Implementation
Playing FPS Games with Deep Reinforcement Learning

Titouan Christophe[1], Florentin Hennecker[2], Nikita Marchant[1] and Bruno Rocha Pereira[1]

[1]Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
[2]Université Libre de Bruxelles, Boulevard du Triomphe - CP 212, 1050 Brussels, Belgium
tichrist@vub.be, fhenneck@ulb.ac.be, {nimarcha,brochape}@vub.ac.be

Abstract

In this article, we study a Deep Recurrent Q-Learning Network implementation playing the DOOM video game. Our findings are based on a publication from Lample and Chaplot (2016). We first present deep Q-learning under two variants (DQN and DRQN) which were applied to video games. We then present how we built our own implementation of a testbed for such algorithms. Finally, we present our results on a simplified game scenario.

Introduction

Deep Q-Learning is a technique which combines the Q-Learning algorithm with a deep neural network. In this particular architecture, the expected return of each action is predicted by a neural net. The agent using this algorithm then chooses the best action based on a policy (we only used ϵ-greedy here). Neural networks, particularly convolutional neural networks, are very good at encoding a highly dimensional state (in this case, a 108x60 RGB image) into a meaningful lower dimensional state. Moreover, a recurrent neural network can help handle the sequential nature of playing a video game by passing a hidden state from one frame to the next.

In order to plug our work to the DOOM game, we are using the ViZDoom environment (Kempka et al., 2016) that provides a programming interface to interact with a DOOM game in various languages (we use Python). ViZDoom provides access to the screen buffer and actions buttons, as well as game metadata (such as player position and the labeled objects in the field of view)[1].

Our agent plays the basic scenario, where he is in a rectangular room. At the beginning, the enemy spawns at a random position on the opposite side of the room. The game ends if the agent kills the enemy, or if 500 game ticks elapsed.

Method

Background

We will give in the paragraphs hereinbelow a state of the art of the technologies that we used in our implementation. We first present the Deep Q-Learning Network from DeepMind, then its variant Deep Q-Learning Recurrent Network.

Deep Q-Networks Many reinforcement learning problems have a very high-dimensional space-state in which each dimension on its own bears very little information. Approximating a state-action Q function over all possible states and actions becomes quickly intractable. However, deep neural networks have an outstanding capability to encode high-dimensional states into a meaningful lower-dimensional hidden state; moreover, they are also known for their function approximation capabilities (Hornik, 1991). Deep Q-Learning (Mnih et al., 2015) is a major breakthrough showing that neural networks can learn to play Atari games.

In the context of a Deep Q-Network (DQN), an agent tries to learn $Q(s_t, a_t)$ which represents an estimation of how valuable an action a_t is in a state s_t. In order to find Q^*, the optimal Q-function, DQNs use a neural network parametrized by θ, which represents the weights of the neural network and gives an approximation called Q_θ further in this paper. Their goal is to find the optimal and real Q-function Q^* which is verifies the Bellman optimality equation:

$$Q^*(s,a) = E\{r + \gamma \max_{a'} Q^*(s',a')|s,a\}$$

With r being the reward of playing a in a state s, $\gamma \in [0,1]$ the discount factor and (s', a') being the next state and its associated action

This gives, since $Q_\theta \approx Q^*(s,a)$ the loss function below:

$$L_\theta(\theta_t) = E_{s,a,r,s'}\{(r+\gamma \max_{a'} Q_\theta(\theta_t)(s',a') - Q_{\theta_t}(s,a))^2\}$$

This loss is computed on uniformly sampled game transitions stored in a replay memory instead of the previously played transition to reduce correlation between samples. Algorithm 1 shows how a Deep Q-Network is trained.

Deep Recurrent Q-Networks One of the issues posed by the model described above is that it assumes that the environment is fully observable. However in the case of DOOM,

[1]The metadata were used only during training time

Algorithm 1 The DQN training algorithm

```
for number of episodes do
    create new episode
    while episode not finished do
        if random() $\leq$ epsilon then
            choose random action
        else
            choose action with highest Q
        end if
        store old state, action, new state and reward in memory
    end while
    for number of back-propagation steps do
        update Q-network with a batch of randomly sampled sequences
    end for
end for
```

we only get a partial observation of the environment, limited by the field of vision of the main character. Previous ways of solving that issue have been to feed a sequence of frames to the DQN as a history. A potentially better way to solve the history issue is to use a Deep Recurrent Q-Network, as described in (Hausknecht and Stone, 2015). A DRQN estimates $Q(s_t, h_{t-1}, a_t)$ instead of the regular $Q(s_t, a_t)$. The parameter h_{t-1} represents the hidden state of the agent at the previous transition. In our case, we will use a Long Short-Term Memory layer (LSTM) (Hochreiter and Schmidhuber, 1997). In addition to the input and output state of usual neural networks, a LSTM unit also carries a hidden state, which has to be fed along with the input, and retrieved with the output. This hidden state is able to convey information about patterns that evolve along time. In our case, it is expected to carry information on portions of the game that have been observed but which are no more visible.

Model details

Our neural network, as presented in Figure 1 is the same as the one presented in (Lample and Chaplot, 2016), with one addition. It is composed of two convolutional layers followed by three separate sub-networks. One of these sub-networks is used in a supervised manner to predict various game features (see below), the second is a standard DRQN network while the last is a standard DQN output. The network is trained with or without the game features and with either the DRQN or the DQN output. The hidden size of the LSTM was set to 300 units.

Game Features As described in the paper (Lample and Chaplot, 2016), agents not using the game features were unable to accurately detect enemies and were thus randomly shooting around hoping that those shots would reach their targets. In order to make up for a lack of efficiency, we used custom made game features to train the agent in a supervised setup jointly with the Q-learning setup to tell the agent if enemies were in his field of vision. The `Label buffer` provided by the Vizdoom API was used to fetch all the entities present in the vision frustum. However, entities hidden by other scene elements, typically walls, were also parts of that buffer. We then used the map characteristics in order to check if an entity was visible or not. Even though this information can't be used during testing phase, it was used during the training of the agents to accelerate training. We also created a game feature estimating the lateral position of an enemy against a wall in a much simpler DOOM map.

Dropout Dropout (Srivastava et al., 2014) was added to the fully connected layers, with a probability of keeping each unit of 75% at training time. This neural network engineering technique prevents from overfitting.

Frame Skip To reduce the computational complexity, it is common to use the skip-frame technique. With this technique, the agent receives a input frame every $k + 1$ game frames, where k is the number of of frames skipped. During those skipped frames, the last action of the agent is repeated automatically. We used a value of $k = 4$ because it is a good trade-off between computing performance and agent accuracy (Kempka et al., 2016).

Sequential Updates When training an LSTM, one feeds sequences of data for which the LSTM will predict one output per time step. However, the LSTM will zero its hidden state before the start of each sequence, and it will take several time steps for the hidden state to initialize. Hence, the outputs for the first time steps might be erroneous since their previous hidden state was not fully initialized. To solve this, we discard the first time steps outputs when computing the loss, as explained in (Lample and Chaplot, 2016).

Hyper-parameters Analogously to Mnih et al. (2015) and Lample and Chaplot (2016), we use a replay memory size of 1000000 frames, and mini batches of 32 samples. The sequence length of the training samples was set to 8, but only the last 4 outputs were accounted in the loss. The optimizers used to train the network use a learning rate of 0.001 (we use RMSProp (Hinton et al., 2012)), and the discount factor (γ) is set to 0.99. Each frame consists of a 8 bit per color `RGB` buffer of dimension $108 * 60$[2]. Our test program consists of 3 phases:

Bootstrap We first need to populate the replay memory, though we don't entirely fill it. In order to do so, we make the agent play random actions until we fill roughly a fifth of

[2]As VizDoom does not support natively this resolution, we used the $200 * 125$ resolution and downsampled it before feeding it to the agent.

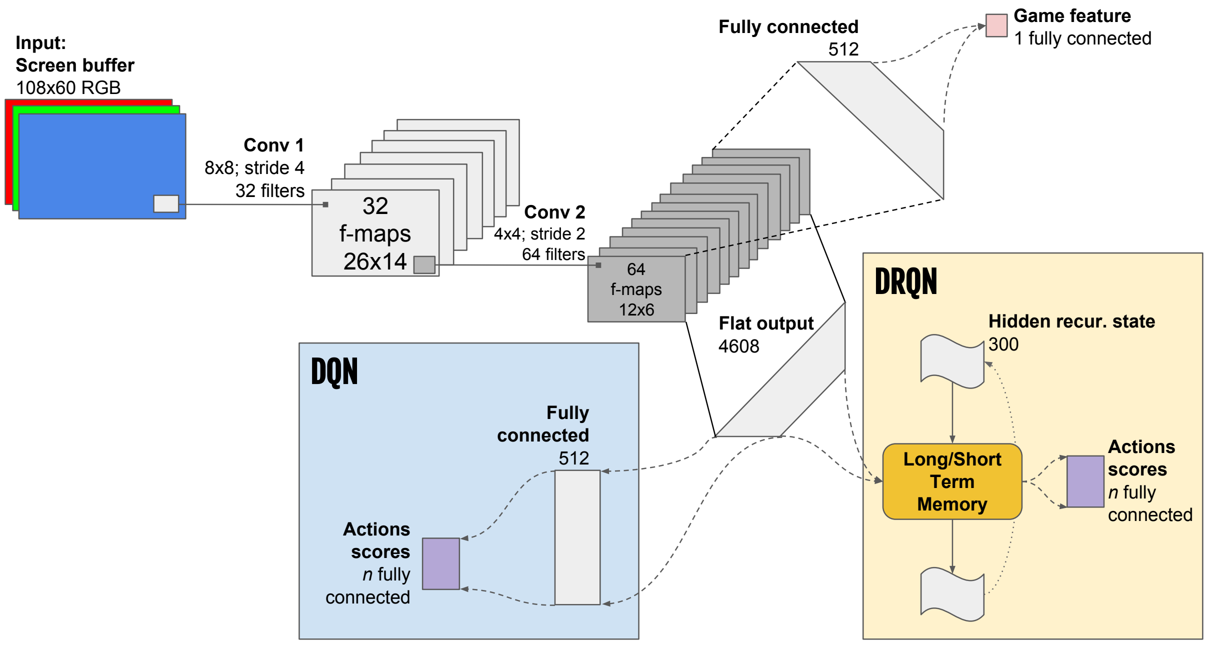


Figure 1: Our network and its variations

the memory capacity. In this phase, the neural network does not do anything.

Learning During the learning phase, we make the agent play with an ϵ-Greedy policy. The ϵ parameter starts at 1 and is linearly decreased over time until it reaches its minimal value of 0.1. Once a game episode is finished, it is inserted in the replay memory. Then, we replay 15 times a different batch of randomly selected episodes from the replay memory. For each of these replays, we run the backpropagation through the main network to learn Q. On some variations of our experiments, we also run the backpropagation on the game features part of our network.

Every 10 played episodes, we update the target network weights with those learned in the main network. Therefore, the target network is updated every 150 back propagations.

Finally, we export a video every 200 played games, where the agent plays on its own without random actions ($\epsilon = 0$), to let human observe how the agent strategy is actually evolving over time.

Testing In this phase we let the agent play on its own, without random actions nor dropout.

Action phase

As opposed to the paper (Lample and Chaplot, 2016) which uses a navigation phase as well as an action phase, we decided to only focus on the action phase. The navigation phase is the phase where the agent focuses on exploring the map to collect items and find enemies. We indeed dropped that step to immediately concentrate our efforts on the action phase, which consists in shooting at enemies when they are detected. This so-called action phase will of course take place after a training phase during which the agent learns how to interact properly with the game.

Implementation

We implemented the model using Python 3 and Tensorflow (Abadi et al., 2015). It is available publicly on Github [3].

Our experiments

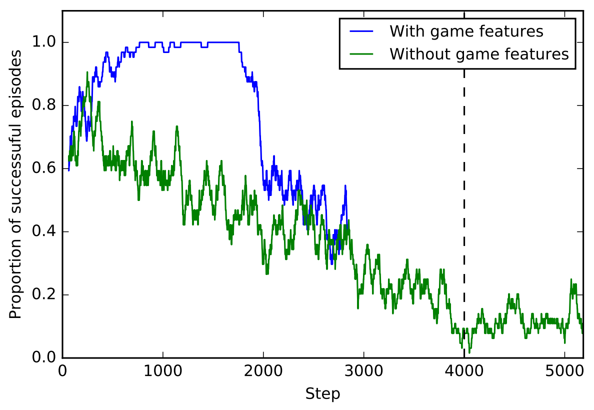
Test environment

We ran our simulations on a high-end machine which specifications are displayed in Table 1. A typical run of 5000 steps on this machine lasts approximately 40 minutes.

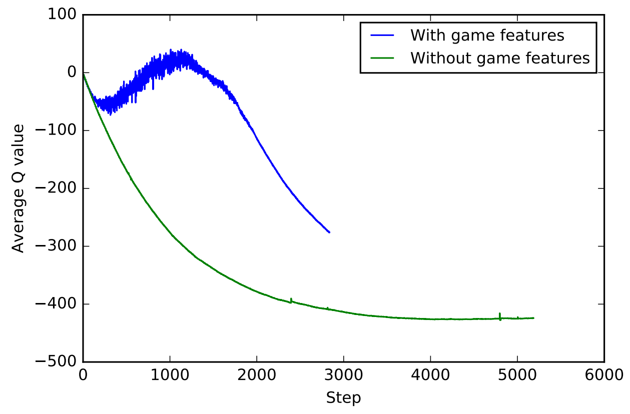
Test Setup	
CPU	Intel i7-6800K 3.4 GHz
GPU	Nvidia GeForce GTX 1080 8Go
RAM	32 Go DDR4

Table 1: Test setup hardware specification

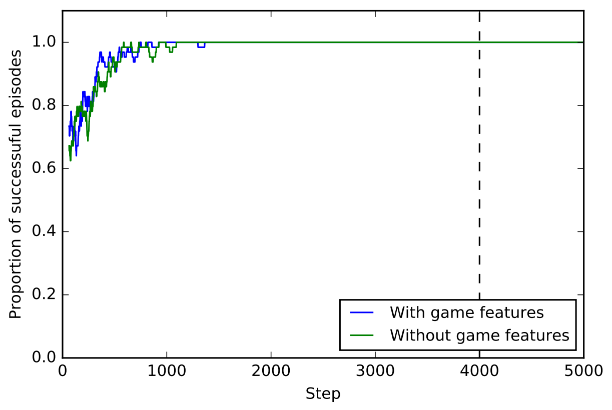
[3] `https://github.com/fhennecker/deepdoom`



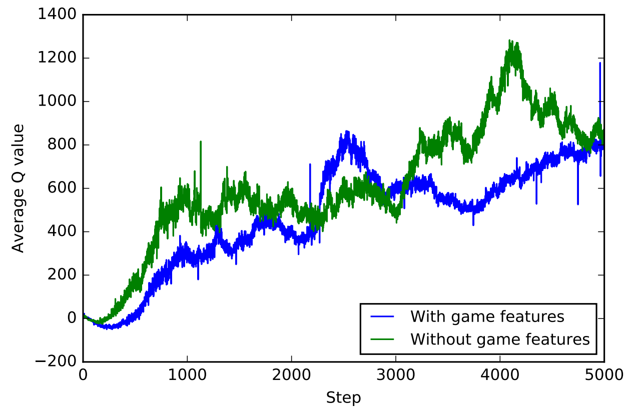
(a) DRQN



(a) DRQN



(b) DQN



(b) DQN

Figure 2: Proportion of successful episodes

Figure 3: Average Q-values

Fully Observable Markovian Process (FOMP) We first allow the agent to play the following actions:

- `MOVE_LEFT` (slide to the left)
- `MOVE_RIGHT`
- `ATTACK` (fire with the gun)

therefore, the enemy never goes out of sight for the agent and the state is fully observable.

Both DQN and DRQN were run with and without the game feature specifying the lateral position of the enemy against the back wall of the basic map. Figure 2 compares the rolling proportion of successful episodes in these 4 settings. The rolling proportion is the number of episodes in which the agent killed the enemy before the episode ended in the last 64 episodes.

The good performance of DRQN then followed by a sudden drop in performance could be explained by the fact that value function approximators are not guaranteed to converge (Fairbank and Alonso, 2011), and this holds true for neural networks (Papavassiliou and Russell, 1999). We make the hypothesis that adding a recurrent connection could break the convergence property claimed by DQN. We can indeed see on figure 3 that even if our agent learns an excellent Q-function, its Q values will at some point start diverging, causing the rolling proportion of successful episodes to drop significantly. This drop doesn't seem to occur when we remove the recurrent feedback and use DQN instead.

In fact, DQN performs very well with or without features. This result could be expected as the environment in this setting is fully observable (i.e. the enemy is always in the field of view of the agent). One frame is then sufficient for the agent to make the optimal decision.

Looking at test episodes of the DRQN agent, the behavior of the agent is excellent (it almost immediately turns to the correct direction and then shoots the enemy) until the Q-values diverge at which point the agent will just pick the same action over and over again; either hugging the left or right wall or shooting indefinitely without moving.

Partially Observable Markovian Process (POMP) We then change the agent's action set to the following:

- `TURN_LEFT` (rotate to the left)
- `TURN_RIGHT`
- `MOVE_FORWARD` (slide forward)
- `ATTACK`

With this action set, the agent is now able to move such that the enemy is no more visible. The state is therefore only partially observable. In this setting, we use another game feature: a binary label indicating whether the enemy is visible or not.

Here, ϵ is decreasing from 1 to 0.1 in 8000 steps. Because our different code branches diverged at some point, we do not have the estimated Q-values for these simulations. However, the rolling success rate graph on Figure 4 probably indicates that the Q values also diverged. However we never reach a performance as good as in the FOMP setup. An interesting point to notice is that DQN stills perform very well in this setup. We emit the hypothesis that fully observable episodes could still occur in this setup, and that the agent optimizes its behaviour to reach those optimal episodes. To verify this hypothesis, we propose to randomly rotate the player at the beginning of the episode, such that it is forced to explore the environment. However, we didn't implement this supplementary test by lack of time.

Looking at the exported videos however, the DQN version can be either really good at shooting the ennemy very fast, or can get stuck in a state where it only use the `ATTACK` action while facing a wall. In the DQRN version, starting around step 8000, the agent often falls in a state where it toggle between the actions `TURN_LEFT` and `TURN_RIGHT`. For most of those flickering, the ennemy or a corner is in the center of our agent's view cone. It probably indicates that the convolutional layer has detected an interesting pattern over there, and stimulates the Q-evaluating layers.

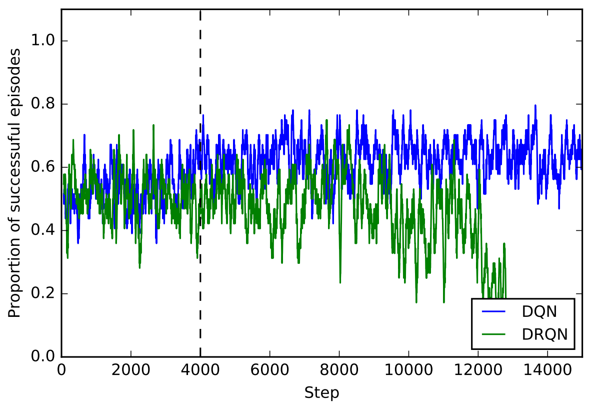


Figure 4: Rolling success rate of the agent in a POMP setup

Discussion

We presented a stable DQN able to play a basic DOOM game in which the enemy is always visible, and a DRQN that was also able to play but only with the help of jointly trained supervised game features to accelerate training. Finding that the performance of the DRQN agent was stable but only until a breaking point, we made the hypothesis that the convergence of DRQN is not guaranteed; however this could also be a sign that our implementation is either wrong or that our parameters are not tuned properly.

Finding that the DRQN did not perform better than the DQN in the partially observable Markov process setting can only hint towards the fact that our implementation and parameters needs more investigating into.

Acknowledgments

We would like to thank the SOFT department from the Vrije Universiteit Brussel, and more particularly Christophe De Troyer, for setting up and lending us a powerful machine to run our simulations. It helped us a lot by reducing the required computing time by an order of magnitude.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Fairbank, M. and Alonso, E. (2011). The divergence of reinforcement learning algorithms with value-iteration and function approximation. *CoRR*, abs/1107.4606.

Hausknecht, M. J. and Stone, P. (2015). Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527.

Hinton, G., Srivastava, N., and Swersky, K. (2012). Neural networks for machine learning, lecture 6a.

Hochreiter, S. and Schmidhuber, J. J. (1997). Long short-term memory. *Neural Computation*, 9(8):1–32.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Netw.*, 4(2):251–257.

Kempka, M., Wydmuch, M., Runc, G., Toczek, J., and Jaskowski, W. (2016). Vizdoom: A doom-based AI research platform for visual reinforcement learning. *CoRR*, abs/1605.02097.

Lample, G. and Chaplot, D. S. (2016). Playing FPS Games with Deep Reinforcement Learning. (2015).

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. a., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Papavassiliou, V. A. and Russell, S. (1999). Convergence of reinforcement learning with general function approximators. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'99, pages 748–755, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958.