Comparing CPU and GPU performance on Deep Q-Network algorithm

Hugo Dupré
ENSAE ParisTech & École Normale Supérieure
hugo.dupre@ensae.fr

Geoffrey Chinot
ENSAE ParisTech & Polytechnique
geoffrey.chinot@ensae.fr

Abstract

In this report, we will present our work on one of the most recent reinforcement learning agent : Deep Q-Network (DQN) and one of its variant (Double-DQN). This agent received great attention since it was able to *solve* impressive reinforcement learning environments such as Atari 2600 games. Here we will focus on presenting the algorithm and then explaining differences in training times when using CPU or GPU, while recreating some of the impressive results they are able to provide.

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, with the use of GPUs. 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]). We will go over the clever tricks introduced by these researchers to make this algorithm work. We will also explain an intuitive extension of DQN, following the work of Van Hasselt et Al. ([3]) : Double-DQN. 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. We will focus on the differences in training times when using CPU and when using GPU : both for the training time of the whole algorithm, and for the training time of the neural network. We will finally try to explain these differences.

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 $\mathcal{X}$ such that $\mathcal{M}$ satisfies the Markov property : the current state only depends on the last state. This is approximately verified when we account for momentum and movement.
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. ([3]), where both the current neural network and the target neural network are used to compute the targets for the regression task. Morally, in DQN, the target is given bu using the target neural network $f_{\theta_{target}}$:

$$y^{DQN} = r + \gamma \max_{b \in \mathcal{A}} Q_k(x', b, \theta_{target})$$

The max operator in standard Q-learning and DQN uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates. To prevent this, we can decouple the selection from the evaluation. This is the idea behind Double Q-learning, where we use both the target neural network $f_{\theta_{target}}$ and the current neural network f_θ to compute the targets :

$$y^{Double-DQN} = r + \gamma Q_k(x', \operatorname*{argmax}_{b \in \mathcal{A}} \{Q_k(x', b, \theta)\}, \theta_{target})$$

This is the algorithm we will implement next.

Other extensions have been proposed, like Bootstrap DQN, proposed by Osband et Al. ([4]). 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. Notably, the best extension is the Asynchronous framework for deep reinforcement learning, proposed by Mnih et Al. in 2016 ([5]), which provides a better performing and memory-efficient way to address the memory problem of DQN and its extensions.

3 Running D-DQN on classic Reinforcement Learning environments

First, we implemented the DQN algorithm on Python. We used Theano in order to construct the neural network, and we made sure that the use of GPU was possible. Then, for all experimentation, we used OpenAI Gym ([6]), a wonderful website which provides ready-to-use classic reinforcement learning environments such as Atari 2600 games 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/Double-DQN_theano_only` and all our results are available on OpenAI Gym : `https://gym.openai.com/users/Caselles` (some results in this page were not obtained specifically with the implementation we describe in this report).

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 network. Hence, the learning is also pretty fast, that allowed us to tune the hyperparameters easily. This is why we chose the CartPole environment to run our comparison of CPU and GPU training times, which is presented in Section 4. 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.

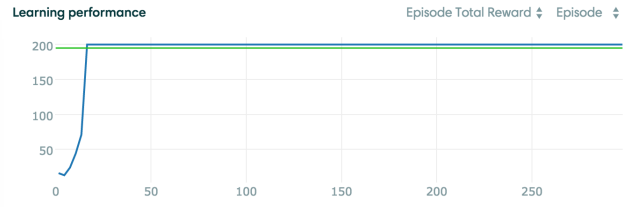

Figure 1: Learning performance of DQN on CartPole.

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

4 Comparing CPU and GPU training times on CartPole

4.1 How did we run the code on GPU ?

The goal of the project was to use the GPU to train the D-DQN algorithm. Luckily, Hugo has a MacBook equipped with a NVIDIA GeForce 750M GPU so we used his GPU to run the code. Since the GPU is NVIDIA, we decided to use CUDA and its library for deep neural networks cuDNN. cuDNN is an NVIDIA library with functionality used by deep neural network. By default, Theano will detect if it can use cuDNN. If so, it will use it. In fact, cuDNN is mostly helpful with convolutional neural networks, and not so much for fully-connected neural networks, but we used it anyway. Finally we configured Theano to run on GPU with CUDA.

For the training, we used CNMeM to allocate a percentage of the memory of the GPU to the training : we used 70% of the total memory. We noticed that if you have too many processes running on your laptop, CNMeM doen't manage to allocate enough memory to the training and you can't use the GPU, because it is already used by the other processes which have a priority over CNMeM.

We hand-crafted the fully-connected neural network with Theano, as requested. We choose to use the following architecture :

- 1 hidden layer with 200 hidden units
- Activation function : RELU
- Loss function : Mean-Squared Error (as seen in Section 2)

We had to manually define the hidden layers, then stack them. For the back-propagation of the error term, we compute the gradient of the loss with respect to the weights of each layer.

4.2 Training time on the whole algorithm

Table 2: Training times on the whole algorithm (5 000 time-steps)

	CPU (2,5 GHz Intel Core i7)	GPU (NVIDIA GeForce GT 750M)
Training time when using Keras and Adam optimizer	$\pm$ 151 seconds	$\pm$ 163 seconds
Training time when using Theano and Stochastic Gradient Descent	$\pm$ 1717 seconds	$\pm$ 1817 seconds

We coded two versions of the algorithm, one where the neural network is defined using Keras and the Adam optimizer which is very fast and efficient. Also, we coded our own version of the neural network using only Theano and its computational graphs, as it was mandatory for the course. The results are worse since we had to use Stochastic Gradient Descent, which is a lot slower and less efficient than other optimizers such as Adam or RMSProp. Also it is worth noticing that the actual reinforcement learning results are also worse with SGD : the learning is much more unstable, it is also incredibly slow and convergence is not acquired every time.

As seen in Table 2, the whole code run faster on CPU than it does on GPU. There are several reasons for that. CPUs are optimized for general tasks. They can handle any task required quite efficiently. On the contrary, GPUs are more specialized, and they are only useful with certain kinds of tasks. For instance, if you can decompose your algorithm into a wide batch of independent operations (such as the backpropagation of a neural network) then a GPU will be able to perform this task very rapidly, and possibly faster than CPUs. Matrix multiplications and kernel convolutions are two examples of this kind of task. However, if your algorithm cannot be broken into many independent calculations (your code has lots of steps that depend on the results of previous steps rather than being independent), then it won't be suited for the GPU. It is clearly the case here : Algorithm 1 shows that there are many dependent steps, such as obtaining the next action via the ϵ-greedy policy, obtaining the batch of inputs and targets the training will be applied to (this calls a method of the ExperienceReplay class, with is composed of many dependent tasks too). There are also a few "if" in the algorithm, which is clearly something we want to avoid when running the code on GPU : it is by definition a dependent mini-task which will slow down the computation on the GPU.

Hence, it is not that surprising to see that the code is faster on CPU than on GPU. In the section 4.4, we will investigate ways to fix that issue.

But all hope is not lost ! There is a major part of our code which is, in theory, particularly suited to run on GPU : the training of the neural network. Let's focus on the differences in training times for this particular task.

4.3 Training time of the neural network

Table 3: Training times of the neural network (5 000 time-steps)

	CPU (2,5 GHz Intel Core i7)	GPU (NVIDIA GeForce GT 750M)
Training time when using Keras and Adam optimizer	$\pm$ 17 seconds	$\pm$ 4 seconds
Training time when using Theano and Stochastic Gradient Descent	$\pm$ 1645 seconds	$\pm$ 1754 seconds

Table 3 confirms us that the training of the neural network is indeed much faster on GPU than on CPU when the code is professionally done (use of Keras) : the speedup is roughly $3\times$ faster. This can explained this the nature of the task : training a neural network consists in a feed-forward of the input into it, then a back-propagation of the error term (here it is the mean-squared error of the prediction and the target y) in order to update the weights of the net. The back-propagation is done by differentiation of the loss with respect to the weights and the chain rule. All this task consists in a bunch of matrix multiplications, which is very fast on GPU.

Unfortunately, our Theano version of the code does not show any speedup when running on GPU... It shows that it has to be improved and we did not know every trick in Theano to optimize GPU performance. We did not have the time to dive into this problem.

4.4 Possible improvements

The code can be improved to be more computationally efficient : it is rarely not the case for non-trivial algorithms. Even though we made a good effort trying to understand the reasons why running on GPU was faster or not, it can still be improved. About the training of the neural network, for the Theano version of our code, one improvement would be to implement Adam or RSMProp : the results should be far better. For the rest of the code, the tasks should be more independent from one another and if statements should be avoided.

5 Conclusion

With this work, we had the chance to fully understand the DQN algorithm, work with neural networks with libraries like Theano and Keras, and study the use of GPU for computational purposes. We were able to reproduce reinforcement learning results on OpenAI gym, and show a case were GPU runs faster than CPU.

If you have any questions or would like to discuss about the project, feel free to contact us.

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

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

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

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