi(A|s_t, \theta_t)) + critic\_loss \tag{10}$$

and is backpropagated to all the parameters with an Adam optimiser using a learning rate of 0.001.

4 Results

Figure 3: Average position reached by the agents as a function of played frames.

The plot shown in Figure 3 shows the average position that each agent was able to reach in the corridor as a function of played frames. We also show the results of a baseline agent that simply plays a random action at each time step. In general, the agents were not able to fully solve the problem, although there clearly are signs of improvements over time. The simplicity of the feature extractor might be one of the factors that explain the underwhelming results of our models. We may have underestimated the relative complexity of learning to survive and navigate in a 3D environment with respect to making decisions in the 2D environments of Atari games, for which the feature extractor we used in our experiments was originally built. We also believe that the task is difficult enough to require more training episodes than those we could afford with our time and resource constraints.

5 Conclusion

The comparison between these two models highlights that performance in Reinforcement Learning is not just a matter of building deeper and more powerful architectures, but also depends on the algorithm that is used for the updates: small variations can really make a difference. In the future we might try to expand the neural architecture used in the experiments and do some additional hyperparameter tuning. Since we have built a highly extensible framework, we could additionally implement more algorithms, enabling a broader analysis on the state of RL techniques. In general, we believe that there's a high chance that more intelligent behaviour would appear taking into account the above considerations; nonetheless, implementing these algorithms has been greatly beneficial to the understanding of the subject.

References

[1] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016.

[2] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski. ViZDoom: A Doom-based AI research platform for visual reinforcement learning. In *IEEE Conference on Computational Intelligence and Games*, pages 341–348, Santorini, Greece, Sep 2016. IEEE. URL `http://arxiv.org/abs/1605.02097`. The best paper award.

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

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. 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. *Nat.*, 518(7540):529–533, 2015. doi: 10.1038/nature14236. URL `https://doi.org/10.1038/nature14236`.

[5] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In M. Balcan and K. Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 1928–1937. JMLR.org, 2016. URL `http://proceedings.mlr.press/v48/mniha16.html`.

[6] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell. Curiosity-driven exploration by self-supervised prediction. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 2778–2787. PMLR, 2017. URL `http://proceedings.mlr.press/v70/pathak17a.html`.

[7] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8:229–256, 1992. doi: 10.1007/BF00992696. URL `https://doi.org/10.1007/BF00992696`.