Deep Reinforcement Learning agents playing DOOM

Hurault Samuel
Ecole Normale Superieure Paris-Saclay
samuel.hurault@ens-paris-saclay.fr

Kashtanova Victoriya
Ecole National des Ponts et Chaussees
victoriya.kashtanova@eleves.enpc.fr

Abstract

We propose to compare two deep Reinforcement Learning frameworks which learn sensorimotor control in an immersive three-dimensional environment. The first method, called Direct Future Prediction (DFP) reformulates the reinforcement learning (RL) problem in a supervised learning (SL) problem. The second one, called Arnold, renews the standard Q-learning methods. Both models where initially trained on the classical game Doom and obtained impressive results at the VizDoom AI Competition. We reproduced the main results from both articles and then improved the DFP model with ideas from the Arnold model.

1 Introduction

After the victory of the AlphaGo program over the human professional Go player in October 2015 ([11]), many researchers started to develop new Reinforcement Learning models capable of outperforming humans.

Our project deals with sensorimotor control with first-person perspective view in immersive semi-realistic 3D environments. Examples of simulated environments are VizDOOM [6], CARLA [6] (autonomous driving) or MINOS [10] (indoor navigation). Our work will be exclusively tested on the VizDOOM platform.

Standart RL approach, and particularly Deep Q-Learning Networks (DQN) [9], have been first widely used in 2D games as it takes the visual game state directly as input and learns a mapping to action. However 3D domains exhibit a multitude of new challenges compared to 2D such as partial observability or extra need for exploration of the environment causing this standard approach to struggle. Researches attempted to optimize DQN. For example, DQN-SLAM [1] uses object categorization and localization with SLAM.

We are going to analyze, optimize and compare the performances of two models that won the 2016 VizDOOM AI Competition : "Arnold" developed by Lample & Chaplot [7] and Direct Future Prediction (DFP) developed by Dosovitskiy & Koltun [2]. The first model also optimizes DQN bringing recurrency with LSTM, and co-training it with supervision on game features provided by the game engine. The second article develops a new approach since it predicts future measurements through a supervised learning scheme. After replicating the main results from both articles, we are going to show that it is possible to optimize the DFP network, thanks to ideas given by the first article and by other previous studies on the DFP model ([8],[4]).

We used the author's implementations of DFP [1] and Arnold [2] as baseline for our experiments. You can find in our github [3] our renewed code for DFP as well as videos of the agents for both methods.

2 Arnold model

2.1 Model

After a brief summary of the DQN and DRQN models, we describe the Arnold model given in the article, and then replicate some of its main results.

2.1.1 DQN

The Deep Q-Network (DQN) is a deep neural network Q_θ that estimates the Q-function of the current policy which will be the closest to the optimal Q-function Q^*.

At each step, we select the action a_t (with ϵ-greedy style) that maximizes the current Q-function estimation and then we update this estimation by the network.

Experience replay is also used to break correlation between successive samples. In practice, at each time steps, agent experiences (s_t, a_t, r_t, s_{t+1}) are stored in a replay memory, and the Q-learning updates are done on batches of experiences randomly sampled from the memory.

2.1.2 DRQN

The Deep Recurrent Q-Network (DRQN) was created by Hausknecht and Stone [5] for partially observable environments (in particular in 3D). DQN has bad performances in this case, because the agent does not receive a full observation of the environment. In this model, the state s_t is only described by the image of the game o_t. DQRN does not estimate $Q(o_t, a_t)$, but $Q(o_t, h_{t-1}, a_t)$, where h_{t-1} is an extra input returned by the network at the previous step. It is done thanks to a LSTM layer $h_t = LSTM(h_{t-1}, o_t)$.

All in all, in the initial DRQN model, every frames are treated by a convolutional neural network (CNN), itself feeding a LSTM that predicts a score for each action based on the current Q-function estimation.

2.1.3 Co-training with game features

The ViZDoom environment gives access to internal variables generated by the game engine. It can return, with every frame, information about the visible entities like health packs, ammo, poison ... At each step, we receive binary value for each entity, indicating whether this entity appears in the frame or not. This information is not available at test time, it can be only exploited during training.

The authors completed the DQRN model to make it sensitive to these game features. They added two fully-connected layers to the output of the CNN that detect the presence of game features. The architecture of the model is illustrated in 1. Jointly training the DRQN model and the game feature detection should boost performances as it allows the convolutional layers to capture the relevant information from the frame.

2.1.4 Action / navigation split

The authors also decided to split the navigation (exploring the map, collecting the items and finding the enemies) and action (fighting enemies) tasks for two different networks. They use the DRQN augmented with game features for action and a simple DQN for navigation. At each step, both networks act jointly. The navigation network is called if no enemy is detected, or if the agent does not have any ammo left.

[1] `https://github.com/IntelVCL/DirectFuturePrediction`
[2] `https://github.com/glample/Arnold`
[3] `https://github.com/samuro95/MVA-project-`

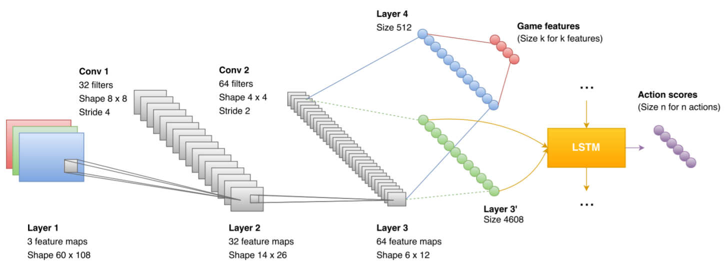


Figure 1: DRQN model with game features co-training.

2.2 Experiments

We use the [7] author's implementation [4] and we experiment on two scenarios described in table 1: Health Gathering and Deathmatch.

Table 1: Scenarios description

NAME	**Health gathering**	**Deathmatch**
NETWORK	DQN	DRQN + DQN
REWARDS	+: picking up health pack -: loosing health	+: killing , picking up object - : loosing health, shooting
AVAILABLE ACTIONS	TURN RIGHT, TURN LEFT, MOVE FORWARD	TURN RIGHT, TURN LEFT, MOVE RIGHT, MOVE LEFT, MOVE FORWARD, MOVE BACKWARD, ATTACK, SPEED, WEAPON SELECTION
TOTAL NB OF ACTIONS	8	53
GAME FEATURES	Health packs	Enemy

The **health gathering** scenario focuses on navigation, the agent is in a maze and its health is declining at a constant rate. It must collect health packs but also avoid poison. The performance evaluated is the average life time of the agent in a game episode (number of steps / episodes).

For the **Deathmatch** scenario. The agent is in battle against built-in Doom bots. The performance metric is K/D : the number of kills divided by the number of times the agent dies.

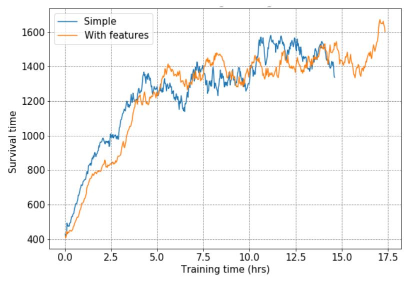
Training Configurations :

- All networks were trained using the RMSProp algorithm and batches of size 32.
- The input frames are in color and 60×108.
- To accelerate training, $frame\_skip = 4$. Only one frame out of four is processed, and the same action is kept during the four consecutive steps.
- Network weights were updated every 4 steps and experiences are sampled on average 8 times during training.
- The replay memory for DQN contains the 100 000 most recent frames.
- The different learning were realized on a Google Cloud Platform Virtual Machine with a NVIDIA Tesla K80 GPU.

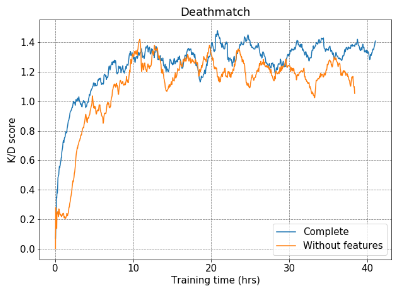
[4] `https://github.com/glample/Arnold`

Game features network augmentation :

In figure 2, we plot the evolution of the learnings on both scenarios with and without game features network enhancement. As already shown in the article [7], the method benefits on the Deathmatch scenario but not on "health gathering". Indeed, with experience replay, DQN takes k frame as inputs, it does not make sense to predict features working only on the last frame. On the opposite, the DQRN model with its LSTM layer takes only one frame as input. It is much more adapted to the game feature detection co-training.



(a) Health gathering



(b) Deathmatch

Figure 2: Performances on both scenarios during training of the models learned with and without game features augmentation.

However the improvement is not as significant as what was found in [7]. We did not find any reason explaining the difference between our and their results.

3 Direct Future Prediction (DFP)

3.1 Introduction

The DFP agent takes an observation (a high dimensional raw sensory input and a low dimensional input measurement), learns how that measurement changes in future time-steps for each possible action, and determines the action that yields the best changes to that measurement based on some given "objective". The objective is formalized by a goal vector that dynamically defines the relative importance of each predicted measurement.

We will see that the main advantage of the method comes from the fact that the model does not use the standart RL scalar reward. Instead, the multi-dimensional measurement stream provides a rich and dense supervision. The RL problem becomes a supervised learning method and learning is realized through supervision provided by experience. The agent acts and observes the effects of different actions in the context of changing sensory inputs and goals. We are going first to present the DFP model developed in [2] before replicating some experiments from the article and finally showing how we improved the given model.

3.2 Architecture of the model

At each time step t, the agent receives an observation o_t and executes an action $a \in A = \{a_1, ..., a_n\}$ based on this observation. The observations is $o_t = (s_t, m_t)$, where s_t is an image and m_t is a set of measurements (like Ammo level, Health, or number of Kills). To predict future measurements, we use a parameterized function approximator, denoted by $F(o_t, a, g; \theta)$, where g is the goal vector in use and θ the learned parameters. The action taken a_t follows :

$$a_t = \underset{a \in A}{\operatorname{argmax}}\, g^T F(o_t, a, g; \theta) \tag{1}$$

The predictor F is a deep network illustrated in figure 3.

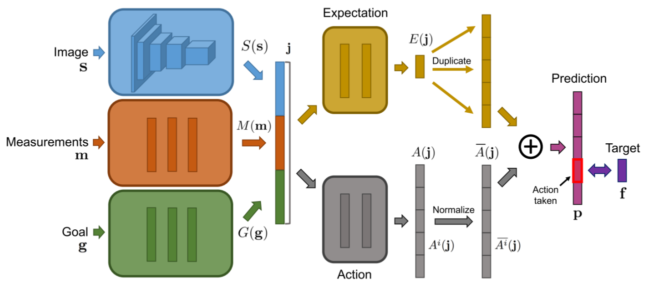


Figure 3: The Network structure

Note that the current goal vector g is also an input of the network. Indeed, in order to allow the predictor to generalize to a variety of behaviors, the predictor is conditioned on the goal vector. Intuitively, instead of predicting "in 4 steps I will have 50 health points", the model predicts "if I act according to goal vector (1,0,0), then in 4 steps I will have 50 health points".

The network has three input modules : a perception module S, which is convolutional, a measurement module M and a goal module G, which are fully-connected. The outputs of these three neural networks are concatenated and sent to an expectation NN and an action NN. The expectation network predicts the average future measurements over possible actions. The action network outputs the fine differences in those measurements among possible actions. The results of these two network are then finally added together to create the prediction vector.

Training is realized thanks to experiences collected by the agent. At the beginning the experience is created with a random policy but when the training begins, we collect new ones. At each step, we keep a batch D of training samples $D = \{(oi, ai, gi, fi)\}_{i=1}^{batch\_size}$ where (o_i, a_i, g_i) are the inputs and f_i is the output of example i. The predictor is trained using the MSE only for the specific action we have taken during training, not for the entire prediction vector :

$$L(\theta) = \sum_{i=1}^{batch\_size} ||F(o_i, a_i, g_i; \theta) - f_i||^2 \tag{2}$$

3.3 Experiments

We use the [2] author's implementation [5] and we experiment on two of their scenarios, illustrated in table 2.

The **health gathering** is the same as the one described in 2.2

For the **battle** scenario. The agent is again is a complex maze, he is armed and under attack by alien monsters. The monsters move and shoot at the agent. Health packs and ammunition are randomly distributed throughout the environment. As this agent is supposed to be tested on other maps, the textures of the scenario are randomized to create 100 different maps. The first 90 are used for training and the last 10 for testing.

During training and testing, we ask the agent to execute a total number of steps, independently of the number of episodes played. An episode is a session of the game, limited in time. The agent begins with qn episode with 100 health level, and 20 ammo level. The episodes ends when the agent is dead, when the $episode\_timeout$ is reached, or when it is the end of the total number of steps.

Training Configurations

- The predicted future steps are steps $t + \tau$ with $\tau = \{1, 2, 4, 8, 16, 32\}$ Only the latest three time steps contribute to the objective function, with coefficients $(0.5, 0.5, 1)$.

[5] `https://github.com/IntelVCL/DirectFuturePrediction`

Table 2: Scenarios description

NAME	Health gathering	Battle
AVAILABLE ACTIONS	TURN RIGHT, TURN LEFT, MOVE FORWARD	TURN RIGHT, TURN LEFT, MOVE RIGHT, MOVE LEFT, MOVE FORWARD, MOVE BACKWARD, ATTACK, SPEED
TOTAL NB OF ACTIONS	8	256
MEASUREMENTS	Health	Ammo, Health, Kills

- The input frames are gray-scale and 84×84.
- Batches of $batch\_size = 64$ are sampled from an experience memory of $20{,}000$ steps. Training is realized for 820000 steps. It makes a total number of $820000 * 64 = 52{,}48$ million steps.
- To accelerate training, $frame\_skip = 4$. Only one frame out of four is processed, and the same action is kept during the four consecutive steps.
- The different learning were realized on a Google Cloud Platform Virtual Machine with a NVIDIA Tesla P4 GPU. It takes around 23 hours to process for the battle scenario.

3.3.1 Health Gathering

Initial configurations :

The article limits the episodes to $episode\_timeout = 525$ steps an study the health level at the end of an episode. In figure 4, we display the evolution of our and their learnings. As expected, we obtain very similar performances.

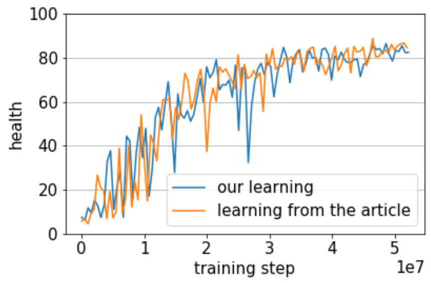


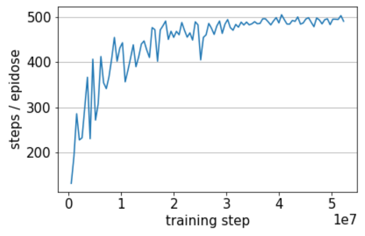
Figure 4: Average health at the end of an episode for the learning we realized and the learning given in [2] in the same conditions, on the health gathering environment.

However, even if limited episodes enable to accelerate learning, the original objective of the health gathering game is not to die with as much health as possible but rather to live as long as possible. We display in figure 5 a) the average life time of the agent with these initial settings. We notice that after 3.5 million steps, the agent lives almost for the entire episode. After this, the average final health in 4 does not increase neither.

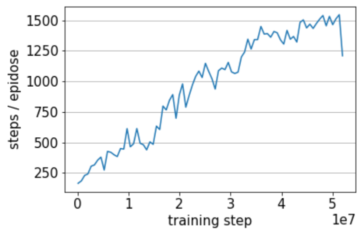
Training on longer episodes : .

Therefore it makes sense to try to learn with a longer $episode\_timeout$. In figure 5 b), we display the same evolution with an episode limit four times bigger (2100 steps). The $episode\_timeout$ is almost never reached by the agent, and the performance keep increasing during learning.

In table 3, we test the two learning in the same configurations. We observe that the agent trained on short episodes does not perform well on long episodes. We explained this by the fact that the agent was almost always trained with a high health level and it did not learn to react wisely with



(a) $episode\_timeout = 525$ steps



(b) $episode\_timeout = 2100$ steps

Figure 5: Average number of steps per episodes during training on the health Gathering scenario for different $episode\_timeout$ values.

health close to zero. For example, we can imagine that, if health is very low at a certain time-step, the neurons related to presence of health packs should react intensively in the prediction network.

Table 3: Average number of steps per episodes for different $episode\_timeout$ values.

Testing \ Training	short episodes	long episodes
short episodes	517	509
long episodes	658	**1166**

3.3.2 Battle scenario

Initial configurations :

We first replicate the results from [2] in figure 6. In this article, they study the number of enemies killed by the agent during the episodes. The goal vector in fixed at $g = (0.5, 0.5, 1.)$ during learning and testing. Therefore, the goal module G is not used.

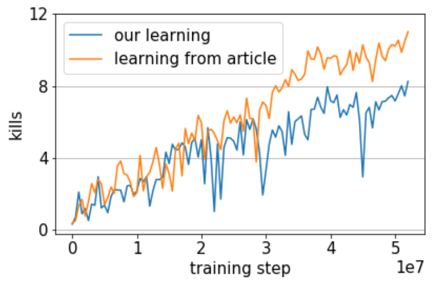


Figure 6: Battle original learning. Average number of kills by episode for the learning we realized and the learning given in [2] in the same conditions.

Surprisingly, our learning did not achieve the same performances. As the article did not mention any $episode\_timeout$ correction, we limited again out episodes to $episode\_timeout = 525$ steps. We suspect that may not have used this limit.

Training on longer episodes : .

We tried again to execute learning with $episode\_timeout = 2100$ and show the comparison with the initial training in table 4. We do the same observation : the agent trained on short episodes does not perform well on longer episodes.

Table 4: Average number of kills per episodes for learnings with different $episode\_timeout$ values.

Training / Testing	short episodes	long episodes
long episodes	9.8	**15.9**

The number of kills obtained in this test is not as good as the number obtained on the training curves because testing is realized on different maps, not explored during training.

Goal agnostic training : .

As realized in [2], we evaluate the ability of the model to learn without a fixed goal. We use two training regimes : first, like previously a fixed goal vector $(0.5, 0.5, 1)$ and, second, a random goal vector with each value sampled uniformly in $[-1, 1]$. In the second regime, the agent has no knowledge of the relative importance of each measurement, it even does not know if they are desirable or not. We expect this second regime to enable generalization to various testing goal vector inputs. We choose to randomize in $[-1, 1]$ and not in $[0, 1]$ even if we already know that the measures Ammo, Health and Kills are desirable because we want to make this agent comparable to an agent potentially trained with other non desirable measurements. Moreover, in [2] the randomization in $[-1, 1]$ and $[0, 1]$ obtained similar performances.

Table 5: Average Kill count for varying input goal vectors.

Training / Testing	$(0.5, 0.5, 1)$	Random goal in $[-1, 1]$
$(0.5, 0.5, 1)$	**15.9**	13.5
$(1, 1, 1)$	15.2	**14.7**
$(0, 0, 1)$	1.6	2.4

Results in table 5 show that without knowing the goal during training, the agent performs nearly as well as when it was specifically trained for it. However, the random model does not generalize really effectively to other input goals. In the following study, we will keep the model trained with random goal inputs. We note that, to maximize frags, if we want to keep the goal vector fixed, the best choice of input goal vector is (1., 1., 1.). Strengthened by this learning, a straightforward idea would be to dynamically change the input goal vector according to the current measurement. Unfortunately, we did have time to implement it.

3.4 Using Game features

As explained in 2.1.3, we can use the ViZDoom environment ability to provide information about the visible entities. At each step, we receive a binary value for each entity, indicating whether this entity appears in the frame or not.

3.4.1 First method

Our first idea was to add these binary variables to the measurement stream. We concatenate to m_t the binary variable representing the presence or not of the k chosen features to acquire. However we do not want to predict the values of these variables like it was done for the other measurements. Therefore we change the network so that it only predicts the future of the initial measurements m_t.

3.4.2 Second method

However, if we want to stand to the rules of the Vizdoom competition, the game features are only available for training, not for testing. We therefore considered the co-training method developed in [7] and previously detailed in 2.1.3. A multi-label classification network constituted of two fully-connected layers are added to the output of the image module S. It predicts the presence of the game features on the given image. At training time, the cost of the network is a combination of the normal cost and the cross-entropy related to this classification. The new architecture is illustrated in 7.

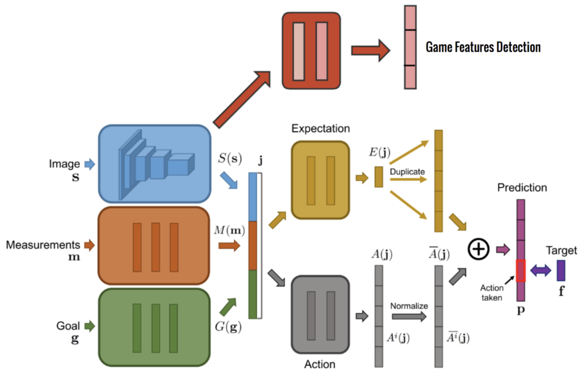
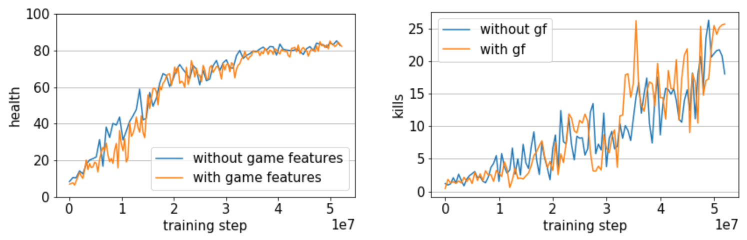


Figure 7: The new network structure

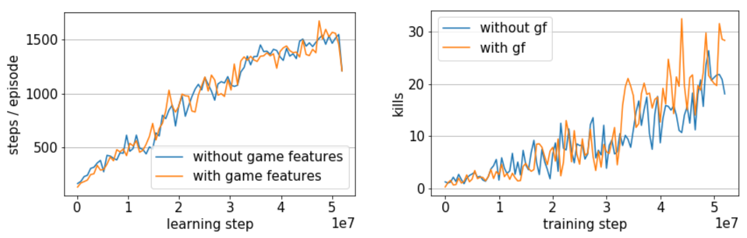
3.4.3 Experiments

The features considered are health packs and poison items for "health gathering" and the enemies for "battle". For both scenarios, the references used for comparison are the training realized with long episodes. We use the "random goal input" learning for battle.



(a) "health gathering" : final health with method 1.

(b) "battle" : number of kills with method 1



(c) "health gathering" : life time with method 2.

(d) "battle" : number of kills with method 2

Figure 8: Performances for both scenarios and both methods.

Figure 8 displays the learnings with and without game feature augmentation (the top left hand graph represents the average health at the end of an episode and not the life time because it was realized, by mistake, on short episodes). We note that both methods did not improve the performances on the "health gathering". However, it seems that they improve the kill count on the battle scenario.

With longer episodes, we did not increase the test time during learning so that the overall learning does not take to long. Therefore the big variation on the curves related to "battle" are explained by the fact that we average a small total number of episodes (around five). Thus, we also give in table 6 the results of a longer and then more reliable test on the battle scenario.

Table 6: Average kill count for the different studied models.

Training	**Testing with goal** $(0.5, 0.5, 1)$
Initial network	13.5
Method 1	**15.6**
Method 2	**15.5**

We observe that both methods boost performances on the battle scenario. However, we keep the second because it uses game features only for training. This co-training method works specially on the battle scenario, because as explained in [7], detecting enemies is critical to the agent's performance and using game features during learning influences the quality of the output of the CNN. It is also interesting to see that both methods perform equally well : giving the ground-truth game features values at every time step (method 1) is as effective as letting the CNN detect them automatically (method 2). This observation is also explained by the fact that the accuracy of the game features classification reaches 90% after 20 M steps.

4 Comparison between methods

We compare the performances between the Arnold agent and our new DFP agent on the same maps, with the same configurations.

We first use the health gathering scenario where, as shown before, both agents were specifically trained. We evaluate on this map the ability of the agent to live as long as possible. In this map, we expect DFP to outperforms Arnold. Indeed, for navigation, Arnold agent is trained with basic DQN and the article [2] argues that DFP method strongly outperforms DQN.

The second comparison scenario is "defend the center". In this scenario, the agent begins at the center of a very basic circular room and is under attacked by bots and monsters. The objective is to kill as much enemies as possible. We use the agents DFP and Arnold trained on their respective battle scenarios. Thus, this map is new for both agents and the monsters have never been encountered before.

Testing on these two scenarios gives an complete knowledge of the performances of the agents.

Table 7: Performance comparison between the two agents. The metrics are, on "health gathering", the average number of steps per episode (life time) and, on "defend the center", the average number of kills per episodes.

Scenario / **Agent**	Health gathering	Defend the center
DFP	**4664**	**8.9**
Arnold	2058	8.6

Results are shown table 7. Note that the DFP agent life time on "health gathering" is multiplied by 4 compared to what given before. It is because the frame-skip is here not taken into account in the step count.

The results are similar on "battle" but the DFP agent outperforms Arnold on "health gathering".

We can observe on "health gathering" videos [6] that, at the beginning of the episode, both agents perform well in navigating and collecting health packs. When the agent is facing a wall, it turns on itself, until it sees health packs and then runs towards them. However, when the Arnold agent

[6] `https://github.com/samuro95/MVA-project-`

can not see any packs in his field of view, it keeps turning on itself and does not try to explore the map deeper. DFP succeeds in learning to the agent to explore when not finding health packs. We explain such ability by the fact that by predicting the health level in 16 or 32 steps, the agent learned that, at time t, with low health level and no detection of health packs in his field of view, it has to move elsewhere on the map.

All in all, the main advantage of the DFP network compared to DQRN, is to use a multi-dimensional and dense feedback on measurements rather than an immediate scalar and sparse reward.

5 Conclusion and Discussion

We presented and trained two different models dealing with sensorimotor control in the Doom 3D immersive environment. After having replicated the main results from both articles, we showed on two scenarios that a supervised learning formulation of the problem without reward but with feedback on measurements outperforms a state-of-the art Q-learning method. Thanks to an idea taken from the Arnold network, the DFP network was upgraded for a better visual understanding of the environment through game features. What is more, we showed that the learning was improved by extending episode time limits. We think that the DFP agent can still improved, particularly by dynamically adapting the game vector or by applying other ideas from [7] like the network split on navigation and action, the continuous action spaces, and the use of memory with the LSTM. We also trusted a large part of the parameters chosen by the original articles [2] and [7] and did not try to optimize them. Finally, we did not have time to apply our agents to an other environment like CARLA [3] (autonomous driving) and MINOS [10] (indoor navigation), because of the complexity of using them with headless virtual machines. Nonetheless, it would be very interesting to discover the performances on such simulators that are closer to realistic robotic research.

References

[1] Shehroze Bhatti et al. “Playing Doom with SLAM-Augmented Deep Reinforcement Learning”. In: *CoRR* abs/1612.00380 (2016).

[2] Alexey Dosovitskiy and Vladlen Koltun. “Learning to Act by Predicting the Future”. In: *CoRR* abs/1611.01779 (2016). arXiv: `1611.01779`. URL: `http://arxiv.org/abs/1611.01779`.

[3] Alexey Dosovitskiy et al. “CARLA: An Open Urban Driving Simulator”. In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16.

[4] Felix Yu. *Direct Future Prediction - Supervised Learning for Reinforcement Learning*. 2017. URL: `https://flyyufelix.github.io/2017/11/17/direct-future-prediction.html`.

[5] Matthew J. Hausknecht and Peter Stone. “Deep Recurrent Q-Learning for Partially Observable MDPs”. In: *AAAI Fall Symposia*. 2015.

[6] Michal Kempka et al. “ViZDoom: A Doom-based AI research platform for visual reinforcement learning”. In: *2016 IEEE Conference on Computational Intelligence and Games (CIG)* (2016), pp. 1–8.

[7] Guillaume Lample and Devendra Singh Chaplot. “Playing FPS Games with Deep Reinforcement Learning.” In: *Proceedings of AAAI*. 2017.

[8] Michael Aboody, Jongmin Jerome Baek, Yu-chieh Lee. *The Reimplementation and Application of Direct Future Prediction to Simple Environments and Online Learning*. 2017. URL: `https://www.ocf.berkeley.edu/~jjbaek/dfp_paper.pdf`.

[9] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: `http://dx.doi.org/10.1038/nature14236`.

[10] Manolis Savva et al. “MINOS: Multimodal Indoor Simulator for Navigation in Complex Environments”. In: *arXiv:1712.03931* (2017).

[11] Wikipedia contributors. *DeepMind*. 2019. URL: `https://en.wikipedia.org/wiki/DeepMind`.