Deep Reinforcement Learning for Superhuman Performance in Doom

Ivan Rodríguez
Deep Reinforcement Learning for Superhuman Performance in Doom

Author: Ivan Rodríguez
1st examiner: Univ.-Prof. Dr. Hans-Joachim Bungartz
2nd examiner: Univ.-Prof. Dr. Thomas Huckle
Assistant advisor(s): M.Sc. Moritz August
Thesis handed in on: July 15, 2017
I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

June 24, 2017

Ivan Rodríguez
Abstract

Over the last years, Reinforcement Learning (RL) has attracted the attention of many researchers. Its powerful combination with Artificial Deep Neural Networks, when used as function approximators, has shown to be successful in many works. Rather than target classical RL problems, the most prominent examples of these works develop techniques that allow agents to learn how to play video and board games from raw input data at human level. In this thesis, we describe the implementation of an algorithm to train an agent for the popular 90’s computer game Doom. Doom features several recurrent problems in RL such as delayed rewards and partial observability which are tackled by the algorithm. In particular, we discuss the efforts to improve the efficiency of our approach and the results obtained in several tests scenarios.
# Contents

Abstract vii

Outline of the Thesis xi

I. Introduction and Theory 1

II. Development of a bot for Doom 3

1. Approach 5
   1.1. DFP setting .............................................. 5
   1.2. Model .................................................... 5
   1.3. Training .................................................. 6

2. A more efficient implementation 9
   2.1. GA3C setting ........................................... 9
   2.2. Asynchronous DFP ....................................... 9

III. Results 11

Appendix 15

A. Implementation Details 15

Bibliography 17
Outline of the Thesis

Part I: Introduction and Theory

CHAPTER 1: INTRODUCTION
This chapter presents an overview of the thesis and the motivation behind it.

CHAPTER 2: CLASSIC REINFORCEMENT LEARNING
We give here the fundamental elements to describe Reinforcement Learning (RL) problems and the abstractions used to solve them. Afterwards, we present the properties of different RL methods which mainly fall in two categories: Tabular and Approximation methods.

CHAPTER 3: DEEP REINFORCEMENT LEARNING
In this chapter, we discuss how Deep Neural Networks come into play for RL problems. Particularly, we present simulation environments for video games (including ViZDoom for Doom) and approaches built around them.

Part II: Development of a bot for Doom

CHAPTER 4: APPROACH
This chapter presents the approach we based our work on and how it tackles the challenges posed by Doom.

CHAPTER 5: A MORE EFFICIENT IMPLEMENTATION
We present here the improvements built on top of the original approach.

Part III: Results

CHAPTER 6: EXPERIMENTS
Four scenarios with increasing difficulty were considered for experimentation. The performance in training and evaluation time of several artificial agents are shown in this chapter.

Part IV: Conclusion

CHAPTER 7: SUMMARY AND OUTLOOK
In this chapter, we present our conclusions.
Part I.

Introduction and Theory
Part II.

Development of a bot for Doom
1. Approach

We followed the approach taken by [2], the winners of the Visual AI Doom 2016 competition for the Full Deathmatch track (with unknown maps and more than one weapon available). In this chapter, their algorithm, called DFP (Direct Future Prediction) is described in detail.

1.1. DFP setting

An artificial Doom player is spawned in an unknown map (environment) with a set of actions $A$. The interaction with the environment is carried out over discrete timesteps $t = 0, 1, 2, ...$ in the form of episodes that end with the death of the player or when a maximum number of steps is reached. At each timestep, the player receives an observation $o_t$ composed of an input image $s_t$ and a vector of measurements $m_t$. Depending on the observation, an agent performs an action $a_t \in A$ and, as a consequence, its measurements are affected. The objective is, thus, to choose actions in such a way that measurement values are maximized during an episode.

For discrete temporal offsets $\tau_1, \tau_2, ..., \tau_n$, the vector $f$ contains the difference between future and current measurements and is defined as $[m_t+\tau_1-m_t, m_t+\tau_2-m_t, ..., m_t+\tau_n-m_t]$. In addition, a maximization objective for the measurements is assumed to take the form

$$u(f; g) = g^\top \cdot f \quad (1.1)$$

where $g$, the goal vector, is a parametrization vector with the same size as $f$, specified at the beginning of the training, but that can be changed during test time. This representation allows us to define in which proportion particular future measurements are more important than others.

1.2. Model

To predict future measurements, a Deep Neural Network (DNN) with parameters vector $\theta$ is used. The network takes an image $s_t$, a vector of measurements $m_t$, and a goal vector $g$ as inputs (Figure 1.1). The inputs are then processed with a convolutional network and several stacked fully connected layers. Next, the results are concatenated and split into two streams, Expectation and Advantage. The former calculates the average of future measurements according to the current observation, while the latter makes an estimation of the advantage of taking a particular action over all the other possible actions.

In the Advantage stream, the Normalize operation is carried out as in Equation ??}. Finally, both streams are added and a prediction is obtained. The prediction is thus defined as
1. Approach

Figure 1.1.: DFP neural network

\[ P_t = F(m_t, s_t, a, g; \theta), \quad a \in A \]  \hspace{1cm} (1.2)

where the function \( F \) represents the computation by the DNN. After doing a prediction on the model at timestep \( t \), we choose an action that maximizes our objective function (Equation 1.1) with respect to the specified goal vector \( g \):

\[ a_t = \arg \max_a g \cdot P_t \]  \hspace{1cm} (1.3)

1.3. Training

The agent starts interacting with the environment according to a \( \epsilon \)-greedy policy (Section ??). Therefore, at the beginning of the training the value of \( \epsilon \) is set to 1.0 (random actions) and is gradually decreased down to 0.1, at the end of the training. This decrease allows the agent to continue exploring in a smaller proportion even when the model already contains information of the environment.

Similar to DQN (Section ??), DFP uses a replay memory to store experiences, which helps to increase the stability of the algorithm by breaking the correlation between sequences. Experiences are represented by tuples \( (m_i, s_i, a_i, g, f_i) \) and sampled randomly in minibatches of size \( N \) every \( k \) steps of the game. When the replay memory reaches its maximum capacity, the oldest experiences are replaced by new ones.

Having a minibatch of experiences, the DNN is trained to minimize the loss function
1.3. Training

\[ L(\theta) = \sum_{i=1}^{N} \left| \left| F(m_t, s_t, a_t, g; \theta) - f_t \right| \right|^2 \]  

(1.4)

which corresponds to a mean square error, i.e. the error of predicting differences between current and future measurements. It is important to highlight that in Equation 1.4 the action performed is used to update only its corresponding part in the prediction function (Figure 1.2).

In a typical RL problem, agents are trained with experiences collected during training, without using any dataset. To collect those experiences, agents must be repeatedly interact with the environment, adding a significant overhead to the overall training time. In order to decrease this overhead, [2] used 8 agents gathering experiences in parallel that synchronously perform actions and do predictions to the model in batches (Figure 1.3). Although this technique helps effectively to reduce the overhead, we describe in the next section a more efficient implementation based on asynchronous updates to the model.

Figure 1.2.: Target and prediction in loss function

Figure 1.3.: Scheme of DFP implementation with multiple agents
1. Approach
2. A more efficient implementation

As mentioned in the last chapter, [2] implemented an algorithm in which several agents perform synchronous updates to the model. We describe below a modification of this strategy, by allowing agents to run asynchronously, as proposed in the work of [1] to speed up A3C (Section ??).

2.1. GA3C setting

In GA3C (GPU-based Asynchronous Advantage Actor-Critic) [1], multiple agents run in parallel asynchronously without sharing any global network parameters, as opposed to the original A3C implementation [3]. Instead, a Server instance receive minibatches to train the model and predict actions for every agent, thus being the only process allowed to communicate with the GPU. Two types of Server threads, Trainers and Predictors, manage two asynchronous queues to manage the data between the Server and the agents. In the first queue (training queue), agents send their minibatches of experiences; in the second queue (prediction queue), predictor threads receive request for predictions to be performed with the model.

In order to use efficiently GPU bandwidth and keep GPU utilization high, a balanced combination of number of agents, predictor and trainers is desired. However, every combination affects the convergence of the algorithm as well, so a trade-off must be found. To that end, [1] designed a Dynamic Adjustment thread which tries different configurations systematically for improving the number of trainings per second TPS, directly related to the number of predictions per second PPS. Since agents perform one update every \( t_{\text{max}} \) steps in the A3C setting, it is expected to have \( PPS \approx TPS \times t_{\text{max}} \), which allows to find a fixed optimal configuration during the entire training by maximizing the TPS metric.

2.2. Asynchronous DFP

We included GA3C’s asynchronous communication scheme for training and predicting in our original approach (Figure 2.1). In our case, to achieve an optimal combination becomes problematic since \( PPS \) is not constant during training. When agents start interacting with the environment, no prediction queries are issued and actions are chosen randomly. As the interaction advances, \( PPS \) increases proportionally to the decrease of \( \epsilon \) (exploration/exploitation rate). Consequently, a fixed configuration of agents, predictors and trainers does not lead necessarily to the fastest solution.
2. A more efficient implementation

In the next chapter, we discuss how we get to an optimal configuration for DFP and present the results of the modified approach.

Figure 2.1.: Implementation of an asynchronous version of DFP
Part III.

Results
Appendix
A. Implementation Details

Here come the details that are not supposed to be in the regular text.
Bibliography

