Udacity – Machine Learning Engineer Nanodegree
Capstone Report

Building an AI-Bot Playing the Game Doom: how Deep Reinforcement Learning enables Agents to Learn Policies in pseudo 3D Environments

Rodolphe Masson
January 31st, 2017
rodolphe.masson8[at]gmail.com

Table of Contents

EXECUTIVE SUMMARY

In this report, we investigate various deep reinforcement learning techniques, standard deep Q-network (DQN), Replay Memory (RM), Double DQN and Dueling DQN, applied to a software agent whose goal is to play autonomously the game Doom through three different scenarios of increasing complexity, “basic”, “health gathering supreme” and “deadly corridor”, from the ViZDoom Environment.

After having explained what deep reinforcement learning is, described the ViZDoom environment and the three scenarios, we implement a standard DQN fitted with a Replay Memory and run it with different sets of parameters to compare performance by analyzing the training, testing and loss scores. We show that DQNs perform very well for fully-observable Markov Decision Process (MDP) situations in the ViZDoom environment, not so well for partially-observable MDPs, specifically where the agent needs to learn different skills like navigation & actions.

We then list and describe the limitations of standard DQNs for partially-observable MDPs, “temporal difference”, “sparse and delayed rewards” and “overestimation” before implementing a dueling double DQN (Dueling DDQN) fitted with a Replay Memory whose aim is to solve “temporal difference” and “overestimation”. We observe that the “sparse and delayed rewards” issue is very dominant among all the issues and that it overshadows the benefits of dueling DDQN. We then recommend prioritized experience replay (PER) as another potential improvement of dueling double DQN to contain the “sparse & delayed rewards” issue.

We then conclude on the most promising techniques to pursue this project in the future like Asynchronous Advantage Actor-Critic (A3C) and Direct Future Predictions (DFP) from the supervised learning field.

1. INTRODUCTION

1.1 The context of Reinforcement Learning

The most fascinating part of Artificial Intelligence are “autonomous agents” capable of sensing their environment, deciding upon and executing an action, and learning from the consequences of such an action. Well-known mature and immature applications are:

- autonomous cars in which, for example, a car automatically slows down after the recognition of a speed limit sign, or drive through a busy crossing after having detected other cars have stopped

- Natural Language Understanding (NLU) chatbots which interact with customers using speech recognition (SR), Text-to-Speech (TTS) and semantic analysis technologies to run intelligent natural conversations and solve customer requests

- Next Best Action (NDA) systems which, in the frame of customer engagement, analyze customer data to decide on the next best action / next best offer to execute (ahead of customer interaction as well), with the aim to increase customers’ engagement with a brand and the sales of the enterprise

Reinforcement Learning (RL) is a field of Machine Learning where an autonomous (software) agent:

- senses the environment it evolves in
- bases decisions and actions on this (incomplete) information
- receives a positive or negative reward from the environment because of the action taken
- leverages the reward (learning signal) to learn a policy (decision model) that will tell him what best actions to do in future situations

This environment is generally described as a Markov Decision Process (MDP) which provides the mathematical framework for modeling decision making in situations where the outcomes are partly random and partly under the control of a decision maker. MPDs are known since the 1950s.

The latest developments in this area (“autonomous agents” winning against top players in Go, AlphaGo, Chess etc.) have confirmed that Reinforcement Learning is not only promising but a key stone of self-learning systems & artificial intelligence.

The biggest RL challenge is the fact that the agent learns from interacting with its environment. This is a trial-and-error methodology where we don’t know the examples to train on, we don’t know the right and wrong values for the examples, we only know an overall goal, sense the environment and take steps to maximize both immediate and long-term rewards. The signals the agent receives is the reward function. On one hand this is a kind of supervised learning (because of the reward function), on the other hand big labeled datasets are not necessary and this is the big advantage of RL because it is, in many real-life cases, not possible or too costly to produce such datasets, specifically in the cases of high-dimension problems.

1.2 Project goals

The goal of this project is to investigate Deep Q-Networks (DQN) and various enhancement techniques like Memory Replay, Double DQN and Dueling DQN to see whether these techniques can master increasingly complex scenarios and make an agent learn autonomously playing the game Doom.

Specifically, the following challenges of Reinforcement Learning are considered:

- Observations from the agent are strongly correlated in time and the latest observation is more important than earlier ones (creating data imbalance)
- Beyond “perfect information” situations like for example the cartpole game (OpenAI Gym) where information is fully available at every time “t”, more standard situations imply “imperfect information” (non-observable) and huge state-action spaces

- the consequence of an action taken by the agent can materialize after many transitions in the environment: this is the "credit assignment" problem, Sutton et al. (2017)
- Finding the right balance between exploration (taking random actions in the state-action space to increase learning opportunities) and exploitation (leverage what the agent has already learned to reach its goal)

We have seen various approaches in the last years attempting to solve this "reward-driven behavior" challenge: value functions based upon Deep Q-Network (DQN), experience replay, target networks, Recurrent Neural Networks (RNN), Policy Gradients, and policy search like Actor-critic, Yuxi et al. (2017).

I have chosen the game Doom (ViZDoom) as environment for this project, see §1.4. This is a game where the agent learns from raw experience, i.e. pixel information, and is fed with "states" composed of 1 grey-scaled frame (30 x 45 pixels) of the video games, and it produces an output, the action the autonomous agent will play. In the case of the "Basic" scenario, the agent will only have the choice of one action at a time between strafe_right, strafe_left and attack (fire), i.e. 3 possible output combinations (not 8). To sum up, the size of the state space we be 1 (frame) x 1350 (pixels) x 256 (grey-levels) x 3 (actions) ~ 1 million. We will see in this project how a Convolutional Neural Network (CNN) can help us implement the learning function in such a huge space; we will also investigate its limitations and possible improvements.

Before describing the game Doom and the ViZDoom environment, let's start with deep reinforcement learning.

1.3 Deep Reinforcement Learning

The object of this project is to implement a neural network to play the game of Doom, where this neural network learns to propose the best next action to the agent playing the game based on the reward received by the game. We could think that this is a typical classification problem from supervised learning where the inputs to the network are the game screen images and the output is one the possible actions. However, this would require us to provide the network with training examples from recording of games played by experts, i.e. a kind of imitation learning. The more simple and intuitive approach we adopt in this project is to build a network that repeatedly play the game and learn the best actions based on the rewards it receives from the game (whether it succeeds or not). This approach is the one that humans use.

For the theoretical explanations of DRL, I have adapted the "AI Game Playing" chapter from (Gulli, 2017) to our context.

Markov Decision Process

The most common way to represent this problem is through a Markov Decision Process (MDP). The game is the environment within which the agent tries to learn. The state of the environment at time t is given by s_t and is the image frame given by the game at this time t. The agent can perform an action a_t which results in a reward r_t which can be positive or negative depending on, for example, an increase or a decrease in the game score or any other game feature we take like, for example, health of the player. This action a_t changes the environment and leads to a new state s_{t+1} where the agent can perform another action a_{t+1}, and so on. The set of states, actions and rewards together with the rules of transitioning from one state to the other, is the MDP. A single game is one episode of this process and is represented by a finite sequence of transitions. According to this MDP modeling, the probability of the next state s_{t+1} only depends on the current states s_t and action a_t.

Maximizing future rewards

For our agent to play successfully the game, we want to maximize the total reward for each game and to do this, the agent needs to maximize the total reward R_t for each step t. However, because it is hard to predict the rewards the further we go in the future, the agent needs to maximize the total discounted future reward at time t. For this we discount the reward at each future time step by a factor γ. If $\gamma = 0$, our network wouldn't consider future rewards at all; if $\gamma = 1$, then we would have a deterministic network. In the notebook, we use a value of 0.99 for the discount_factor, an empirical value recommended by Gulli et al. (2017).

$$R_t = r_t + \gamma * R_{t+1}$$

Q-learning

No policy the agent needs to follow is defined upfront, i.e. the policy will need to be learnt by the agent through training: this is what we call a model-free learning. DRL uses Q-Learning to perform this model-free learning. Q-Learning is used to find an optimal action for any given state in a finite MDP. Q-Learning tries to maximize the value of the Q-Function which represents the maximum discounted future reward when the agent in state s(t) performs action a(t):

$$Q(s_t, a_t) = \max (R_{t+1})$$

Once we know the Q function, the optinal action a for state s is the one with the highest Q-value. We can then define a policy π (s) that gives us the optimal action at any state:

$$\pi(s) = \arg_a \max Q(s,a)$$

We can then define the Q-function for a transition point $\{ s_t, a_t, r_t, s_{t+1} \}$ in terms of the Q-function at the next point $\{ s_{t+1}, a_{t+1}, r_{t+1}, s_{t+2} \}$ similar to the total discounted feature reward. This is the Bellman equation:

$$Q(s_t, a_t) = r + \gamma * \max_{at+1} Q(s_{t+1}, a_{t+1})$$

The Q-function can be approximated using the Bellman equation. Intuitively, the Q-function is a kind of look-up table where the states "s" are rows and actions "a" are columns and the table elements are the Q(s,a) are the rewards the agents gets when in state "s" and performing action "a". The best action to be performed by the agent at any state is the action with the highest reward. We will see in the algorithm implemented later that the agent first performs random actions to observe the rewards and to update this look-up table. The values get then iteratively refined. Here is a high-level description of the algorithm:

- Observe initial state s
- Repeat
    - Select and carry out action a
    - Observe reward r and move to new state s'
    - $Q(s,a) = Q(s,a) + \alpha*(r + \gamma*\max_{a'} Q(s',a') - Q(s,a))$
    - s = s'
- Until game over

The general concept of this algorithm is to perform stochastic gradient descent on the Bellman equation, backpropagating the reward through the state space (episode) and averaging over many trials (epochs). Alpha is the learning rate that determines how much of the difference between the previous Q-value and the discounted new maximum Q-value should be incorporated.

<u>The deep Q-network as a Q-function</u>

In our implementation, a state s is going to be represented by a frame provided by the Doom Game, a frame of size 30 x 45 with 256 grey levels (we implement the lowest resolution for time calculation reasons) and 3 possible actions for the "basic" scenario" (strafe_right, strafe_left, fire): this leads us to a state space of dimension 30 x 45 x 256 x 3 = ~1 million. It is important to note that many of these states are impossible or highly improbable combinations of pixels. Additionally, convolutional neural networks (CNN) have local connectivity so they will avoid these improbable pixel combinations. Neural networks are generally very good at finding good features out of structured data like images. Here I want to show that a CNN can be used very effectively to model the Q-function.

In my implementation of the Q-function, I will use convolution and fully connected layers, but no pooling layers because pooling layers make the network less sensitive to the location of specific objects in the image. In our case of playing a game, it is very important to keep the location to compute the reward.

In the case of the "basic" scenario where there are three possible actions, i.e. 3 possible Q-values as network output, we are in a regression case and we can optimize the network by minimizing the difference of the squared error between the current value of Q(s,a) and the computed value in terms of the sum of the reward and discounted Q-value(s',a') one step into the future. The current value is known at the beginning of the iteration and the future value is calculated with the Bellman equation. We have then the following loss function:

$$L = ½ * [r + \gamma*\max Q(s', a') - Q(s,a)]^2$$

I will come back to the exact architecture of the network later in this report.

Balancing exploration with exploitation

Reinforcement learning is a case of "online learning" where training and prediction steps are taking place together. At the opposite of batch learning we know from supervised learning where learning takes place on the entire training data, here a predictor is continuously improving as it trains on new data (new states from new episodes).

Therefore, a deep Q-network provides random predictions during the initial epochs of training, random predictions leading to poor Q-learning performance. We are going to use an epsilon-greedy exploration mechanism to solve this problem: the agent chooses the action suggested by the network with a probability (1-epsilon) or a totally random action. The epsilon function is going to decay over the epochs from exploration to exploitation phases:

- At the beginning, we want to ensure that the system balances the unreliable predictions from the Q-network with completely random actions to explore the state space - exploration

- Then we want the agent to use the predictions from the Q-network which get more consistent along the number of epochs, i.e. perform actions based on what the Q-network has "learnt" over the epochs - exploitation

Exploitation is possible because the Q-function is going to converge along the epochs, a convergence we are going to monitor with the loss function.

In the implementation, I am going to tune the decay function to find the optimum between exploration and exploitation.

The Replay Memory (RM) stores experience

Based on the equations presented above, we could train the network to predict the best next state s' given the current state (s, a, r). It turns out that this tends to drive the network into a local minimum because consecutive training samples are strongly correlated in time (data imbalance).

To avoid this, we collect all the transitions (s,a,r,s') into a cyclic queue called the Replay Memory when playing the game. This replay memory represents the experience of the agent. We then randomly select batches of transitions from the replay memory (instead of taking the most recent transition) and train the network with these batches (batches of typical size 64). Thanks to training on batches of randomly chosen past transitions, the network trains better and avoid getting trapped into local minima, i.e. the Replay Memory is a solution to the temporal correlation issue created by serial transitions.

Since we don't have a starting dataset of "good" transitions, the network will train on incorrect q-values at the beginning, because these are the q-values the network will have predicted with its initial random parameters. Over time we will see that the correctness of these q-values improves along with the network weights and biases.

I will provide the details of the implemented replay memory in the next chapter.

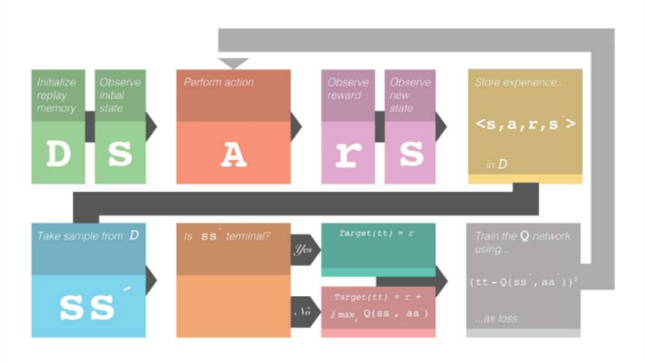


Figure 1: DQN Overview with Replay Memory (https://www.kdnuggets.com/2017/09/5-ways-get-started-reinforcement-learning.html)

1.4 The game Doom

Doom is a First-Person-Shooting (FPS) game where a player needs to navigate through mazes, shoot enemies and collect items like ammunition, keys, protecting jackets and medkits (to recover health). The successful player will have learnt the following variety of skills:

- navigating through a map
- collecting items
- recognizing and fighting enemies

More details about the game Doom, its credits and history, and where to download demo software can be found here: https://en.wikipedia.org/wiki/Doom_(series)

Figure 2: a screenshot of the game Doom

1.5 The ViZDoom environment

The ViZDoom environment is a Doom-based AI research platform for reinforcement learning from raw visual information. It allows developing AI bots that play Doom using only the screen buffer. ViZDoom is primarily intended for research in machine visual learning and deep reinforcement learning.

Credits

The developers of this environment are Michal Kempka, Grzegorz Runc, Jakub Toczek, Marek Wydmuch and Wojciech Jaskowski from the Institute of Computing Science, Poznan University of Technology, Poland. Their latest publication is Kempka et al. (2016).

Links

The home of ViZDoom is http://vizdoom.cs.put.edu.pl/

A very helpful tutorial is available here: http://vizdoom.cs.put.edu.pl/tutorial

The GitHub repo is: https://github.com/mwydmuch/ViZDoom, where you will find the environment, some examples and the configuration files for multiple scenarios

Installation

Please note the installation of WiZDoom on Windows can be very cumbersome, this is why I recommend PC users to install Linux Ubuntu 16.04 (www.ubuntu.com/download) on a VMware Virtual Machine (www.vmware.com/products/workstation-player/workstation-player-evaluation.html) in order to install the ViZDoom environment without losing time in debugging.

The installation link is: https://github.com/mwydmuch/ViZDoom/blob/master/doc/Building.md

Please note that after having reached the situation where you can call vizdoom from any place on your Linux machine (you can test this in a terminal by calling >python3 then >import vizdoom), then it is not necessary to perform the make and cmake steps.

Examples

The GitHub repo contains example files and the configuration files for multiple scenarios are available.

For this capstone project, the code available in the notebook is an adaptation of the file /ViZDoom/examples/python/learning_tensorflow.py available on the GitHub repo /mwydmuch/ViZDoom/tree/master/examples/python

The credits for this file go to the developers of the ViZDoom platform listed above.

The ViZDoom Python API

The object "game" gets instantiated with game = DoomGame()

Then we use the following routines to manage this object / the environment:

- game.init() : starts the environment
- game.load_config() : loads configuration files
- game.set_window_visible() : "true" for watching, "false" for (offline) training/testing
- game.set_mode() : player, spectator etc.
- game.set_screen_format() : selects screen format
- game.set_screen_resolution() : selects screen resolutions
- game.new_episode() : starts a new episode
- game.is_episode_finished() : "true" means the episode is finished (to test ends of game condition)
- game.close() : closes the window

These are the routines used for the agent-environment interactions:

- game.get_available_buttons_size() : collects available actions the agent can do (defined in configuration file)
- game.get_state().screen_buffer : gets a frame (=a state), raw visual inputs
- game.make_action() : get the agent execute the action in the current frame
- game.set_action(), game.advance_action() : same as game.make_action(), smooth rendering for watching the agent play
- game.get_total_reward() : collects reward from the environment

More details in the tutorial.

These are all the ViZDoom-related routines you will find in the algorithm.

2. STANDARD DQN IMPLEMENTATION

2.1 The 3 scenarios “Basic”, “Health gathering supreme” and “Deadly corridor”

These are the 3 scenarios I have chosen to test and tune my implementations in this capstone project:

1. Basic
   - Objective: the player needs to "recognize/locate and shoot the monster”
   - Map: the map is a rectangle with gray walls, ceiling and floor. The player is spawned along the longer wall, in the center. A red, circular monster is spawned randomly somewhere along the opposite wall
   - Possible actions for the player (3): strafe_right, strafe_left, attack (fire)
   - Successful game: 1 hit is enough to kill the monster. The episode finishes when monster is killed or on timeout (timeout = 300)
   - Game variable: ammunition
   - Reward (from game): +101 for killing the monster, -5 for missing
   - Reward (through configuration): -1 for living
   - Configuration file: basic.cfg
   - Complexity level: low

Figure 3: a screenshot of the “Basic” scenario where the player’s environment is fully observable – The monster is on the left

2. Health gathering supreme
   - Objective: the player needs to recognize, locate and collect medkits to heal himself because he is evolving on an acid ground. Here we want to teach the agent how to survive without knowing what makes him survive. The agent only knows that life is precious, and death is bad, so he must learn what prolongates his existence, i.e. collecting medkits. While reward shaping could be a powerful solution here, we don’t provide rewards on collecting medkits.
   - Map: the map is a rectangle with a green, acid floor which hurts the player periodically. Initially there are some medkits spread uniformly over the map. A new medkit falls from the skies every now and then. Medkits heal some portions of player's health - to survive the agent needs to pick them up. Episode finishes after player's death or on timeout.
   - Possible actions for the player (3): turn_left, turn_right, advance
   - Game variable: health
   - Reward (through configuration): 1 for living, 100 is the death penalty
   - configuration file: health_gathering.cfg with health_gathering_supreme.wad
   - Complexity level: mid

Figure 4: a screenshot of the “Gathering health supreme” scenario where the player’s environment is only partially observable – The boxes are the medkits the player needs to collect

3. Deadly Corridor
   - Objective: the player needs to recognize/locate/shoot monsters while navigating towards an objective (green jacket). The reward is proportional (negative or positive) to the change of the distance between the player and the vest. If player ignores monsters on the sides and runs straight for the vest he will be killed somewhere along the way. To ensure this behavior the difficulty level is high (doom_skill = 5)
   - Map: the map is a corridor with shooting monsters on both sides (6 monsters in total). A green vest is placed at the opposite end of the corridor
   - Possible actions for the player (7): strafe_left, strafe_right, attack (fire), forward, backward, turn_left, turn_right
   - Successful game: the agent has survived, timeout = 4200
   - Game variable: health
   - Reward (from game): +dX for getting closed to the jacket, -dX for getting going away from the jacket
   - Reward (through configuration): death penalty is 100
   - Configuration file: deadly_corridor.cfg
   - Complexity level: high

Figure 5: a screenshot of the “Deadly corridor” scenario where the player’s environment is only partially observable. Please notice the green jacket at the end of the corridor (the navigation target) and the monsters on both sides

The provided complexity level is somewhat subjective and reflects the number of separate set of actions the agent needs to learn, from low level "locate and shoot / fully observable" to mid-level "locate and collect or avoid / partially observable", to high-level "locate and shoot several times while following a target / partially observable".

We don’t have non-observable state in the first scenario because the agent moves strafe_right or strafe_left with a scene always developing in the sight field of the agent. This is not the case for scenarios 2. and 3. where

the agent can turn, and the game develops in a 360-degree field. The scenario 2 and 3 have an additional level of complexity because several sets of actions need to be learnt, “locate/collect medkits” & “locate/avoid poison” for scenario 2, and “locate/shoot monster” & “navigate to a target” for scenario 3. Specifically, for the “deadly corridor” scenario, the number of actions is 7 which considerably increases the size of the state space to be explored.

2.2 Algorithm structure & metrics

Here is the detailed structure of the program (please consider following on the notebook in parallel)

- import tools
- define gobal variables: config_file_path, epochs (20), learning_steps_per_epoch (2000), batch_size(64), test_episodes_per_epoch(100), episodes_to_watch(10)
- create Doom instance, calling initialize_vizdoom() with "config_file_path" of the scenario
- collect which buttons can be pressed into "actions"
- create replay memory Object, "memory", calling ReplayMemory Class with "replay_memory_size
- instantiate tensor flow "session"
- instantiate functions learn(), get_q_values(), get_best_action() and neural network, calling create_network() for the object session and for the number of "actions" len(actions)
- timestamp

TRAINING_TESTING loop on epochs, for epoch = 1 to 20
- init counter "train_episodes_finished" and "train_scores" table
- print epoch, print "now starting training phase"
- instantiate new episode of game, calling game.new_episode()

LEARNING_STEP loop, for learning_step = 1 to "learning_steps_per_epoch" (2000)
- **perform_learning_step(epoch)**
- if episode finished:
  - collect score, calling game.total.reward()
  - add score to "train_score"
  - start new episode, calling game.new_episode()
  - increment "train_episodes_finished" counter

END OF LEARNING_STEP LOOP

- print number of episodes played, "train_episodes_finished"
- print results of training "train_score"

- print "testing starts"
- init "test_episode" and "test_scores"

TEST_EPISODE loop, for test_episode = 1 to "test_episodes_per_epoch" (100)
- create new episode, calling game.new_episode()
- while episode is not finished (= play the episode)
  - sets state, calling preprocessed game.get_state().screen_buffer
  - get best_action_index for this state, calling **get_best_action()**
  - make action, calling game.make_action() for the action with this index "best_action_index"
- collect total reward, calling game.get_total_reward()
- store reward in "test_score"

END OF TEST_EPISODE loop

- print test results, mean, min & max
- save the network weights to model_savefile
- Print time to run this epoch (learning + testing loops)

END OF TRAINING & TESTING loop on epochs

Close game, calling game.close()
print("learning & testing are finished")

print ("watching the agent play")
- reinitialize the game with window visible, calling game.set_window_visible(True) and game.set_mode(Mode.ASYNC_PLAYER), game.init()
- loop on number of episodes to watch, global variable "episodes_to_watch" (10)
  - start new episode, calling game.new_episode()
  - while episode not finished, game.is_episode_finished()
  - while episode is not finished (= play the episode)
    - sets state, calling preprocessed game.get_state().screen_buffer
    - get best_action_index for this state, calling **get_best_action()**
    - make action, calling game.make_action() for the action with this index "best_action_index"
    - smooth rendering with game.advance_action()
    - sleep 1 sec between episodes
    - collect score, calling game.get_total_reward()
    - print score

Plot traing/testing scores and loss results

<u>Replay Memory</u>

The object "memory" is instantiated by calling the ReplayMemory Class with "replay_memory_size", an object with the following child functions:

- add_transition(): to add transitions to the object

- get_sample(): to get a sample of size batch_size from the list of already stored transitions

Important note: as explained in §1.3, the Replay Memory stores ALL transitions {s1, a ,s2 , isterminal , r} for all "learning_steps_per_epoch" and for all "epochs" via the routine memory.add_transition() called by perform_learning_step(). The memory is cyclic and replaces older transition over time when it has reached its maximum size. "isterminal" is a last transition flag.

learn(), get_q-values(), get_best_action()

The learn(), get_q_values() and get_best_action() functions are instantiated when calling create_network() for the object "session" and for the number of "actions" len(actions). The neural network is also instantiated.

- **learn()** performs learning of the neural network, by minimizing the loss between each of the 64 computed q-values and each of the 64 target_Q provided by the memory (64 = batch_size), see MSE-backpropagation described in §1.3

- **get_q_values()** calculates the q_values of a given list of states s, returns a list of size (len(actions)) of q values, the computed probability of each of the actions for a given state

- **get_best_action()** takes the highest value of the results from get_q_values(), i.e. select the action with the highest q-value

learning_step()

Here is what it does:

- sets s1, the state at time "t" (preprocessed game.get_state().screen_buffer)
- depending on epsilon (eps calculated by exploration_rate())
    - either makes a random action "a" (among possible action), or
    - choose the best action according to the network, "a", calling **get_best_action()**
- collect reward "r", calling game.make_action(the chosen action "a")
- check if episode is finished
- sets s2, the state at time "t+1" (preprocessed game.get_state().screen_buffer)
- store the occurred transition into "memory", calling memory.add_transition(s1, a, s2, isterminal, r)
- learn from a single transition using replay memory, calling **learn_from_memory()**

Learn_from_memory()

Here is what it does:

- gets a random minibatch of transitions from the replay memory, a minibatch of size "batch_size" (only if memory.size is bigger than batch_size), calling **memory.get_sample():** here, s1, a, s2 and r are lists with 64 items ("batch_size")
- sets q2-list as the max of the q_values for the 64 states s2, calling **get_q_values()** for state-list s2
- sets target_q to the q_value calculated by **get_q_values()** for state-list s1
- **Bellman equation**: target_q(s1,a) = reward + gamma * maxQ(s2,a)
- make network learn this new pair state-target_q, calling **learn()** for the state-list s1 and for the q-list target_q

Metrics

For the upcoming tests, we are going to use "training scores", "testing scores" and the "loss function" as metrics.:

- During each learning_step and up to learning_steps_per_epoch, the agent plays and the network learns: this is the "training score"
- At the end of each epoch, 100 (test_episodes_per_epoch) episodes are run and the "testing score" is reported
- The loss function is defined as: $L = ½ * [r + \gamma * \max Q(s', a') - Q(s,a)]^2$, and the mean of the loss function over one epoch is reported
- All metrics are displayed with matplotlib

Training and testing scores will tell us how effectively the agent learns skills and the loss function will tell us if the neural network weight and biases converge.

Benchmarks

It is important to define a benchmark before we build new models in order to make objective comparisons between techniques possible. In this project, we define our benchmarks as the average score an agent achieves over 50 runs, an agent making random actions: this is implemented by the function “choice()” from “random” library, please refer to the notebook “vizdoom_discover_final.ipynb” provided with this report. A video of this „random walk“ is provided later in this report for each scenario (see also the Resources annex). This is the benchmark we will compare our models against, to determine whether these new models have brought us closer to our goals or not. Here are the mean scores calculated over 50 runs for each of the scenarios:

- scenario "Basic": benchmark_Basic = -160
- scenario "Health gathering supreme": benchmark_HGS = +300
- scenario "Deadly corridor": benchmark_DC = -115

2.3 Neural network architecture

Fig. 6 describes the architecture of the neural network we will use to implement the Q-Learning function.

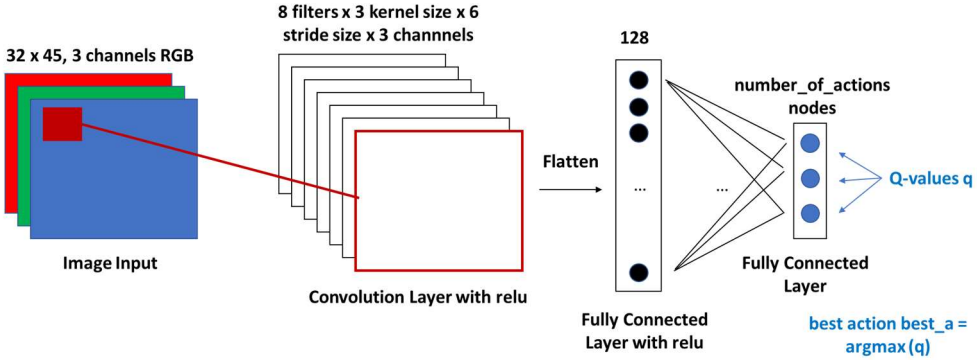


Figure 6: the NN with “feature” (the convolutional layers) and “classification” (the fully-connected layers)

The implementation with tensor flow is available in the notebook (see chapter Resources) provided with this report.

2.4 “Basic” scenario tuning

I have run a series of runs on the “Basic” scenario to tune the parameters batch_size, decay function and network architecture. An executable jupyter notebook is provided with this report, see chapter Resources.

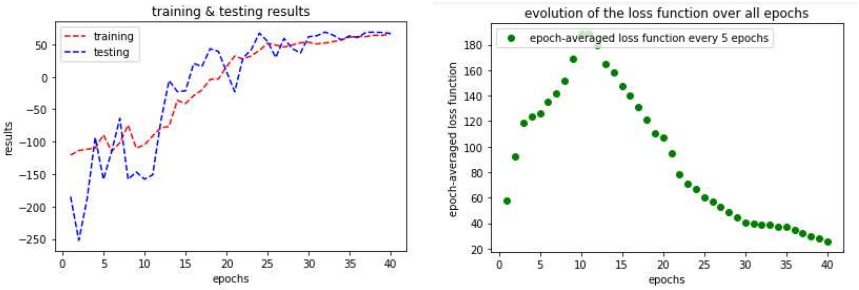
I start my tests with the following network architecture:

- 1 convolutional layer (kernel 6, stride 3, relu, Xavier weights, constant bias = 0.1)
- 1 flattening layer
- 1 fully connected layer (128 outputs, relu, Xavier weights, constant bias = 0.1)
- 1 fully connected layer (3(=possible actions) outputs, no activation, Xavier weights, constant bias = 0.1): no activation because we have to have all q-values and we separately get the best action with argmax

This architecture is very close to the one I had started with during my deep learning (dog) mini-project, specifically for the activation functions, for the careful initialization of the weights and the bias and the kernel/stride size for image analysis. The last fully connected layer is specific to our case where the num_output corresponds to the available actions for which the network will predict a probability, and important note, without activation function: this is to get the prediction probabilities from the network for all actions at the end of the network (the q-values). We calculate the “best action” with an argmax argument right after this layer.

RUN #1

Scenario = basic / batch_size = 64 / decay = **exp(-0.1*epoch)** / epoch = 40 / learning steps per epoch = 2000
Network architecture like above: 1Conv-1Flat-2FC



Figures 7a & 7b: RUN #1

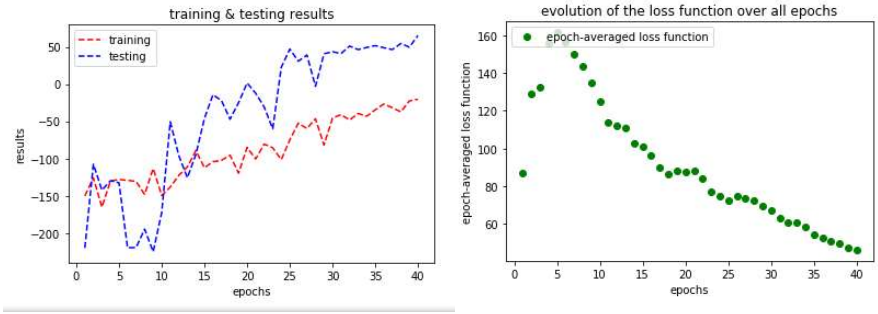
Fig. 7a shows that the final testing scores are well above 50 (max is 95 for this scenario), well above the random benchmark of -160 and a visual verification of the 10 runs confirms that the agent has well learnt to recognize the monster, go in front of it and fire. Fig. 7b shows that the loss function decreases to almost zero around epoch 40: this is the proof that there is (almost) no difference anymore between the q-values predicted by the network and the learnt ones. The loss function started to converge after epoch 12.

RUN #2

Scenario = basic / batch_size = 64 / decay = **exp(-0.01*epoch)** / epoch = 40 / learning steps per epoch = 2000

Network architecture like above: 1Conv-1Flat-2FC

We try more exploration here to see whether we can speed up learning.



Figures 8a & 8b: RUN #2

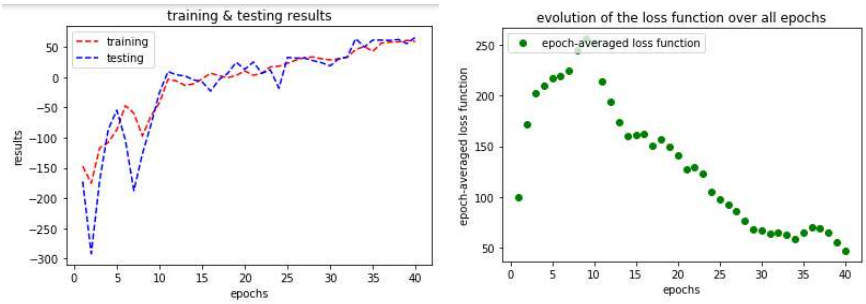
In comparison with RUN #1, we see the effect of a decay function favorizing exploration by the fact that testing and training follow the same trend but don't converge by epoch 40. Fig 8b shows that the loss decreases almost down to zero. But in comparison with RUN#1 where the testing score was above 50, here we stay at around 50. So, RUN #1 was better (even if this RUN #2 was good)

RUN #3

Scenario = basic / batch_size = 64 / decay = **exp(-epoch)** / epoch = 40 / learning steps per epoch = 2000

Network architecture like above: 1Conv-1Flat-2FC

Here, we try more exploitation to understand whether this improves performance.



Figures 9a & 9b: RUN #3

Like for RUN #2, Fig. 9a shows us that the testing score stays at about 50 whereas RUN #1 was above 50. Fig. 9b confirms the convergence of the Q-function.

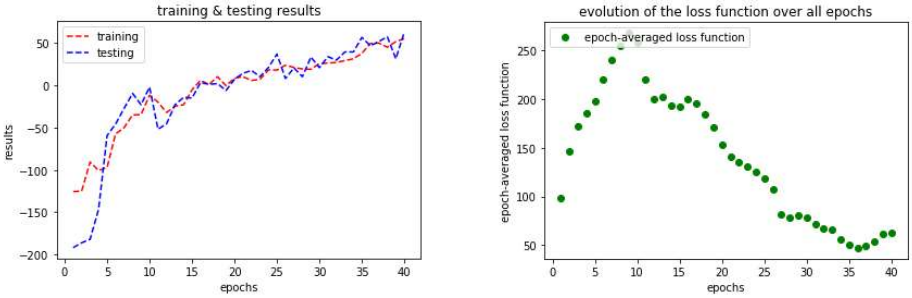
Partial conclusion: the decay function exp(-0,1*epoch) is optimal.

RUN #4

Scenario = basic / **batch_size = 32** / decay = exp(-0.1epoch) / epoch = 40 / learning steps per epoch = 2000

Network architecture like above: 1Conv-1Flat-2FC

We try a smaller batch size to see whether this impacts learning.



Figures 10a & 10b: RUN #4

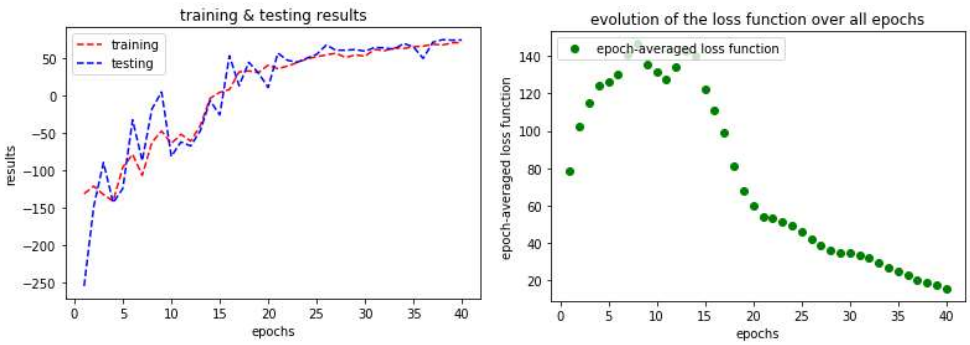
Fig. 10a tells us that the agent has not reached a testing score of 50 by epoch 50. The convergence of the loss function is fine, but batch_size = 64 was better.

RUN #5

Scenario = basic / **batch_size = 128** / decay = exp(-0.1*epoch) / epoch = 40 / learning steps per epoch = 2000

Network architecture like above: 1Conv-1Flat-2FC

We try a bigger batch size to see whether this speeds up learning convergence.



Figures 11a & 11b: RUN #5

Final testing scores on Fig. 11a are slightly above 50 (max is 95): as good as the first run, and we see a stable training curve. But the overall time (2,5 hours, against max 1,5 hour for the other runs) for the 40 epochs make this set up not so attractive. This tells us that a batch_size of 64 is the best value (RUN #1).

Conclusion

A simple DQN implementation with replay memory and with a single convolutional layer has beaten the random benchmark and makes an excellent job in this case of fully observable MDP. This is the proof that DRL works very well in 2D environments.

Demo videos

The agent taking random actions: https://youtu.be/WsG09c5YHng
The agent acting autonomously: https://youtu.be/KiaKKpqLlBU

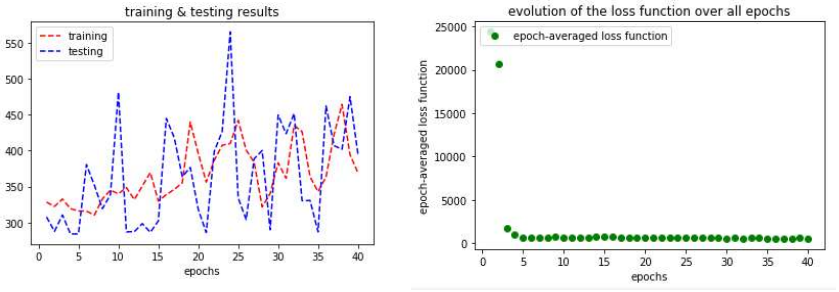
Let's now test a more complicated "3D" scenarios.

2.5 “Health gathering supreme” scenario tuning

RUN #6

Scenario = health gathering supreme / batch_size = 64 / decay = exp(-0.1*epoch) / epoch = 40 / learning steps per epoch = 2000 / **gamma = 0,99**

Network architecture like for the “Basic” scenario: 1Conv-1Flat-2FC



Figures 12a & 12b: RUN #6

The results presented by Fig. 12a and 12b are very interesting:

- the random benchmark of +300 is beaten, but not by much
- A visual check of the agent playing shows a navigation that works quite well, i.e. the agent has learnt that medkits are important for his survival and recognize them and navigate to them
- The agent hasn’t learnt to recognize and avoid the poison vials and collects them
- There is some upward trend in the testing score over the first 40 epochs, but the overall score sounds to be limited around 500 while the maximum score is 2100. This is because the navigation is sometimes not smooth, and the health level goes down fast in areas where there is no medkit. Additionally, poison vials get collected impacting the health score through a negative reward
- The loss function goes down very quickly

Before going back to this last observation about the loss function, I have done additional run with different parameters to be sure these results from RUN #6 are repeatable:

- more epochs (from 40 to 80):
  - same observations, no improvement, same quickly converging loss function, no improvement of testing score over the epochs
- more exploration (exp(-0,01*epoch))
  - same observations with a slightly improvement of navigation
- more exploitation (exp(-epoch))
  - same observations, no real improvement
- one additional convolutional layer
  - conv2 = tf.contrib.layers.convolution2d(conv1, num_outputs=8, kernel_size=[3, 3], stride=[2, 2], activation_fn=tf.nn.relu, weights_initializer=tf.contrib.layers.xavier_initializer_conv2d(uniform=True), biases_initializer=tf.constant_initializer(0.1))
  - same observation, no real improvement
- increase of the size of Replay Memory (from 10.000 to 50.000)
  - same observation, no real improvement
- more learning steps per epoch (40.000)
  - same observation, no real improvement

As explained in §1.3, we train the network by minimizing the difference of the squared error between the current value of Q(s,a) (known at the beginning of the iteration) and the computed value in terms of the sum of the reward and discounted Q-value(s’,a’) one step into the future (the future value is calculated with the Bellman equation): $L = ½ * [r + \gamma * \max Q(s', a') - Q(s,a)]^2$

The fact that the loss function quickly goes to zero means that the Q(s,a) value is the same as the discounted Q-value (s’,a’), i.e. all q-values are “up to date” and “there is nothing more to learn during training”. I have

investigated the influence of gamma on the results. Reminder: if gamma = 0, our network wouldn't consider future rewards at all and if gamma = 1, then we would have a deterministic network.

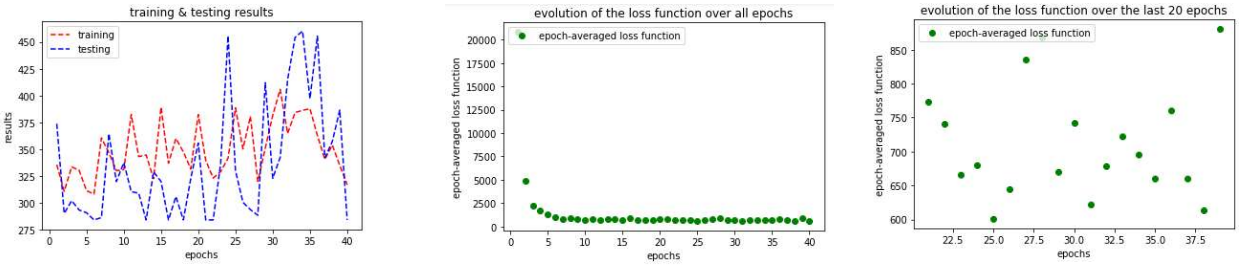
RUN #7

I am not reporting my tests with **gamma = 0.9** and **gamma = 0,999** which were not successful. The agent hadn't learnt to target the medkits, no navigation skill.

Scenario = health gathering supreme / batch_size = 64 / decay = exp(-0.1*epoch) / epoch = 40 / learning steps per epoch = 2000 / **gamma = 0,994**

Network architecture like for the "Basic" scenario: 1Conv-1Flat-2FC

Here we increase the importance of future rewards.



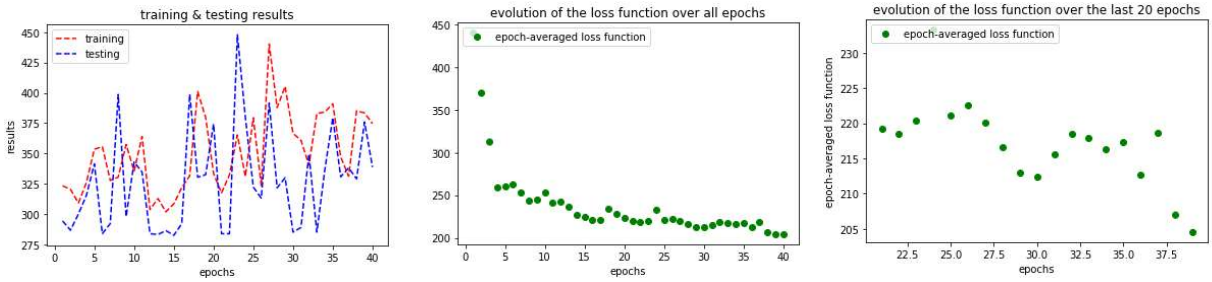
Figures 13a & 13b & 13c: RUN #7

The visual verification shows that the navigation skill has not been learnt and Fig 13.c shows the loss function is unstable.

RUN #8

Scenario = health gathering supreme / batch_size = 64 / decay = exp(-0.1*epoch) / epoch = 40 / learning steps per epoch = 2000 / **gamma = 0,96**

Network architecture like for the "Basic" scenario: 1Conv-1Flat-2FC



Figures 14a & 14b & 14c: RUN #8

Visual verification shows that navigation is working but not as good as for RUN #6. This is confirmed by the testing score clearly below 500 on Fig. 14a (lower than for RUN #6). And learning is confirmed by the loss function which has gone down to almost zero towards epoch 40.

Demo videos

The agent taking random actions: https://youtu.be/jxYesqMhx1Q

The agent acting autonomously: https://youtu.be/G7LBuRdrXl0

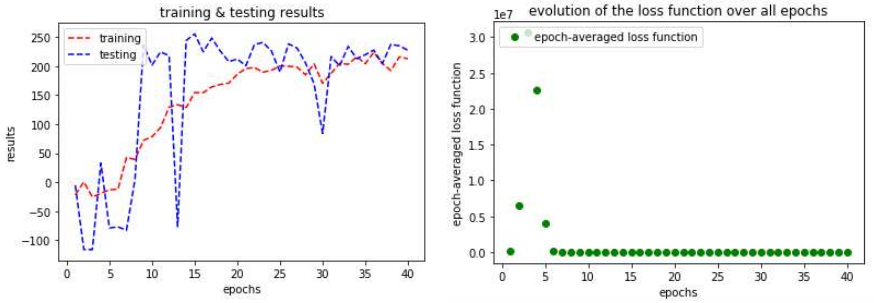
Conclusion

A simple neural network with a single convolutional layer and an algorithm leveraging a Replay Memory beats the random benchmark of +300 and does quite a good job at learning "navigation" in a 3D environment. With a gamma parameter of 0.99 and training time of roughly 2 hours (Ubuntu on CPU), it is possible to have an agent learn the policy that makes him survive on an acid ground (recognizing/locating/collecting medkits). But it is not possible to make the agent learn more than that: it was not possible for this model to also learn recognizing/avoiding poison vials "additionally to" the navigation skills. I believe we have reached the limits of simple DQN models where learning various skills in partially-observable 3D environment doesn't take place "naturally". Let's now check whether this is also the case for the next scenario, the "Deadly corridor".

2.6 “Deadly Corridor” scenario tuning

RUN #9
Here we re-use the parameters which were optimal for the previous scenarios.

Scenario = deadly corridor / batch_size = 64 / decay = exp(-0.1*epoch) / epoch = 40 / learning steps per epoch = 2000 / gamma = 0,99 / resolution 30x45 / replay memory size 10.000 / learning rate 0,00025
Network architecture like for the “Basic” scenario: 1Conv-1Flat-2FC

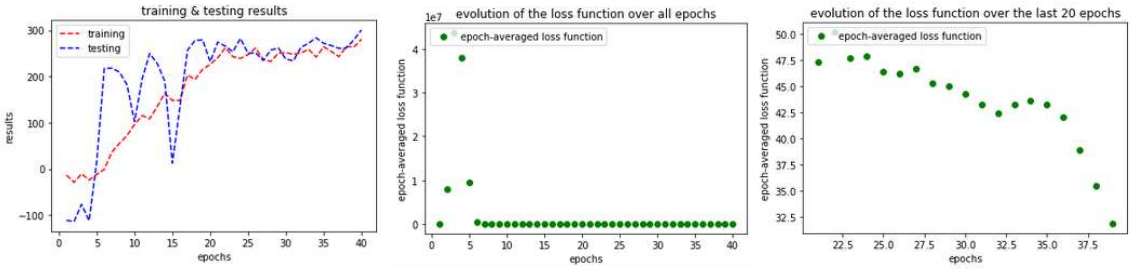


Figures 15a & 15b: RUN #9

We observe a very similar behavior as for the scenario “Health gathering supreme” where the test scoring on Fig. 9a plateaus at 250, whereas the maximum score for this game is 1000. We observe on Fig 15b a fast converging Q-function, showing that there is not much more to learn through training. Additionnally, the random benchmark of -115 is beaten. Visually, we see that the agent has only learnt to go straight to the target (the green jacket) but gets killed on his way to the corridor. A good strategy would be “turn 45° right and shoot, turn 90° left and shoot, turn 45° right and run straightforward”, to the next areas with monsters and repeat the sequence “turn 45° right and shoot, turn 90° left and shoot, turn 45° right and run straightforward”. Here we see that the agent hasn’t learnt this complex combination of navigation & action behavior.

Let’s try other parameters for the network.

RUN #10
Scenario = deadly corridor / batch_size = 64 / decay = exp(-0.1*epoch) / epoch = 40 / learning steps per epoch = 2000 / gamma = 0,99 / resolution 30x45 / replay memory size 10.000 / learning rate 0,00025
Network architecture like for the “Basic” scenario: 1Conv-1Flat-2FC, with the **bias of the convolutional layer set to 0**:
weights_initializer=tf.contrib.layers.xavier_initializer_conv2d((uniform=True),
biases_initializer=tf.constant_initializer(0))

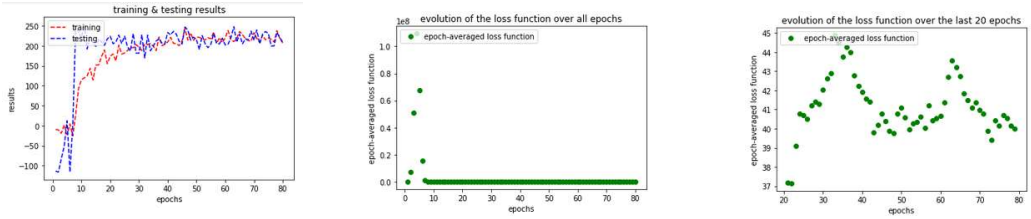


Figures 16a, 16b & 16c: RUN #10

Same observations as for RUN #9, test scoring plateauing at 250-300, a fast converging q-function and the navigation/action not learnt by the agent.

Fig. 16a shows us that the test scoring could go further up above 300 after 40 epochs. We try more epochs.

RUN #11
Scenario = deadly corridor / batch_size = 64 / decay = exp(-0.1*epoch) / **epoch = 80** / learning steps per epoch = 2000 / gamma = 0,99 / resolution 30x45 / replay memory size 10.000 / learning rate 0,00025
Network architecture like for the "Basic" scenario: 1Conv-1Flat-2FC, with the **bias initiated to 0.1**



Figures 17a, 17b & 17c: RUN #11

Fig. 17.a shows us an asymptote for test scoring between epochs 40 and 80, Fig 17b shows a fast converging Q-function in the first 10 epochs and Fig. 17c shows oscillations of the q-function with a period of roughly 40 epochs. Visually, we confirm the confirm the agent hasn't learnt the navigation/action behavior.

In the next paragraph, I am going to explain why we are having difficulties with such scenarios where the success of the agent is based on learning a complex combination of skills (navigation, action).

Demo videos
The agent taking random actions: https://youtu.be/fGpUp8b1m-E
The agent acting autonomously: https://youtu.be/UbtQP0CIdJY

2.7 Limitations of standard DQNs for partially observable MDPs

Temporal Correlation
The first issue that we usually see with DQN, is that observations from the agent are strongly correlated in time and the latest observation is more important than earlier ones, creating data imbalance. But this temporal correlation issue is solved with the Replay Memory feature of the algorithm, where we randomly select transitions (experiences) from the past and train the network with.

Fully-observable vs partially-observable
The second issue is the fact that the Q-Learning model takes the strong assumption that the agent has the full knowledge of the current state of the environment. While this is true in the "Basic" scenario (2D), it is not true for the 3D scenarios "Health gathering supreme" and "deadly corridor". For such partially-observable cases, the agent needs to remember previous states to select the optimal action. Here the Replay Memory (where batches of transitions are selected randomly) doesn't solve this issue completely, see the next issue.

Sparse and delayed rewards
The third issue is the fact that the consequence of an action taken by the agent can materialize after many transitions in the environment: this is the "credit assignment" problem, Sutton et al. (2017). Let's take an illustration with the "Basic" scenario and let's assume the monster is located 10 moves away from the agent on the right. The successful behavior will be:

- decide to take action "move strafe_right" 10 times then decide to take action "fire"

Please note, here the agent needs to take ALL these decisions before being successful, i.e. the model shouldn't only learn the last action "firing" is the success: this is a typical "n-step Q-Learning" case, Mnih et al. (2016). But the model only gets a reward for the last action "firing": this is the case of "delayed & sparse reward", Schaul et al. (2016), which makes it difficult for the agent to learn which set of actions is responsible for what reward, Lample et al. (2016). This explains why learning different skills like navigation and action is challenging with this model.

Overestimations
A fourth issue is overestimation of action values because the single estimator used in the Q-Learning update approximate the "maximum **expected** action" with the "maximum action value". Van Hasselt et al. (2015) have

proven in their publication that Q-Learning overestimates action values at a point that they are common, lead to suboptimal policies and harms algorithm performance. Specifically, in the case of non-uniform overestimation not impacting states we wish to learn more, overestimation can impact the quality of the policy the agent needs to learn.

3. STATE-OF-THE-ART TECHNIQUES IMPROVING DQNs

Here is a non-exhaustive list of techniques and approaches complementing DQNs which have solved the issues described in §2.6. A complete list of state-of-the-art techniques is available in the detailed overview from Li et al. (2017).

3.1 Double DQN (DDQN)

Van Hasselt et al. (2015) proposed Double DQN (DDQN) to address the overestimation problem in Q-Learning, see §2.7. As we have seen in §1.3 In standard DQN, the parameters are updated as follows:

$$\Theta_{t+1} = \Theta_t + \alpha \left(y_t^Q - Q(s_t, a_t ; \Theta_t) \right) \text{ηabla } \Theta t \left(Q (S_t, a_t; \Theta_t) \right)$$

…where $y_t^Q = R_{t+1} + \gamma \max_a Q(s_{t+1}, a; \Theta_t)$

This means that the max operator uses the same values to both select and evaluate action: these were the functions get_q_value() and get_best_action() we have used in the implementation in chapter 2. Van Hasselt et al. (2015) proposed to decouple selection from evaluation, i.e. to evaluate the greedy policy according to the online network but using the target network to estimate its value. This can be achieved with a minor change to the DQN algorithm, replacing y_t^Q with:

$$Y_t^{D\text{-}DQN} = R_{t+1} + \gamma Q (s_{t+1}, \arg_a \max Q(s_{t+1}, a_t; \boldsymbol{\Theta_t}); \Theta_t^-)$$

…where Θ_t are the parameters of the online network and Θ_t^- are the parameters for the target network. These two networks Θ_t and Θ_t^- are the two value functions. For each update, one set of weights is used to determine the greedy policy and the other to determine its value. Please note in this last equation that the selection of the action (in the argmax), is still done on the online weights $\boldsymbol{\Theta_t}$, i.e. we are still estimating the value of the greedy policy according to the current values $\boldsymbol{\Theta_t}$. However, we use the second set of weights Θ_t^- to evaluate the value of the policy. The target network Θ_t^- is a periodic copy of the online network Θ_t.

This is a technique we want to experiment.

3.2 Prioritized Experience Replay (PER)

Schaul et al. (2016) have shown that the agent can learn better policies when the network is not trained on batches of (uniformly) randomly-sampled transitions from the Replay Memory, regardless of the significance of the experience, but from carefully-selected batches of transitions. This concept is about replaying frequently "important" transitions for the agent to learn more efficiently (sparse & delayed rewards challenge), where the "importance" of a transition is measured by its Temporal Difference (TD). Practically, each transition is stored in the Replay Memory with its TD parameter and a stochastic prioritization based on these TDs samples the transitions. PER has shown improved performance of DQN and DDQN.

While this technique looks promising, we won't test it in this project.

3.3 Dueling architecture (Dueling DQN)

Wang et al. (2016) propose the dueling network architecture to separately estimate the state value function V(s) and its associated advantage function A(s,a), and then combines them to estimate action value function Q(s,a), to converge faster than Q-Learning.

Reminder:

- the state value function $V^\pi(s)$ is the expected return when starting in state s and following the policy π: $V^\pi(s) = E(R \mid s, \pi)$
- the optimal policy π^* has a corresponding state-value function V*(s) and vice-versa, the optimal state-value function can be defined as $V^*(s) = \max_\pi V^\pi(s)$, for all s in state space

As depicted by Fig. 18. architecturally, in DQN a CNN layer (blue) is followed by a fully connected layer (red). In dueling architecture, a CNN layer is followed by two streams of fully connected layers to estimate value function and advantage function separately: then the two streams get combined (green links) to estimate the action value function.

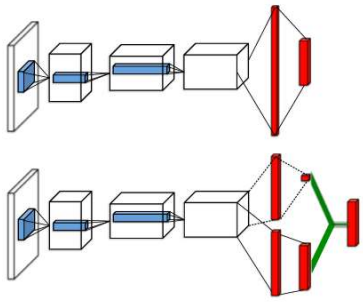


Figure 18: The Dueling Architecture, Wang et al. (2016)

Fig. 2 or their publication illustrates how the agent benefits from separated value and advantage functions on the Atari game Enduro, the first (value) stream learns to pay attention to the road and the second (advantage) stream learns to pay attention to immediate obstacles, the proof that the agent has learnt two skills for this game.

More in details, the dueling architecture can learn which states are or aren't valuable, without having to learn the effects of each action for each state. This is particularly useful in states where the action does not affect the environment in any relevant way. In fact, while for some states it is of utmost importance to know which action to take, for many states it is unnecessary to estimate the value of each action choice for many states. From a mathematical perspective, the re-combination of both streams looks like:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + [\, A(s,a; \theta, \alpha) - (1/A) * \Sigma_{a'} A(s, a'; \theta, \alpha) \,]$$

…where θ are the parameters of the convolution layers and α, β the parameters of the fully-connected layers.

Wang et al. (2016) have demonstrated that the dueling architecture can more quickly identify the correct action during policy evaluation as redundant or similar actions are added to the learning problem. The combination of the dueling DQN with Prioritized Experience Replay is one state-of-the-art technique, outperforming standard single-stream Q networks.

This is a technique we are going to experiment in the next chapter.

3.4 Asynchronous Advantage Actor-Critic (A3C)

Mnih et al. (2016) proposed A3C. in A3C, the use of parallel actors situated in their own and independent environments, not only stabilizes improvement in the parameters but has the additional benefit of creating more exploration. Experience replay is not utilized. A3C combines advantage updates with the actor-critic concept and relies on asynchronously updated policy and value function networks trained in parallel over several processing threads.

Mnih et al. (20016) have shown that A3C generalizes quite good in 3D mazes using visual inputs, where an agent will face a new maze in each new episode, i.e. the agent has learnt a general strategy to explore random mazes.

This is a promising technique that should be tested beyond this project.

3.5 Deep Recurrent Q-Networks (DRQN) & Deep Attention Recurrent Q-Networks (DARQN)

An interesting extension of DQNs is to leverage a Recurrent Neural Network (RNN) to deal with Partially Observable MDPs by integrating information over long time periods. Recurrent connections provide an efficient means of acting conditionally on temporarily distant prior observations, Hausknecht et al. (2017).

Further improvements have been gained by introducing "attention", a technique where additional connections are added from the recurrent units to lower layers, to DRQN resulting in a DARQN. "Attention" gives a network the ability to choose which part of its next input to focus on and allowed DARQN to beat both DQN and DRQN

on games which require long-term planning, Sorokin et al. (2015). However, the DQN outperformed DRQN and DARQN on games requiring quick reactions (like for example the “Deadly corridor” scenario), where Q-values can fluctuate rapidly. Therefore, we don’t test this technique in this project.

Lample et al. (2016) have demonstrated the efficiency of DRQN when enhanced with games information like “presence of monster” flags at training time.

3.6 Hierarchical DQN (h-DQN)

Hierarchical Reinforcement Learning is a way to learn, plan and represent knowledge with a spatio-temporal abstraction at multiple levels. This is an approach for issues of sparse rewards and/or long horizons.

Kulkarni et al. (2016) proposed h-DQN by organizing goal-driven deep RL modules hierarchically to work at different time scales. H-DQN integrates a “top-level action” value function and a “lower level action” value function: the former learns a policy over intrinsic sub-goals or “options”; the latter learns a policy over raw actions to satisfy given sub-goals. In our specific case of the “deadly corridor” scenario, the top-level goal is “reaching the green jacket” that gets satisfied by sub-goals “navigate through the corridor” and “locate/shoot monsters on the way (to survive)”.

Kulkarni et al. (2016) have reported that h-DQN outperformed DQN and A3C in Montezuma’s Revenge Atari game.

3.7 Direct Future Prediction (DFP)

Dosovitskiy et al. (2017) from Intel propose an innovative approach called “Direct Future Prediction” to this sensorimotor control challenge. Instead of a monolithic state and scalar reward, they consider a stream of sensory inputs {s(t)} and a stream of measurements {m(t)}.

As depicted on Figure 1 of their publication, the image s, measurements m and goal g are first processed by three input models, whose outputs are concatenated into a joint representation j. This joint representation j is processed by two parallel streams that predict the expected measurement E(j) and the normalized action-conditional differences: these are re-combined to produce the final prediction of each action.

The two significant benefits of this approach are:

- instead of an occasional scalar reward used in traditional RL, the measurement stream provides rich and temporally dense supervisions that can stabilize and accelerate training. A multidimensional stream of sensations is a more appropriate model for an organism that is learning to function in an immersive environment
- it supports training without a fixed goal and pursuing dynamically specified goals at test time. If the goal can be expressed in terms of future measurements, the model can be trained to take the goal into account in its prediction of the future. At test time, the agent can predict future measurements given its current sensory inputs, measurement and goal, and then simply select the action that best suits its present goal

This technique is not RL per se but a Supervised Learning approach that can generalize across environments and goals, as proven during the latest Visual Doom AI Competition where it worked well in previously unseen environments. This approach is impressive and worth further investigations:
https://www.youtube.com/watch?v=rPKwMWFo7Nk

In the next chapter, we implement “Dueling Double DQN”, the combination of Dueling DQN and of Double DQN which should outperform the simple DQN architecture because addressing the issue of q-value overestimations and of single scalar reward.

4. Dueling Double-DQN Implementation

4.1 DDQN pseudo-code, implemented code & recommendations

Here is the pseudo-code implementing DDN, adapted from Wang et al. (2016)

Pseudo-code

Init: Replay Memory (RM); Θ and Θ^- (=copy of Θ); batch_size; N^- = target network replacement frequency
For epoch in range (epochs)
    init game
    for t in range(learning_steps_per_epoch)
        get frame from game = state s, make action a, receive reward r
        get next frame = state s', store transition {s, a, s', r} (replace old transition if RM-size reached)
        sample minibatch (of size batch_size) of transitions from RM
        construct target values, one for each of the elements in the minibatch
        $a^{max}(s', \Theta) = argmax_{a'} Q(s', a', \Theta)$
        y_j = either "r" if s' is terminal, or "$r + \gamma * Q(s', a^{max}(s', \Theta), \Theta^-)$" (Bellman)
        gradient descent step with loss $|| y_j - Q(s, a; \Theta) ||^2$
        replace target parameters Θ^- with Θ every N^- steps
    end
end

Here are commented extracts (using pseudo-code notations) of the code implementing this algorithm.

Implementation

In my implementation and tests, I have adapted the files from Michal Kempka available on GitHub:
https://github.com/mihahauke/deep_rl_vizdoom

The DDQN is implemented in the file "_dqn_algo.py", here is a commented extract of the function train():

```
while self._epoch <= self._epochs:                    ### the loop on epochs
    self.doom_wrapper.reset()                         ### init game & scores
    train_scores = []
    test_scores = []
    train_start_time = time()
                                                      ### the loop on learning_steps
    for _ in trange(self.train_steps_per_epoch, desc="Training, epoch {}".format(self._epoch),
                    leave=False, disable=not self.enable_progress_bar, file=sys.stdout):
      self.steps += 1
      s1 = self.doom_wrapper.get_current_state()                    ### state s

      if random() <= self.get_current_epsilon():          ### exploration at the beginning
        action_frameskip_index = randint(0, self.actions_num * len(self.frameskips) - 1)
        action_index, frameskip = self.get_action_and_frameskip(action_frameskip_index)
      else:
        action_frameskip_index = self.network.get_action(session, s1)    ### exploitation as epsilon decays
        action_index, frameskip = self.get_action_and_frameskip(action_frameskip_index)

      reward = self.doom_wrapper.make_action(action_index, frameskip)       ### reward r from action a
      terminal = self.doom_wrapper.is_terminal()
      s2 = self.doom_wrapper.get_current_state()                            ### state s'
                                                                ### store transition {s, a, s', r} in RM
      self.replay_memory.add_transition(s1, action_frameskip_index, s2, reward, terminal)

      if self.steps % self.update_pattern[0] == 0:
        for _ in range(self.update_pattern[1]):             ### train minibatch selected by RM.get_sample()
          self.network.train_batch(session, self.replay_memory.get_sample())
```

```
            if terminal:
                train_scores.append(self.doom_wrapper.get_total_reward()) ### total reward R
                self.doom_wrapper.reset()
            if self.steps % self.frozen_steps == 0:
                self.network.update_target_network(session)  ### θ⁻ network parameters update with θ parameters
```

network.get_action() and network_update_target_network() are implemented in “dqn.py” (“networks” directory on GitHub)

Commented extract from network.get_action():

```
def get_action(self, sess, state)
        feed_dict = {self.vars.state_img: [state[0]]}
        if self.use_misc:
                feed_dict[self.vars.state_misc] = [state[1]]
        return sess.run(self.ops.best_action, feed_dict=feed_dict)
```

…where ops.best_action is calculated in _prepare_ops.

Commented extract from _prepare_ops():

```
def _prepare_ops(self):
    frozen_name_scope = self._name_scope + "/frozen"

    with arg_scope([layers.conv2d], activation_fn=self.activation_fn, data_format="NCHW"), \
         arg_scope([layers.fully_connected], activation_fn=self.activation_fn):

        ### creation of the convolution/flatten/fully-connected layers for network θ
        q = self.create_architecture(self.vars.state_img, self.vars.state_misc, name_scope=self._name_scope)

        ### creation of the (same) convolution/flatten/fully-connected layers for network θ⁻
        q2_frozen = self.create_architecture(self.vars.state2_img, self.vars.state2_misc,
name_scope=frozen_name_scope)

        if self.double:                         ### true for our Double-DQN case
            ### creation of the convolution/flatten/fully-connected layers for the network θ
            q2 = self.create_architecture(self.vars.state2_img, self.vars.state2_misc,
name_scope=self.name_scope, reuse=True)      ### reuse=True, network θ

### Important note on reuse =True
Here we want to have a network which first takes “s” as an input and produces q. Then we want this SAME
exact network (=the same instance) to take “s2” as an input and produce q2. Broadly spoken, we have a
function f(x) where f is compiled, and x is a placeholder. In our case, we need to run f(x1) for “s” and f(x2) for
“s2” separately to realize the wanted function g defined by : g( f(x), f(x2) ): this is why we set “reuse” to True.

            best_a2 = tf.argmax(q2, axis=1)   ### q2 is used to evaluate the best action on s', using θ
                                              ### => a^max(s', Θ) = argmax_a' Q(s', a', Θ)

            best_q2 = gather_2d(q2_frozen, best_a2)   ### q2_frozen is used to evaluate best_q2 with the θ⁻
            ### network, the best q-value from q2_frozen for the action best_a2. => Q(s', a^max(s', Θ), Θ-)
        else:
            best_q2 = tf.reduce_max(q2_frozen, axis=1)
                                                        ### target_q = Bellman (best_q2)
    target_q = self.vars.r + (1 - tf.to_float(self.vars.terminal)) * self.gamma * best_q2

    target_q = tf.stop_gradient(target_q)
    tf.stop_gradient(target_q)              ### target_q not to be taken into account to compute gradient
    active_q = gather_2d(q, self.vars.a)    ### q is used to evaluate active_q, for state s

    loss = 0.5 * tf.reduce_mean((target_q - active_q) ** 2)   ### gradient descent on (target_q - active_q)
```

```
        self.ops.best_action = tf.argmax(q, axis=1)[0]          ### calculation of  best_action using θ
        self.ops.train_batch = self.optimizer.minimize(loss)

        # Network freezing
        ### collects parameters θ
        net_params = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope=self._name_scope)

      ### collects parameters θ-
        target_net_params = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope=frozen_name_scope)

        ### assigns θ- parameters with θ parameters, into unfreeze_ops
        unfreeze_ops = [tf.assign(dst_var, src_var) for src_var, dst_var in zip(net_params, target_net_params)]
self.ops.unfreeze = tf.group(*unfreeze_ops, name="unfreeze")
```

Commented extract from network.update_target_network():

```
def update_target_network(self, session):
    session.run(self.ops.unfreeze)    ### this is the activation of ops.unfreeze which contains the parameters of θ
```

Note to the attentive reader: you have seen in these functions that we don't only give "state_img" or "state2_img" as argument (the states s and s'), but also "state_misc" and "state2_misc"; these are game variables like AMMO (ammunition) or HEALTH we propagate into the network when made available by the game. if use_misc is True and the scenario config file has game variables available then the inputs consist of an image and a misc vector. The input from the vector is merged with flattened output from the conv layers. If there is no game variable available, the misc is just ignored. In the following tests, we have set use_misc to False to allow a fair comparison with past tests with the standard DQN.

Recommendation for CPU users

The python files we can find here https://github.com/mihahauke/deep_rl_vizdoom are ready-to-use for GPU users. For CPU users, you need to change the following:

- in the file "dqn.py" (networks directory): change "NCHW" to "NHWC" for the data format in the "_prepare_ops()" function:
  - N refers to the number of images in a batch, H refers to the number of pixels in the vertical (height) dimension, W refers to the number of pixels in the horizontal (width) dimension and C refers to the channels
  - NCHW ("channels first") enables optimal performance on GPU
  - Only NHWC ("channels last") is supported on CPU
- In the file "_train_test.py", you need to set "config.gpu_options.allow_growth" to False for CPUs

Directions for use

After having cloned/unpacked the zip files from https://github.com/mihahauke/deep_rl_vizdoom go the directory and:

- to train the model for the scenario "basic":
  - *your_dir*$ ./train_dqn.py -s settings/examples/basic.yml
- to test the model
  - *your_dir*$ ./test_dqn.py model/basic/DuellingDQNNet/01.28_09-18
- to visualize the training and testing scores with tensorboard
  - *your*_dir$tensorboard --logdir=tensorboard_logs

4.2 Dueling architecture and implementation

Architecture

See Fig. 18

Implementation

Here again, my implementation and tests are based on the excellent work done by Michal Kempka available on GitHub: https://github.com/mihahauke/deep_rl_vizdoom

The Dueling DQN is implemented in the file "dqn.py" ("networks" directory), here is a commented extract of the class DuelingDQNNet(DQNNet):

```
class DuelingDQNNet(DQNNet):
    def __init__(self, *args, **kwargs):
        super(DuelingDQNNet, self).__init__(*args, **kwargs)

    def _get_name_scope(self):
        return "dueling_dqn"

    def create_architecture(self, img_input, misc_input, name_scope, reuse=False, **specs):
        with arg_scope([layers.conv2d, layers.fully_connected], reuse=reuse), \
             arg_scope([], reuse=reuse):

            ### this inputs layer calls get_input_layers() which returns the flattened convolutional layers
            fc_input = self.get_input_layers(img_input, misc_input, name_scope)

            ### fully connected layer with fc_units_num nodes (128 or 256), right after the convolutional layers
            fc1 = layers.fully_connected(fc_input, num_outputs=self.fc_units_num, scope=name_scope + "/fc1")

            ### the value function with 256 nodes, right after fc1
            fc2_value = layers.fully_connected(fc1, num_outputs=256, scope=name_scope + "/fc2_value")
            ### the value function itself with one output, the effective value
            value = layers.linear(fc2_value, num_outputs=1, scope=name_scope + "/fc3_value")

            ### the advantage function with 256 out, right after fc1
            fc2_advantage = layers.fully_connected(fc1, num_outputs=256, scope=name_scope + "/fc2_advantage")


            ### the advantage itself the q-value as output for each of the possible ations (actions_num)
            advantage = layers.linear(fc2_advantage, num_outputs=self.actions_num, scope=name_scope +
"/fc3_advantage")

            mean_advantage = tf.reshape(tf.reduce_mean(advantage, axis=1), (-1, 1))
            q_op = advantage + value - mean_advantage ### the combination of value and advantage
### => Q(s, a; θ, α, β) = V(s; θ, β) + [ A(s,a; θ, α) – (1/A) * Σa' A(s, a'; θ, α) ] from §3.3
return q_op
```

Here, the neural network architecture has been adapted to have "value" and "advantage" parallel streams.

Metrics

For the upcoming tests, we use only training and testing scores as metrics. We have seen from chapter 2 that the loss function was not indicative of the skills effectively learnt by the agent.

4.3 Results

Settings

To be in the position to compare the results with chapter 2, the standard DQN, I have updated the files from Michal Kempka to allow comparisons: I have reduced the resolution to RES_320x240. I have kept his "epsilon decaying" function which is closed to the "more exploration" decay function I used in the previous runs, exp(-0.01*epoch). Here are the settings common to all tests in this chapter with explanations.

Common Settings

```
activation_fn:              tf.nn.relu      # all convolutional layers have a relu activation function
batchsize:                  64              # same batch_size as before
double:                     True            # this is for double DQN
constant_learning_rate:     True            # greedy policy
dynamic_frameskips:         None            # not used
initial_epsilon:            1.0             # epsilon decay
epsilon_decay_start_step:   0               # epsilon decay
epsilon_decay_steps:        200000.0        # epsilon decay
final_epsilon:              0.1             # epsilon decay
initial_learning_rate:      0.00025         # epsilon decay
learning_rate_decay_steps:  100000000.0     # epsilon decay
final_learning_rate:        1e-07           # epsilon decay
frozen_steps:               5000            # this is the number of steps during which we freeze the network;
frozen_steps should be bigger than learning_steps_per_epoch in order not to test the network from the last
epoch. Values between 5.000 and 20.000 are fine.
gamma:                      0.99            # the learning rate in the Bellman equation, same as before
init_bias:                  0.1             # biases of convolutional layers initiated to 0.1 (same as before)
input_n_last_actions :      False           # not used
memory_capacity:            10000           # same as before
network_class:              DuelingDQNNet   # the dueling architecture
prioritized_memory:         False           # not used (and not implemented)
resolution:                 [40, 30]        # same as before
reward_scale:               0.01
rmsprop:
    decay:                  0.99            # same as before
    epsilon:                1e-10           # same as before
    momentum:               0.0             # same as before
test_episodes_per_epoch:    100             # same as before
train_steps_per_epoch:      2000            # same as before
update_pattern:             [4, 4]          # pattern for network updates in “double” DQN
use_misc:                   False           # as explained above, we don’t use use_misc
vizdoom_resolution:         RES_320X240     # same as before
write_summaries:            True            # for tensorboard
writer_flush_secs:          120             # for tensorboard
writer_max_queue:           10              # for tensorboard
```

Scenario “Basic”

RUN # 12

```
config_file:                basic.cfg
conv_filters_num:           [32]            # 1 convolutional layer with 32 filters of
conv_filters_sizes:         [4]             # kernel size 4, and
conv_strides:               [2]             # stride length 2
fc_units_num:               128             # number of nodes of the fully connected layer
epochs:                     80
```

```
Epoch 1
Training steps: 2000, epsilon: 0.991
TRAIN: Episodes: 56, mean: -1.041±2.02, min: -3.900, max: 0.950, Speed: 30 STEPS/s, 0.11M STEPS/hour, time: 1m 6s
TEST: Episodes: 100, mean: -1.516±2.39, min: -4.050, max: 0.910, Speed: 251 STEPS/s, 0.90M STEPS/hour, time: 15s
Writing summaries.
Saving model to: models/basic/dqn_basic/DuelingDQNNet/01.25_23-41/model
Total elapsed time: 1m 23s
[...]
Epoch 80
Training steps: 160000, epsilon: 0.28
TRAIN: Episodes: 103, mean: 0.080±0.99, min: -3.800, max: 0.950, Speed: 17 STEPS/s, 0.06M STEPS/hour, time: 1m 59s
TEST: Episodes: 100, mean: 0.309±0.92, min: -3.750, max: 0.950, Speed: 230 STEPS/s, 0.83M STEPS/hour, time: 6s
Writing summaries.
Saving model to: models/basic/dqn_basic/DuelingDQNNet/01.25_23-41/model
Total elapsed time: 2h 33m 26s
```

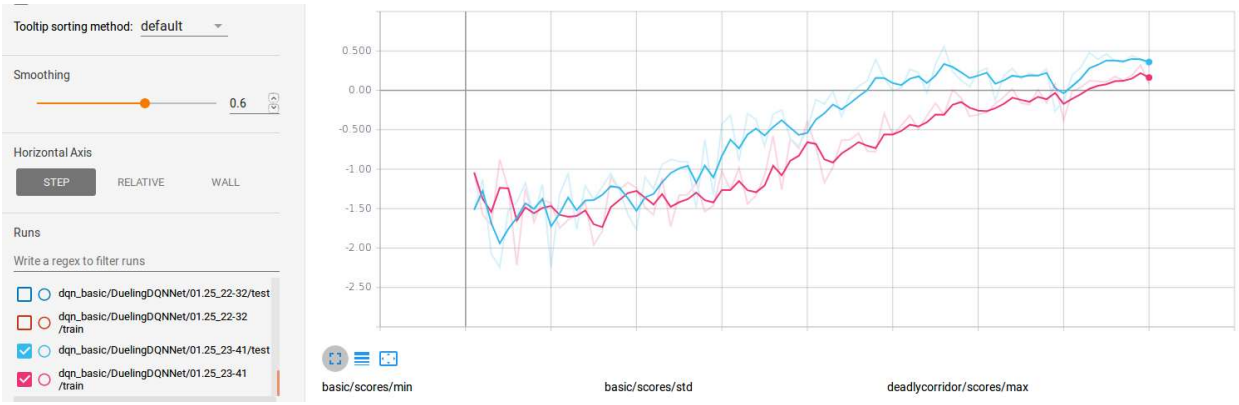


Figure 19: RUN #12

As expected, we have a model that works well in this fully-observable scenario, even with a single convolutional layer. This is confirmed by both the testing score approaching the max possible score (and largely beating the random benchmark) on Fig. 19 as well as visually with the demo video: https://youtu.be/Ny4008kaqL0

Let's now check the more challenging scenarios.

Scenario "Health gathering supreme"

RUN # 13

config_file:	health_gathering_supreme.cfg	
conv_filters_num:	[32]	# 1 convolutional layer with 32 filters of
conv_filters_sizes:	[4]	# kernel size 4, and
conv_strides:	[2]	# stride length 2
fc_units_num:	128	# number of nodes of the fully connected layer
epochs:	80	

Epoch 1
Training steps: 2000, epsilon: 0.991
TRAIN: Episodes: 46, mean: 3.230±0.80, min: 1.220, max: 6.040, Speed: 29 STEPS/s, 0.10M STEPS/hour, time: 1m 8s
TEST: Episodes: 100, mean: 3.192±0.69, min: 1.560, max: 6.040, Speed: 219 STEPS/s, 0.79M STEPS/hour, time: 19s
Writing summaries.
Saving model to: models/healthgatheringsupreme/hgs/DuelingDQNNet/01.26_16-06/model
Total elapsed time: 1m 30s
[…]
Epoch 80
Training steps: 160000, epsilon: 0.28
TRAIN: Episodes: 45, mean: 3.366±0.91, min: 1.560, max: 5.080, Speed: 15 STEPS/s, 0.06M STEPS/hour, time: 2m 9s
TEST: Episodes: 100, mean: 3.625±1.18, min: 1.560, max: 8.280, Speed: 199 STEPS/s, 0.72M STEPS/hour, time: 23s
Writing summaries.
Saving model to: models/healthgatheringsupreme/hgs/DuelingDQNNet/01.26_16-06/model
Total elapsed time: 2h 55m 7s

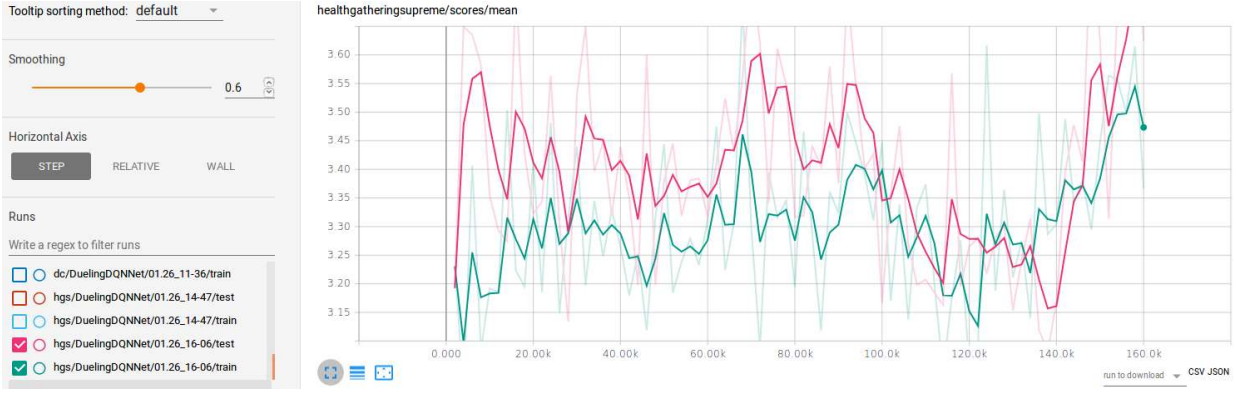


Figure 20: RUN #13

Fig. 20, shows a testing score plateauing around 360 where the maximum score is 1200, only slightly beating the random benchmark of +300 and far away from a successful model. Visual verification also shows that navigation has been hardly learnt and collecting poison vials takes place (by chance).

Let's now increase the number of epochs and the complexity of the network to see if converging takes place with these new conditions.

RUN # 14

```
config_file:        health_gathering_supreme.cfg
conv_filters_num:   [32, 64]      # 2 convolutional layers with 32 and 64 filters of
conv_filters_sizes: [8, 4]        # kernel sizes 8 and 4, and
conv_strides:       [4, 2]        # stride lengths 4 and 2
fc_units_num:       256           # number of nodes of the fully connected layer
epochs:             120
```

```
Epoch 1
Training steps: 2000, epsilon: 0.991
TRAIN: Episodes: 44, mean: 3.421±1.05, min: 1.560, max: 7.640, Speed: 13 STEPS/s, 0.05M STEPS/hour, time: 2m 34s
TEST: Episodes: 100, mean: 3.246±1.00, min: 1.560, max: 7.320, Speed: 204 STEPS/s, 0.74M STEPS/hour, time: 20s
Writing summaries.
Saving model to: models/healthgatheringsupreme/hgs/DuelingDQNNet/01.27_18-13/model
Total elapsed time: 2m 56s
[…]
Epoch 120
Training steps: 240000, epsilon: 0.1
TRAIN: Episodes: 47, mean: 3.130±0.68, min: 1.560, max: 5.080, Speed: 1 STEPS/s, 0.00M STEPS/hour, time: 41m 40s
TEST: Episodes: 100, mean: 3.010±0.54, min: 1.560, max: 5.080, Speed: 2 STEPS/s, 0.01M STEPS/hour, time: 31m 43s
Writing summaries.
Saving model to: models/healthgatheringsupreme/hgs/DuelingDQNNet/01.27_18-13/model
Total elapsed time: 7h 47m 33s
```

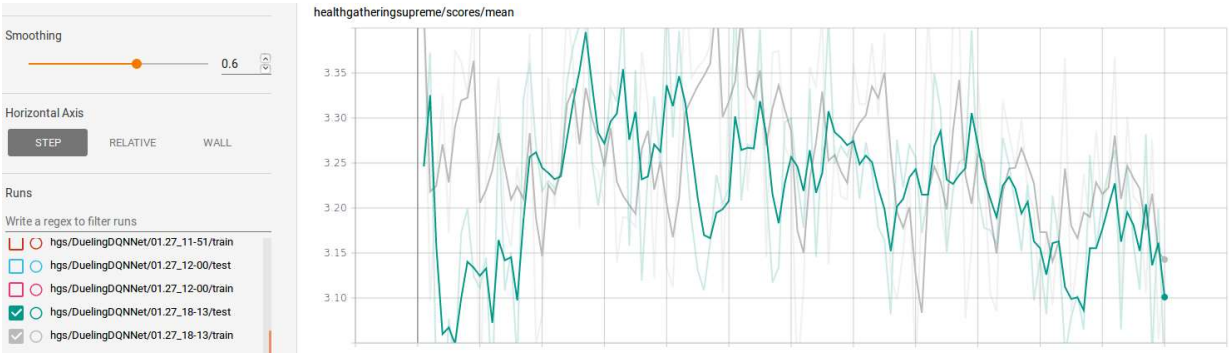


Figure 21: RUN #14

Fig. 21 shows us that increasing the complexity of the network and the number of epochs doesn't lead to convergence: the testing score stays around 340, slightly above the random benchmark but far away from the maximum 1200. The demo video confirms the inefficiency of this model which has hardly learnt navigation: https://youtu.be/nP39eapBCq4

At this stage we can see that dueling double-DQN doesn't solve this complex scenario. We can imagine (but not really confirm) that the limitations "temporal difference" & "overestimations" are solved, but not "sparse & delayed rewards".

Two partial conclusions:

- Deep Reinforcement Learning with a standard DQN is a good technique for navigation, not for learning additional skills on the top of navigation
- The Direct Future Prediction (DFP) solution from the supervised learning field should have great chance to master this scenario by taking the game variable HEALTH as stream of sensory inputs

Scenario "Deadly corridor"

RUN # 15

```
config_file:          deadly_corridor.cfg
conv_filters_num:     [32]          # 1 convolutional layer with 32 filters of
conv_filters_sizes:   [4]           # kernel size 4, and
conv_strides:         [2]           # stride length 2
fc_units_num:         128           # number of nodes of the fully connected layer
epochs:               80
```

```
Epoch 1
Training steps: 2000, epsilon: 0.991
TRAIN: Episodes: 112, mean: -0.231±1.08, min: -1.160, max: 3.471, Speed: 27 STEPS/s, 0.10M STEPS/hour, time: 1m 13s
TEST: Episodes: 100, mean: -0.511±0.85, min: -1.160, max: 3.082, Speed: 198 STEPS/s, 0.71M STEPS/hour, time: 6s
Writing summaries.
Saving model to: models/deadlycorridor/dc/DuelingDQNNet/01.26_11-36/model
Total elapsed time: 1m 21s
[…]
Epoch 80
Training steps: 160000, epsilon: 0.28
TRAIN: Episodes: 116, mean: 1.076±1.57, min: -1.160, max: 6.983, Speed: 16 STEPS/s, 0.06M STEPS/hour, time: 2m 8s
TEST: Episodes: 100, mean: 1.561±1.45, min: -1.160, max: 5.720, Speed: 158 STEPS/s, 0.57M STEPS/hour, time: 9s
Writing summaries.
Saving model to: models/deadlycorridor/dc/DuelingDQNNet/01.26_11-36/model
Total elapsed time: 2h 40m 3s
```

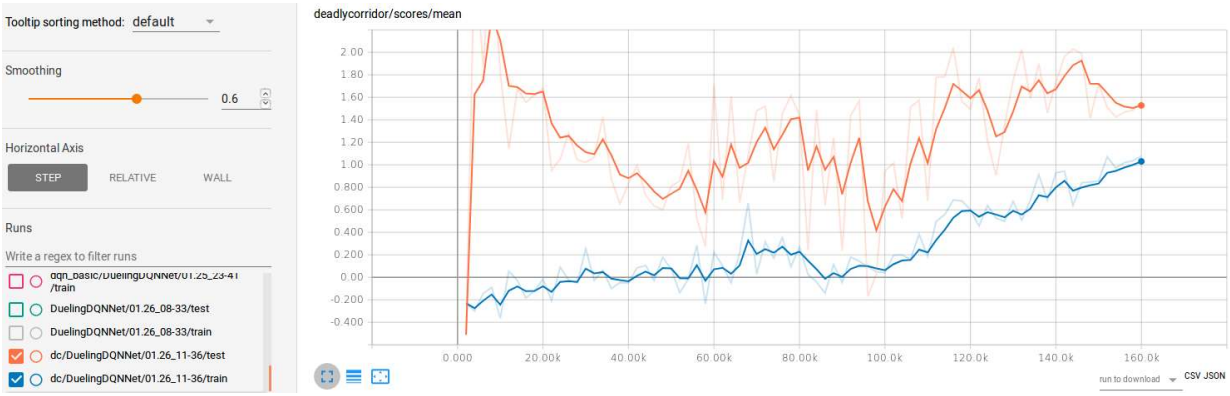


Figure 22: RUN #15

Fig. 22 shows that the testing scores improve over the epochs up to 200 points (maximum is 2300), well above the random benchmark of -115, i.e. that the agent learns. Visually, we can confirm improvements from the standard DQN with an agent which addresses the monsters on its sides. Navigation is also working well. Let's now check whether this learning trend continues over more epoch. We also increase the complexity of the network in the next run to capture more complexity of the expected behavior.

RUN # 16

```
config_file:          deadly_corridor.cfg
conv_filters_num:     [32, 64]      # 2 convolutional layers with 32 and 64 filters of
conv_filters_sizes:   [8, 4]        # kernel sizes 8 and 4, and
conv_strides:         [4, 2]        # stride lengths 4 and 2
fc_units_num:         256           # number of nodes of the fully connected layer
epochs:               120
```

```
Epoch 1
Training steps: 2000, epsilon: 0.991
TRAIN: Episodes: 107, mean: -0.290±1.07, min: -1.160, max: 3.744, Speed: 12 STEPS/s, 0.04M STEPS/hour, time: 2m 41s
TEST: Episodes: 100, mean: 0.253±0.56, min: -1.156, max: 1.449, Speed: 137 STEPS/s, 0.49M STEPS/hour, time: 5s
Writing summaries.
Saving model to: models/deadlycorridor/dc/DuelingDQNNet/01.28_09-18/model
Total elapsed time: 2m 48s
[…]
Epoch 120
Training steps: 240000, epsilon: 0.1
TRAIN: Episodes: 106, mean: 0.576±1.34, min: -1.160, max: 4.133, Speed: 9 STEPS/s, 0.03M STEPS/hour, time: 3m 36s
TEST: Episodes: 100, mean: 0.454±1.36, min: -1.160, max: 6.254, Speed: 167 STEPS/s, 0.60M STEPS/hour, time: 7s
Writing summaries.
```

Saving model to: models/deadlycorridor/dc/DuelingDQNNet/01.28_09-18/model
Total elapsed time: 7h 29m 21s

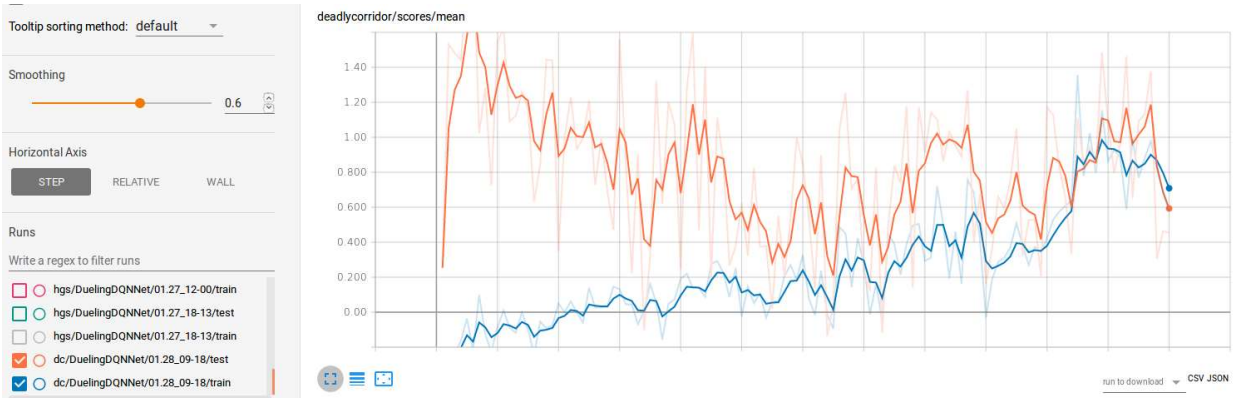


Figure 23: RUN #16

Fig. 23 shows that the testing score doesn't go above 1000, which is even worse than the RUN #15. Visually, we see that the agent hasn't learnt navigation and action skills. The demo video show for some of the tests that the agent is better than with a standard DQN, i.e. it kills the monsters one-by-one, but navigation is erratic: https://youtu.be/vw6cOEkcUn8

Conclusion

While Dueling Double-DQN beats the random benchmark in all our experiments, it is not convincing for the partially observable scenarios "Health gathering supreme" and "Deadly corridor" and increasing the complexity of the neural network doesn't solve this problem.

5. CONCLUSIONS

We have reached the end of our investigations and can conclude that:

- DQNs perform very well for scenarios where it is about learning “navigation & actions” in fully-observable MDP contexts in pseudo 3D environments like the ViZDoom environment (2D environments). DQNs beat the random benchmark and visual verification both confirm this point

- DQNs perform also well for scenarios when it is about learning “navigation without actions” in partially-observable MDP contexts in pseudo 3D environments. Again, this point is confirmed by the beaten random benchmark and visual verification

- DQNs do not perform well for scenarios with partially-observable MDPs in the ViZDoom (truly pseudo 3D) environment: while the constructed models (slightly) beat the random benchmark, learning is not stable, different skills like navigation & action don’t get learnt. DQNs either need to be complemented with further techniques to address the challenges brought by partially-observable MDPs, particularly when the task is to learn different skills, or new techniques need to replace DQNs

In this second case of partially observable MDPs, we have seen that “temporal correlation”, “sparse & delayed rewards” and “overestimation” are challenges for the agent to learn efficiently. Specifically, Replay Memory did solve the “temporal correlation” issue.

But it was not possible to fully observe the benefits of Dueling Double-DQN, especially for the most complicated scenarios “Health gathering supreme” and “deadly corridor”. The “sparse & delayed rewards” issue overshadowed them. We couldn’t observe that “overestimation” was solved and could only “guess” that the agent did learn different actions in the “deadly corridor” scenario. The benefits of dueling double DQN should be investigated again after the “sparse and delayed rewards” issue gets contained.

Future investigations should:

- implement Prioritized Experience Replay, from Schaul et al. (2016), whose benefits are to solve “sparse & delayed rewards”, and confirm if it effectively improves DQNs for such scenarios

- focus on other algorithms which have shown superior performance in various pseudo 3D environments, like for example:

    - the A3C algorithm from Mnih et al. (2016), and
    - Direct Future Prediction from Dosovitskiy et al. (2017), whose philosophy is to replace a single scalar reward with a measurement stream providing rich and temporally dense supervision, see §3.7. This supervision learning technique supports training without a fixed goal and pursues dynamically specified goals at test time and is said to generalize across environments and goals

From a broader perspective, we have seen that the act of “self-learning” has many challenges: learning from what information, in what environment, with what reward function, using what algorithm, learning what skills etc. Even more challenging is a “Learning” that generalizes across scenarios & environments we haven’t tested here. This is what is at stake for many applications currently being investigated, developed and improved for the customer service area like:

- chat bots, which are currently more “rules-based” than “self-learning”, using conversation-trees. See the comprehensive publication from Serban et al. (2017) on a DRL ChatBot

- Next Best Actions systems, where a company can proactively engage its customers upfront of any customer-initiated interaction

ACKNOWLEDGEMENTS

A **BIG Thank you!** to the **Udacity Team** for the support and for the comprehensive Machine Learning Nano Degree. This has been a great learning experience!

A **BIG Thank you!** to **Michal Kempka & Team** from the Institute of Computing Science, Poznan University of Technology, Poland for the comprehensive AI-Bot development Platform ViZDoom and for their great support: TDziękuję bardzo i tak trzymać!

REFERENCES

Arulkumaran Kai, Deisenroth Marc Peter, Brundage Miles, Bharath Anil Anthony, "A Brief Survey of Deep Reinforcement Learning", Sept 28th, 2017, http://arxiv.org/pdf/1708.05866.pdf

Dosovitskiy Alexey, Koltun Vladlen, Intel Labs, "Learning to Act by Predicting the Future", Feb 14th, 2017, https://arxiv.org/pdf/1611.01779.pdf, youtube link: https://www.youtube.com/watch?v=rPKwMWFo7Nk

Gulli Antonio, Pal Sujit, "Deep Learning with Keras", Packt, 2017

Hausknecht Matthew, Stone Peter, "Deep Recurrent Q-Learning for Partially Observable MDPs", Jan 11th, 2017, https://arxiv.org/abs/1507.06527

Kempka Michal, Wydmuch Marek, Runc Grzegorz, Toczek Jakub, Jaskowski Wojciech, "ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning", Sep 20th, 2016, https://arxiv.org/pdf/1605.02097.pdf

Kulkarni Tejas D., Narasimhan Karthik R., Saeedi Ardavan, Tenenbaum Joshua B., "Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation", May 31st, 2016, https://arxiv.org/abs/1604.06057

Lample Guillaume, Singh Chaplot Devendra, "Playing FPS Games with Deep Reinforcement Learning", Sep 18th, 2016, https://arxiv.org/pdf/1609.05521.pdf

Li Yuxi, "Deep Reinforcement Learning: An Overview", Sept 15th, 2017, http://arxiv.org/pdf/1701.07274.pdf

Mnih Volodymyr, Puigdomènech Badia Adrià, Mirza Mehdi, Graves Alex, Lillicrap Timothy P., Harley Tim, Silver David, Kavukcuoglu Koray, "Asynchronous Methods for Deep Reinforcement Learning", June 16th, 2016, https://arxiv.org/pdf/1602.01783.pdf

Schaul Tom, Quan John, Antonoglou Ioannis, Silver David, "Prioritized Experience Replay", Feb 25th, 2016, https://arxiv.org/pdf/1511.05952.pdf

Serban Iulian V., Sankar Chinnadhurai, Germain Mathieu, Zhang Saizheng, Lin Zhouhan, Subramanian Sandeep, Kim Taesup, Pieper Michael, Chandar Sarath, Ke Nan Rosemary, Rajeshwar Sai, de Brebisson Alexandre, Sotelo Jose M. R., Suhubdy Dendi, Michalski Vincent, Nguyen Alexandre, Pineau Joelle, Bengio Yoshua, "A Deep Reinforcement Learning Chatbot", Nov 5th 2017, http://arxiv.org/pdf/1709.02349.pdf

Sorokin Ivan, Seleznev Alexey, Pavlov Mikhail, Fedorov Aleksandr, Ignateva Anastasiia, "Deep Attention Recurrent Q-Network", Dec 5th, 2015, https://arxiv.org/abs/1512.01693

Sutton S. Richard, Barto G. Andrew, "Reinforcement Learning: An Introduction", 2nd Edition in progress, Nov 5th, 2017, http://www.incompleteideas.net/book/bookdraft2017nov5.pdf

van Hasselt Hado, Guez Arthur, Silver David, "Deep Reinforcement Learning with Double Q-learning", Dec 8th, 2015, https://arxiv.org/pdf/1509.06461.pdf

Wang Ziyu, Schaul Tom, Hessel Matteo, van Hasselt Hado, Lanctot Marc, de Freitas Nando, "Dueling Network Architectures for Deep Reinforcement Learning", April 5th, 2016, https://arxiv.org/pdf/1511.06581.pdf

RESOURCES

This Project

Github

- https://github.com/Rodolphe21/AI-Bot_playingDoom
- Notebook “ViZDoom Discovery”
- Notebook “Basic DQN”
- The link to Michal Kempka’s GitHub repo
- This report in pdf format

Scenario „Basic“

- Agent performs random actions: https://youtu.be/WsG09c5YHng
- Standard DQN, 1 conv layer, 40 epochs: https://youtu.be/KiaKKpqLlBU
- Dueling Double-DQN, 1 conv layer, 80 epochs: https://youtu.be/Ny4008kaqL0

Scenario “Health Gathering Supreme”

- Agent performs random actions: https://youtu.be/jxYesqMhx1Q
- Standard DQN, 1 conv layer, 40 epochs: https://youtu.be/G7LBuRdrXl0
- Dueling Double-DQN, 2 conv layers, 120 epochs: https://youtu.be/nP39eapBCq4

Scenario “Deadly corridor”

- Agent performs random actions: https://youtu.be/fGpUp8b1m-E
- Standard DQN, 1 conv layer, 40 epochs: https://youtu.be/UbtQP0CIdJY
- Dueling Double-DQN, 2 conv layers, 120 epochs: https://youtu.be/vw6cOEkcUn8

Smart Cab Project

https://github.com/Rodolphe21/Udacity_ML_nanodegree/blob/master/smartcab_RodolpheMasson_22.07.2017.ipynb

ACRONYMS

ANN	Artificial Neural Networks
CNN	Convolutional Neural Networks
DFP	Direct Future Prediction
DL	Deep Learning
DRL	Deep Reinforcement Learning
DNN	Deep Neural Networks
DDQN	Double DQN
D-DDQN	Dueling Double DQN
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
DARQN	Deep Attention Recurrent Q-Network
DRQN	Deep Recurrent Q-Network
GAN	Generative Adversarial Networks
h-DQN	Hierarchical Deep Q-Network
NBA	Next Best Action
NLU	Natural Language Understanding
PER	Prioritized Experience Replay
Q	Q-Learning, Q-Network, Quality Function (state-action-value)
RL	Reinforcement Learning
RM	Replay Memory (the agent's experience)
RNN	Recurrent Neural Networks
SR	Speech Recognition
TD	Temporal Difference
TTS	Text-to-Speech (vocalization)