Experiments on Deep Q-Network (DQN) and Asynchronous Advantage Actor-Critic (A3C)

Hugo Dupré
MVA Master's degree
École Normale Supérieure
Cachan, France
hugo.dupre@ensae.fr

Pierre Perrault
MVA Master's degree
École Normale Supérieure
Cachan, France
perrault.pierre.12@gmail.com

Abstract

In this report, we will present our work on two of the most recent reinforcement learning agents : Deep Q-Network (DQN) and Asynchronous Advantage Actor-Critic (A3C). These agents, and mostly the first one, received great attention since they were able to *solve* impressive reinforcement learning environments such as Atari 2600 games. Here we will focus on the reinforcement learning theory behind these agents, while recreating some of the impressive results they are able to provide. Finally, we will try to get an intuition on how to improve those methods.

1 Introduction

Since the dawn of Deep Learning, going "Deep" in other machine learning field has been a general tendency : that's how the weirdly-called Deep Reinforcement Learning framework appeared. Recently, artificial intelligence has received massive attention, especially deep learning and reinforcement learning. This combined with impressive results understandable by anyone, such as agents more skillful at playing Atari games than humans, made Deep RL go viral. Nevertheless, all theory for these algorithms have existed for several years now. For instance, Q-Learning has been discovered by C. Watkins in his 1989's PhD Thesis ([1]).

What really changed recently is that it is now computationally possible to train very deep neural networks, with millions of parameters. Thus, we can now work with a novel class of functions allowing us to approximate other functions of interest. This is useful for Approximate Dynamic Programming, where the goal is to approximate Q-functions, value functions or even directly policies in order to solve a reinforcement learning environment. Using reinforcement learning theory and neural networks, agents can be trained to do various task at a human level.

In this report, we will first go over DQN following the Nature article of Google DeepMind by Mnih et Al. ([2]). Since this report is focused on the reinforcement learning theory in the *Deep RL* framework, this will be rather quick because it is plain Approximate Dynamic Programming, as we saw in class. We will go over the clever tricks introduced, and this will help understand the next part about A3C, an agent also proposed by Mnih et Al. ([3]). The reinforcement learning theory behind this agent is much more rich, we will have to go over new concepts that we did not study in class: policy gradient methods, actor-critic framework and variance reduction using the advantage function. Then, we will present our implementation of these algorithms, and the quantitative results obtained in different benchmarks environments for reinforcement learning, in terms of performance and training times. Finally, we will conclude and try to give an insight on what could be done to improve those methods.

2 Deep Q-Network

As we said in the introduction, **DQN is basically an implementation of online approximate value iteration algorithm, where a neural network is used for the regression task.** Nevertheless, there are 3 main innovations, simple but clever, to this straightforward implementation. We will cover them after a quick reminder of what online approximate value iteration is.

Formally, let $\mathcal{M} = \{\mathcal{X}, \mathcal{A}, \mathcal{P}, \mathcal{R}\}$ be the Markov Decision Process of the environment considered. We have fully working algorithms from dynamic programming if the transition matrix and the reward function, respectfully $\mathcal{P}$ and $\mathcal{R}$, are known, and that we can interact from the environment (i.e sample from $\mathcal{X}$ when choosing in $\mathcal{A}$). Two of these algorithms are Value Iteration and Policy Iteration, they converge towards the optimal value function V^* and optimal policy Π^*.

However, in practice, $\mathcal{P}$ and $\mathcal{R}$ are not available, but we can interact with the environment : this is where approximate dynamic programming comes into play. Basically, instead of updating the Q-function by applying the Bellman Operator $\mathcal{T}$ each iteration $k \in \{1, .., K\}$ like this :

$$Q_{k+1}(x, a) = \big[\mathcal{T}(Q_k)\big](x, a) = \mathcal{R}(x, a) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(y|x, a) * \max_{b \in \mathcal{A}} Q_k(y, b)$$

It is possible is to learn from transitions (x, a, r, x') by creating an unbiased estimator y of $\big[\mathcal{T}(Q_k)\big](x, a)$:

$$\mathbb{E}[y] = \mathbb{E}[r + \gamma \max_{b \in \mathcal{A}} Q_k(x', b)] = \mathcal{R}(x, a) + \gamma \int_{\mathcal{X}} \max_{b \in \mathcal{A}} Q_k(y, b) \, \mathrm{d}p(y|x, a) = \big[\mathcal{T}(Q_k)\big](x, a)$$

We first need to define a class of parametric functions $\mathcal{F}_\theta = \{f(x, a, \theta) : \mathcal{X} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}\}$, in order to approximate the real Q-function Q^*. The set $\mathcal{F}_\theta$ is a neural network in DQN : it can be a convolutional neural network if the input are images, but it is not compulsory : the architecture of the neural network should be adapted to the environment it is trying to solve. Then the loss function, for any transition (x, a, r, x'), used to train the neural network is, using the previous unbiased estimator :

$$L(\theta_k) = \mathbb{E}\Big[\big[(r + \gamma \max_{b \in \mathcal{A}} f(x', b, \theta_{k-1})) - f(x, a, \theta_k)\big]^2\Big]$$

With that, we can train a neural network to estimate the Q-function by a class of parametric functions $\mathcal{F}_\theta$ by simply updating the weights of the neural network every time there is a new transition (x, a, r, x'), using back propagation. This will allow to approximate Q^* in an online fashion, such that :

$$\forall (x, a) \in \mathcal{X} \times \mathcal{A} \quad f(x, a, \theta) \approx Q^*(x, a)$$

This is the straightforward implementation of approximate value iteration, now let's review the three innovations brought by Mnih et Al. ([2]) :

1. Each state x needs to account for momentum and movement. For instance in Atari games, the states have to be a sequence of frames of a given length, in order to describe what is going on is this state (ex : Is the Pong's pad going up or down ?). Thus, this is a modification of the state space $\mathcal{X}$.
2. Performing gradient descent with every new transition does not work, because of the correlation existing among a sequence of transition. In Pong, for example, if one pad is a the bottom of the screen, it will also be close to the bottom of the screen in the next few frames. The solution to address this issue is to create an experience memory, consisting of former transitions that occurred in the game, or in previous game. Then, when receiving a new transition, it is added to the experience memory, and the gradient descent is performed over a batch of transitions randomly sampled from the experience memory. This way, it reduces correlation and allows proper learning.

3. The last trick is to have a target neural network updated every C time-steps, by cloning the current neural network. This means that the targets of the regression task are fixed for C time-steps, which provides better stability for training.

Finally, the last choice to be made is the exploration policy used to address the exploration-exploitation dilemma. The exploration part is fundamental. In complex environments where there is a goal, the agent is forced to explore enough to at least encounter success in the task it is given : for instance driving up a hill in the classic reinforcement learning environment Mountain Car.

Mnih et Al's DQN uses an ϵ-greedy policy, which is simply to take an action at random with probability ϵ and taking the action that maximizes the current Q-function in state the current state with probability $1 - \epsilon$. A good heuristic is to explore a lot at first with a large ϵ, then gradually decrease ϵ in order to optimize, hoping that the agent explored enough to approximate Q^* well enough.

The full algorithm Deep Q-Network is recapitulated in Algorithm 1.

Algorithm 1 Deep Q-Network

```
Set experience memory D to capacity N
Randomly pick weights θ for the current neural network f
Create target neural network f_t by cloning the current neural network
for episode=1,..,M do
  Get first state x_1
  k ← 1
  while x_k is not terminal do
    Choose action a_k according to ε-greedy policy
    Execute action a_k, observe the transition (x_k, a_k, r_k, x_{k+1})
    if D is full then
      Erase first entry of D
    end if
    Add transition (x, a, r, x') to D
    Sample a random mini-batch of transitions B_k from D
    Use f_t to create targets for the regression task.
    Perform gradient descent on B_k
    if k ≡ 0 (mod C) then
      Update f_t by cloning f
    end if
    k ← k + 1
  end while
end for
```

This algorithm was greatly improved in many ways. One simple thing to do is Double-DQN by Van Hassel et Al. ([4]), where both the current neural network and the target neural network are used to compute the targets for the regression task. Then, a way to improve the exploration is Bootstrap DQN, propose by Osband et Al. ([5]). The idea is to let several agents play, and add their experience to others randomly, in order to improve the exploration.

These extension rely on the concept of experience memory, which is by definition not memory efficient, hence the need for a different extension.

3 Asynchronous Advantage Actor-Critic (A3C)

The Asynchronous framework for deep reinforcement learning, proposed by Mnih et Al. in 2016 ([3]), provides a better performing and memory-efficient way to address the memory problem of DQN.

First of all, it uses the advantage actor-critic framework of reinforcement learning theory. Since it was not studied in the class, we will present it.

3.1 Advantage Actor-Critic framework

For this part, we used the David Silver's great course on Policy Gradient Methods, available on Google DeepMind's YouTube channel ([6]).

For a given environment $\mathcal{M} = \{\mathcal{X}, \mathcal{A}, \mathcal{P}, \mathcal{R}\}$ which we can interact with, it is possible to use value-based methods of approximation, as we saw in DQN. Others methods are policy-based : the goal is to estimate the optimal policy π^* using a set of parametric functions $\mathcal{F}_\theta = \{\pi(x, a, \theta) : \mathcal{X} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathcal{A}\}$. A major drawback of value-based methods is that the policy derived from the approximate Q-function or the approximate value function is always deterministic. Some environments can only be solved with stochastic policies, and it is possible to learn stochastic policies using policy-based methods. Also, with value-based methods, there is a $\max$ operation over the action space needed to compute the policy : it can be very hard, especially with continuous action spaces. Ultimately, the agent will need to derive a policy to interact with the environment, thus the relevance of policy-based methods.

So, the goal is to find θ such that $\pi(x, a, \theta)$ is the best policy. *Best* only has a meaning if there is a criterion to evaluate policies. Several criteria exist for this, and all the following stands for each of the classic criteria. We will use the start value criterion $J(\pi_\theta) = J(\theta) = V^{\pi_\theta}(x_1)$, where x_1 is the first state encountered. Hence, the optimization problem is simply to find θ which maximize $J(\theta)$. The methods to solve this problem is simply to compute the gradient of $J(\theta)$, and update θ in order to modify π_θ in such a way that is increases $J(\theta)$. $\nabla_\theta J(\theta)$ tells us which direction we should go if we want to increase $J(\theta)$. It turns out that this gradient has a simple form, and this is the essential result that makes policy-based methods work :

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta log(\pi_\theta(x, a)) * Q^{\pi_\theta}(x, a)\right] \tag{1}$$

The intuition behind this result is that you want to adjust the policy in the direction that get more value, this is why the Q-function appears. With this result, a simple algorithm called Monte-Carlo Policy Gradient can be derived. With observations of one complete episode, at each time-step, the update on θ is the following :

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta log(\pi_\theta(x_k, a_k)) * v_k$$

Where v_k is the return, given which state the agent was at time-step k. Hence, this is a forward view approach. In this algorithm, the variance is very high : the environments are generally very noisy, so episodes may greatly differ from one another. The actor-critic framework is a solution to reduce this variance. Instead of using the return to estimate the Q-function, we will directly estimate the Q-function, and use this estimate to compute the gradient $\nabla_\theta J(\theta)$. The actor will be taking actions, and he will approximate the policy using the estimated Q-function given by the critic, who will be judging whether the actions taken by the actor are good or not. This means that the critic has to estimate Q^{π_θ}, with a class of parametric functions $\mathcal{F}_w$, while the actor has to estimate π^* using another class of parametric functions $\mathcal{F}_\theta$ and the estimate of his current Q-function given by the critic. Thus, the critic has to evaluate a particular policy given sampled observations following that policy : Temporal Difference methods can be applied to solve this problem, such as $TD(0)$ or $TD(\lambda)$. Finally, using the actor-critic framework, equation (1) becomes :

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta log(\pi_\theta(x, a)) * Q_w^{\pi_\theta}(x, a)\right] \tag{2}$$

Where $Q_w^{\pi_\theta}$ is the estimated Q-function given by the critic.

Finally, another way to reduce the variance of the policy estimation without introducing any bias is to use subtract a baseline function $B(x)$ to $Q_w^{\pi_\theta}$, which only depends of the state x. This does not change the expectation since $\mathbb{E}_{\pi_\theta}\left[\nabla_\theta log(\pi_\theta(x, a)) * B(x)\right] = 0$ for any function that only depends

of x. This is a way to rescale the problem in order to reduce the variance. One particular good choice for $B(x)$ is the value function $V^{\pi_\theta}(x)$. Finally, equation (2) becomes :

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta log(\pi_\theta(x,a)) * A_w^{\pi_\theta}(x,a)\right] \tag{3}$$

Where the advantage function $A_w^{\pi_\theta}(x,a) = Q_w^{\pi_\theta}(x,a) - V_w^{\pi_\theta}(x)$ is introduced. Intuitively, the advantage function tells us how better or worse than usual action a is, depending on the sign of $A_w^{\pi_\theta}(x,a)$. Of course, these are estimates made by the critic, hence the w present in those functions.

The advantage actor-critic framework is compatible with neural networks, and this is what was used by Mnih et Al. ([3]). Not only that, they also implemented the asynchronous reinforcement learning framework, which we will present now.

3.2 Asynchronous reinforcement learning framework

The main idea is to use asynchronous actor-learners, each having its own copy of the environment. The i^{th} actor-learner has its own parameters θ_i and w_i, while there are globally shared parameters θ_g and w_g. An actor-learner will first copy the shared parameters, then he will takes actions according to its policy, observe the rewards and the states visited, accumulate gradients over n time-steps for the policy and the advantage function. Then he will asynchronously update the shared parameters, and repeat this process until the exploration policies of each actor-learner are over.

The parameters update can be done at different time-scales : at each time-steps, every n time-steps or at the end of every episode. This is basically the difference between, respectively, $TD(0)$, $TD(\lambda)$ and Monte Carlo methods. A n time-steps approach is used in [3]. Although they can use a backward view using eligibility traces, they choose to use a forward view, meaning that they have to wait until n time-steps are completed to compute the updates.

Each actor-learner can use a different exploration policy in order to maximize diversity in the exploration, thus improving the actual learning part. In the actor-critic framework, the policies are always changing since they are directly estimated by the actors, thus providing proper asynchronous exploration.

3.3 Combining the previous frameworks : A3C

For simplification purposes, a neural network will be assimilated to its set of parameters.
When using the neural networks θ and w to estimate π_θ and $A_w^{\pi_\theta}$, there is no need of having two separate neural networks to train independently. The parameters θ of the policy and w of the value function are shared over one neural network. In [3], they use a convolutional neural network that has one softmax output for the policy and one linear output for the value function, with all non-output layers shared.

Another trick is adding the entropy of the policy π_θ to the objective function in order to improve exploration by discouraging premature convergence to sub optimal deterministic policies. The gradient of the full objective function including the entropy regularization term with respect to the policy parameters takes the final form, using equation (3) :

$$\nabla_\theta log(\pi_\theta(x,a)) * A_w^{\pi_\theta}(x,a) + \beta \nabla_\theta \mathcal{H}(\pi_\theta(x)) \tag{4}$$

where $\mathcal{H}$ is the entropy (remember that $\pi_\theta(x)$ is a distribution over the action space $\mathcal{A}$. The hyperparameter β controls the strength of the entropy regularization term.

Finally, we get the final algorithm presented in Algorithm 2 by combining all this theory and these tricks.

Algorithm 2 Asynchronous Advantage Actor-Critic : n time-step algorithm for one actor.

Randomly pick shared parameters θ_g and w_g
Initialize global time-step counter: $T \leftarrow 1$
Initialize time-step counter for this actor: $t \leftarrow 1$
while $T < T_{max}$ **do**
  Reset accumulated gradients : $\mathrm{d}\theta = 0$ and $\mathrm{d}w = 0$
  Synchronize shared parameters with actor parameters : $\theta = \theta_g$ and $w = w_g$
  $t_{start} = t$
  Get first state x_t
  while x_t is not terminal or $t - t_{start} < n$ **do**
    Choose action a_k according to current policy $\pi_\theta(x_t)$
    Execute action a_k, observe the transition (x_k, a_k, r_k, x_{k+1})
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  end while
  Set $R = V_w^{\pi_\theta}(x_t)$ for any state x_t
  for $i \in \{t-1, .., t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Policy gradient : $\mathrm{d}\theta = \mathrm{d}\theta + \nabla_\theta log(\pi_\theta(x_i, a_i)) * (R - V_w^{\pi_\theta}(x_i))$
    Value gradient : $\mathrm{d}w = \mathrm{d}w + \frac{\partial (R - V_w^{\pi_\theta}(x_i))^2}{\partial w}$
  end for
  Perform asynchronous update of θ_g and w_g using $\mathrm{d}\theta$ and $\mathrm{d}w$
end while

4 Experiments

4.1 DQN

First, we implemented the DQN algorithm on Python. We used Keras on top of Theano in order to construct the neural network. Keras is a very-high level open source library for Deep Learning, which is exactly what was needed here. We followed a nice tutorial ([7]), which helped us understand how Keras works. Then, for all experimentation, we used OpenAI Gym ([8]), a wonderful website which provides ready-to-use classic reinforcement learning environments such as Atari or CartPole.

Then, we coded entirely the algorithm DQN and uploaded our results to the OpenAI Gym leaderboard. The code is available on GitHub : `https://github.com/Caselles/Deep_Q-Network` and all our results are available on OpenAI Gym : `https://gym.openai.com/users/Caselles`.

DQN works very well on *simple* environments, such as CartPole, or Acrobot. The results are very satisfying : in a small number of episodes, the agent learns how to play properly. For these environments, there is no need for a convolutional neural networks. Hence, the learning is also pretty fast, that allowed us to tune the hyperparameters easily. For instance, on CartPole, the learning curve with the accumulated reward over episodes (Figure 1) shows the agent find the optimal policy rather quickly, in only 20 episodes.

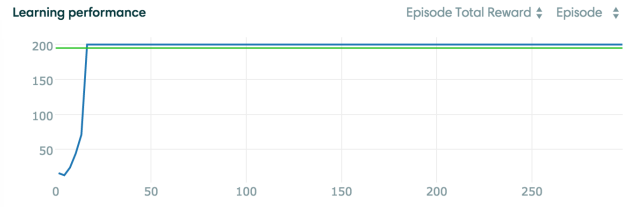

Figure 1: Learning performance of DQN on CartPole.

We averaged, on 100 trials, the number of episodes needed to solve the environment with DQN. Solving the environment means having a high enough average score on 100 consecutive episodes. The results are available in Table 1.

Table 1: Average number of episodes needed to solve the episode

Environment	CartPole	Acrobot
Average number of episode needed to solve environment	21.6	52.3

However, we never managed to make it work on several environments, either because of learning time or local optimas. For Atari games, the learning times on the best computer we could get our hands on, a MacBook Pro equipped with a NVIDIA GeForce GT750M 2048 Mo GPU, were simply too long to let us tune the hyperparameters. It also never worked on the MountainCar environment, because of local optimas (see `https://gym.openai.com/evaluations/eval_hYYZuwQbQG6v2Fz5yIeQSA#reproducibility`) : the agent car was always trapped in the valley, and never reached the flag. It seems that whenever there is a particular thing to do that will give a lot of rewards, DQN will struggle to find the optimal policy. On the contrary, if the agent has to get gradually better in the environment, then DQN will likely work well. The algorithm is also very sensitive to the random parameters initialization.

For the actual code, we used RMSProp and Adam for optimizers, they were really fast and worked well. The neural networks consisted of 2 fully connected layers with roughly 100 hidden units, with the $RELU$ function as the activation function. For more details, see the GitHub code.

4.2 A3C

Computationally speaking, the algorithm A3C relies on a distributed architecture and the asynchronous framework. Each actor-learner has to compute its gradient and then communicate it to the shared critic. This is possible to code in libraries like TensorFlow or Theano, but it is quite hard to make it work (graph framework), so we used this code in GitHub : `https://github.com/ppwwyyxx/tensorpack/tree/master/examples/OpenAIGym`. Our goal was rather to understand the reinforcement theory behind the algorithm rather than spending a lot of time coding massively distributed architecture. Furthermore, the asynchronous part has to be specifically coded otherwise the performance are very low. Using this repository, we were able to reproduce state-of-the-art results on Atari Games, for example in Breakout, Space Invaders or Pong (see `https://gym.openai.com/users/Caselles`). In Pong, the reward per episode curves obtained is presented in Figure 2.

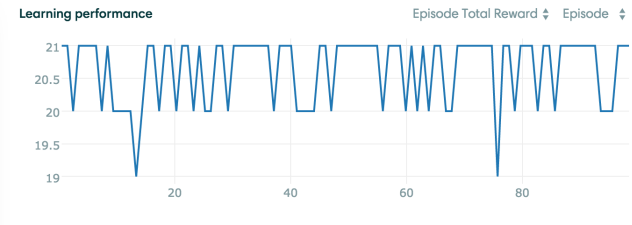

Figure 2: Performance of A3C on Pong.

The agent easily beats its *AI* counter-part, using mostly the same method : it hits the first ball on the edge of its pad, which makes it goes very fast so that the enemy does not manage to hit the ball back. The A3C agent has learn through exploration this quasi-optimal policy, which is exactly what it is supposed to do. On Space-Invaders and Breakout, the agent behavior can be considered as better than an expert human player. Even when the speed is incredibly high, it can survive and win the game.

Even though we couldn't properly compared the two algorithms on Atari Games, we know that A3C outperforms DQN. Some Atari Game remain unsolved, there are still improvements to make. For instance, A3C does not succeed at playing at Montezuma's revenge, an Atari Game that involves long-term planning. Recently, a novel agent called UNREAL ([9]) has been proposed by Jadeberg et Al. (DeepMind). It uses auxiliary control tasks and auxiliary reward tasks combined to A3C, and is nearly at human-level in Montezuma's revenge.

5 Conclusion and possible improvements

DQN and A3C can achieve human-level performance on a wide variety of classic reinforcement learning environments, by taking advantage of strong reinforcement learning theory and the novel framework that is deep learning.

Nevertheless, the methods can be improved. One straightforward improvement would be to perform a backward view rather than a forward view in A3C. Also, the architecture of the neural networks can be tuned : there is still a lot of discussion on how to tune convolutional neural networks' architectures. Some of the recent works shows that you should maybe use a lot of small convolutions. More generally, the hyperparameters tuning is an issue in this kind of models. There a simply too much hyperparameters to do classic grid-search cross validation : all the neural networks' hyperparameters, γ, number of actors, learning rate of the optimizer, n, β,... Moreover, the model's performance is very sensitive to the tuning of these parameters. Finally, exploration seems to be the most important aspect of these models : A3C provides a different, and better, exploration than the classic ϵ-greedy method used in DQN. Is it an optimal exploration though ? This could be a very interesting domain of research.

This project has given us the chance of discovering deep learning while working on very interesting reinforcement learning theory, using state-of-the-art methods. In this sense, it was a really rewarding experience. We would like to thank A. Lazaric for letting us work on this, and for his continuous help throughout the project.

References

[1] C. J. C. H. Watkins, *Learning from delayed rewards*. PhD thesis, University of Cambridge England, 1989.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[3] 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," *arXiv preprint arXiv:1602.01783*, 2016.

[4] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning,"

[5] I. Osband, C. Blundell, A. Pritzel, and B. Van Roy, "Deep exploration via bootstrapped dqn," *arXiv preprint arXiv:1602.04621*, 2016.

[6] D. Silver, "Rl course by david silver - lecture 7: Policy gradient methods."

[7] "Keras plays catch." `http://edersantana.github.io/articles/keras_rl/`.

[8] "Openai gym." `https://gym.openai.com/`.

[9] M. Jaderberg, V. Mnih, W. M. Czarnecki, T. Schaul, J. Z. Leibo, D. Silver, and K. Kavukcuoglu, "Reinforcement learning with unsupervised auxiliary tasks," *arXiv preprint arXiv:1611.05397*, 2016.