DQN in ViZDoom

(by Popkov Sergei, the UEF Summer School student, the course “DRL for CG”)

Task description:

For all challenges, report/return following (One zip file with everything in it. Report in PDF format):

- All related code. You do not have to include trained models or Python libraries.
- Description of the learning algorithm, along with the hyperparameters and network sizes.
- Description of the environment (observation space, action space, reward signal, task)
- Agent's performance in the environment (e.g. a plot that shows how agent improves over time, performance of the final agent)
- Conclusion and how results could be improved / what you would try next.

Return your project submissions via email to anssk@uef.fi with title beginning with `[Summer School]`

Apply Deep Q Learning to ViZDoom environments:

- Write DQN learning code for ViZDoom (you can use Wednesday’s practicals code as template)
- Write the training loop for ViZDoom. Note that ViZDoom by default does not offer Gym API, so you do not have single convenient “step()” function.
- Evaluate the algorithm in couple of environments. You can find scenarios from the following link (you need both .cfg and .wad files for scenario). `simpler_basic.cfg` is a good starting point to debug your implementation. Try also `health_gathering.cfg` and if you have time, try `defend_the_center.cfg`.

Notes:

- Track performance by measuring the episodic reward (sum of rewards from one game)
- Neural networks do not like values with large magnitudes. If episodic rewards are too large (e.g. above 50), try rescaling rewards to be smaller.

Foreword: during the first practicals (a little bit ahead-of-time “head start”) I started to implement this project and was able to make its draft version, a proof of concept, if you will, that it is indeed possible to unify the two models from our practical work to make working (trainable) DQN model in ViZDoom environments (different configurations).

That was not enough to make a good independent product. So, I started working to improve the project in different ways during the recent days. Here’s the diary:

19.08, Monday) Corrected the code for frame preprocessing to keep it compatible with ViZDoom original format. The color formats became unified (all RGB, no GRAY) across all configurations to provide monotonous way of image representation for the neural network inputs. A special initialization python script ("init_scenarios.py") was written and successfully tested. This script provides an easy way to modify the scenarios in corresponding folder to support correct screen format for the model. It's supposed to be executed once, before the first model usage with all provided scenarios.

20.08, Tuesday) The "Failed day": tried to apply cosmetic corrections to the code, but training failed miserably after that. Kept testing half a day till found the typo in a model class. The lesson was properly learned, "if it works, don't fix it". Still, added the comments where necessary. Extracted the whole original model-related code to the outer module named "DQN_DOOM.py". Got rid of redundant dependencies and debug code. Tested it, this time it worked well. Still, not much overall progress for the project, thus the nickname ("Failed day").

21.08, Wednesday) The bunch of functions for redefined (not "vanilla") reward rules had no unified standard approach at this point. Moreover, they were embedded in the main function. To make the model more flexible, I added a first-order function parameter (for any given reward function) and rewrote these reward functions as independent modules outside of the model. That way, I was finally able to unify the model code, yet apply it to the different ViZDoom environments.

22.08, Thursday) The agent most of the time was rather unstable at the beginning. To make the training process much more effective and robust from the start, I've implemented the epsilon decay technique during aforementioned "head start" period. But it has been done in a blunt way, with hard-coded parameters, which is never good. I made special parameters that control the decay process: the value of the epsilon (at the start), speed of the decay and the decay limit. I also linked it to the result of the reward function to allow it to ignite the decay under certain reward-related conditions (vaguely called "success"). The whole code with parameters was extracted to the outer module ("CONFIG.py")

23.08, Friday) I got tired of console approach to the training/evaluating process launching, with all the keys, parameters and whatnot. I erased that part from the project and made a bunch of Python script files in a simple way instead, so I could just double-click them to get the desired result. If such script is able to find the model it's supposed to use, it runs it in evaluation mode. Otherwise, the script starts the training process, creating the model along the way. After some trial-and-error steps I was able to make pretty suitable model launchers for my solution. Also, I got rid of logging parameters. If I need logs, I redirect the output of the script or program to the appropriate file or device (like here: "python script.py > logs" or even better, something along the lines (example

for Linux OS): “python script.py 2>tensorflow_garbage.log | tee real.log”, to both see and log the important info).

26.08, Monday) Thoroughly scrutinized “matplotlib” library documentation, as I rarely used it before (preferred Octave and Excel). Included plots for “vanilla” (non-modified default) episodic reward and the function-based (possibly modified) one. Got frustrated with all these popup windows with graphics, replaced them with graphic output in SVG format (with timestamp, as it is more easy that way to find and compare the required results). The SVG file format is also more suitable for the task in a way that it provides more information about every certain point (i.e. accumulative reward per episode) due to the nature of the format.

27.08, Tuesday) Checked the results of the model training for different environments and reward functions (“vanilla” vs “modified”). Made a draft of the report: required descriptions, answers for the various questions and conclusions. Read the papers and literature to find some ideas about possible future steps to improve the model (“What should I do next?”). Thoroughly checked this report.

Model: DQN-based with decaying greedy epsilon.

Neural Network structure:

1) 2-D Convolutional Layer (16 filters, kernel size = 6, stride = 3, activation function = ReLU)
2) 2-D Convolutional Layer (16 filters, kernel size = 3, stride = 1, activation function = ReLU)
3) Flatten Layer
4) Fully-connected Layer with 32 units and ReLU activation function.
5) Output: Fully-connected Layer with number of units equal to the size of the action space, no activation function (f(x)=x), kernels are initialized with zeros, biases are initialized with ones.

Most Important Hyperparameters (not relayed to the memory, frames and other implementation details – for values of these parameters please consider looking into CONFIG.py file) **:**

Discount factor (0.99), Learning rate (0.00025), Epsion (0.9 at the training start moment), Epsilon Decay Speed (0.05), Epsilon Decay Limit (0.02)

Short General Description:

The algorithm works as follows: first, the model is trained and stored into a file. During training, the aforementioned Neural Network is used to analyze the observations (the preprocessed game frames) as inputs and calculate the Q-value and fire the appropriate action. To improve the learning speed, the Greedy Epsilon parameter was provided with reward-based decay. Epsilon parameter sets the probability of the random action of the agent. This value is fading during the training after event that set in the reward function (usually, after successful finish of the game episode or positive, in the context of the goal, action; for “vanilla” (default) rewards it’s just the episode finish). Epsilon value is decreased then by the Epsilon Decay Speed till the Epsilon has not reached the threshold set by the Epsilon Decay Limit. During the evaluation, the Epsilon value at the beginning is always equal to the Epsilon Decay Limit to force more Exploitation than Exploration from the already trained agent (to actually check the performance of the model, not some random actions). The evaluation mode is fired automatically if the corresponding model file has been found upon script launch (in this case, there’s no need to create and train the model). The program echos corresponding graphics and results of the performed training or evaluation process.

Environments (scenarios) :

1) Name: “Simpler Basic”.
Original observation frame format: 640x480, CRCGCB.
Episode timeout: 300 tics.
Vanilla reward: -1 per frame.
Action space: turn left, turn right, attack.
Additional game-related observable parameters and variables: none.
Task: shoot the monster as soon as possible.
Count of steps to train: 100 000
Count of steps to evaluate: 100

2) Name: “Health Gathering”.
Original observation frame format: 320x240, CRCGCB.
Episode timeout: 2100 tics.
Vanilla reward: 1 per frame, -100 per death.
Action space: turn left, turn right, move forward.
Additional game-related observable parameters and variables: health.
Task: survive as long as possible (using medicine packs).
Count of steps to train: 100 000
Count of steps to evaluate: 3 000

3) Name: “DTC (Defend The Center)”.
Original observation frame format: 640x480, CRCGCB.
Episode timeout: 2100 tics.
Vanilla reward: -1 per death.
Action space: turn left, turn right, attack.
Additional game-related observable parameters and variables: pistol ammo, health.
Task: survive as long as possible (eliminating as much incomers as possible before they hurt the agent).
Count of steps to train: 200 000
Count of steps to evaluate: 1 000

Additional reward parameters:

The usually large magnitudes of the rewards are indeed pretty troublesome in regards of the Neural Network training performance. Thus, the classic rewards are not satisfying for the learning (or demand much more time to reach proper quality level than scaled rewards). But I’ve observed that the scaling still is not sufficient in terms of time and overall quality of the solution. For example, the agent with the scaled reward version of the “Simpler Basic” environment sometimes could not properly react to the walls and learnt that he “needs to fire” to achieve result – much more often than walking around to properly aim at the target, apparently. If monster is spawned way too often right at the agent aim, the network may even learn to just fire from the start of the episode, regardless of the presence of the monster in the further episodes. To unlearn such behavior pattern the system needs more epochs/episodes, then... So, this part of the algorithm has been improved by making my own goals and rewards based, nevertheless, on the original task. The performance of the agent is also represented relatively to the new (modified) and original (vanilla) reward values as well. Here’s the list of the additional reward parameters (with related script names) and the reasoning behind it:

1) File: “basic_reward.py”.
Environment name: “Simpler basic”.
Additional reward-related parameters:

- FAILURE_TIME (timeout, end of the episode, in tics) = 300
- POS_REWARD (positive reward value) = 1
- NEG_REWARD (negative reward value) = -0.01

Reward function: -0.01 for every frame, +1 for finishing the episode before the timeout (killing the monster).
Epsilon decay policy: the epsilon value decreases after each episode finished before timeout.

Reason: this way, the agent is expected to learn wasting less time and ammo to solve the task.

2) File: “health_reward.py”.
Environment name: “Health Gathering”.
Additional reward-related parameters:

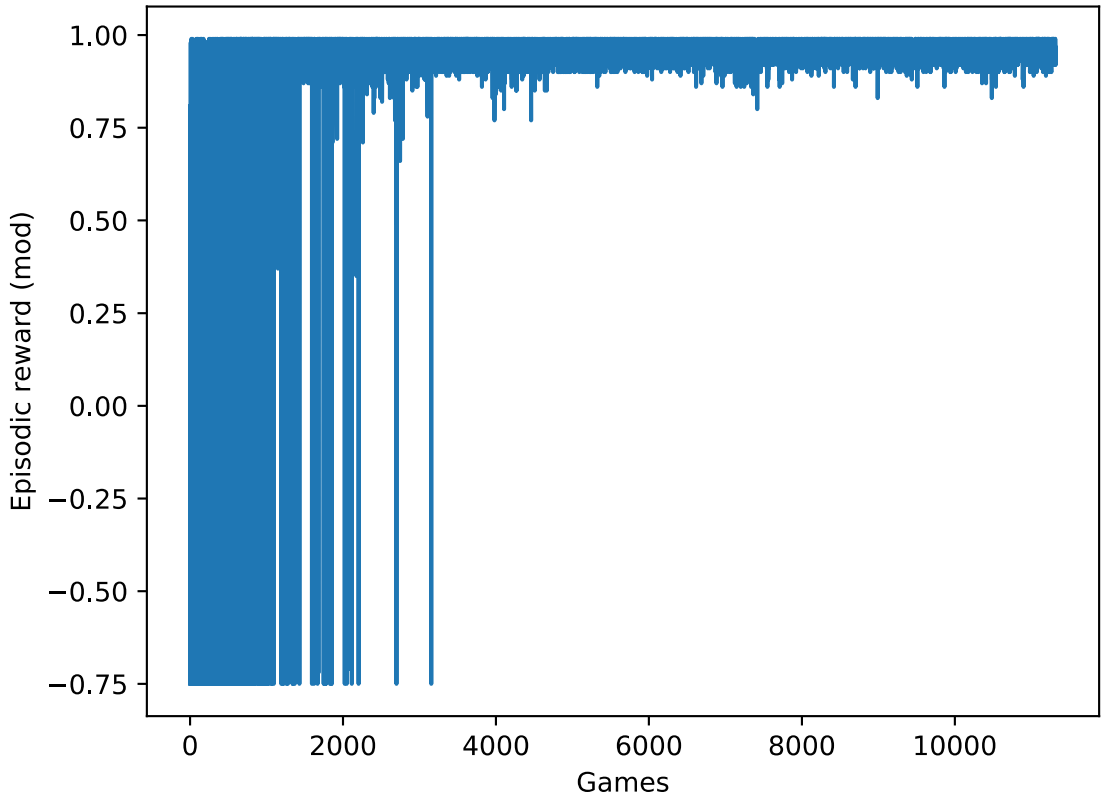
- POS_REWARD (positive reward value) = 1
- NEG_REWARD (negative reward value) = -0.01

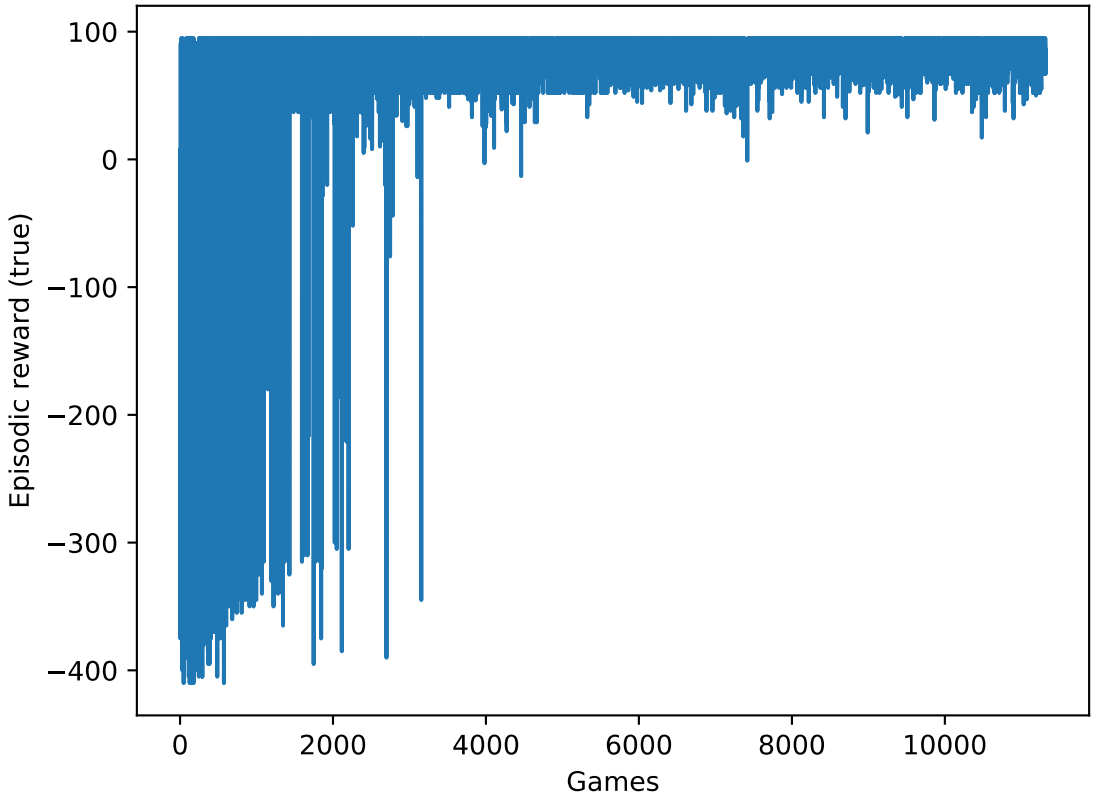
Reward function: +1 for every frame where health condition of the agent became better (more health gathered from the health packs), -0.01 for every other frame.
Epsilon decay policy: the epsilon value decreases after each episode.
Reason: such reward function provides the means to the agent to comprehend the necessity of the health packs to survive; it’s as if human being feels pain and takes a pill to make that pain disappear; these symptoms (or signals) let the brain know what to do to heal the pain.

3) File: “default_reward.py”
Environment name: “DTC”
Additional reward-related parameters: none.
Reward function: vanilla reward function.
Epsilon decay policy: the epsilon value decreases after each episode.
Reason: vanilla function works fine. I tried to invent and apply some complicated rewards, but they brought much worse performance quality than the original reward function.

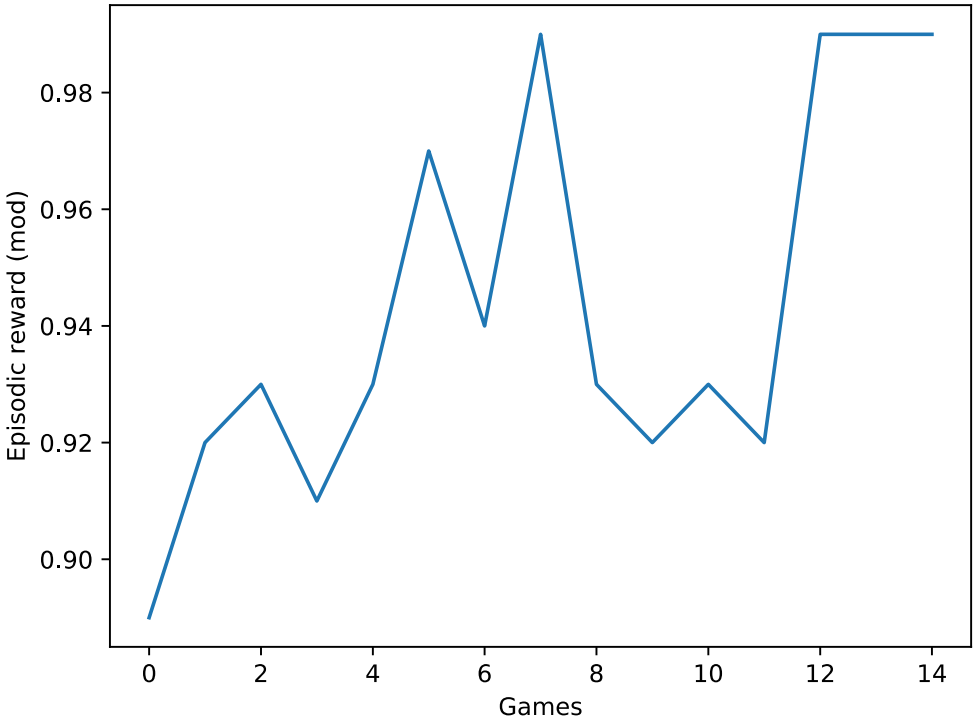
Performance of the agent (graphics):

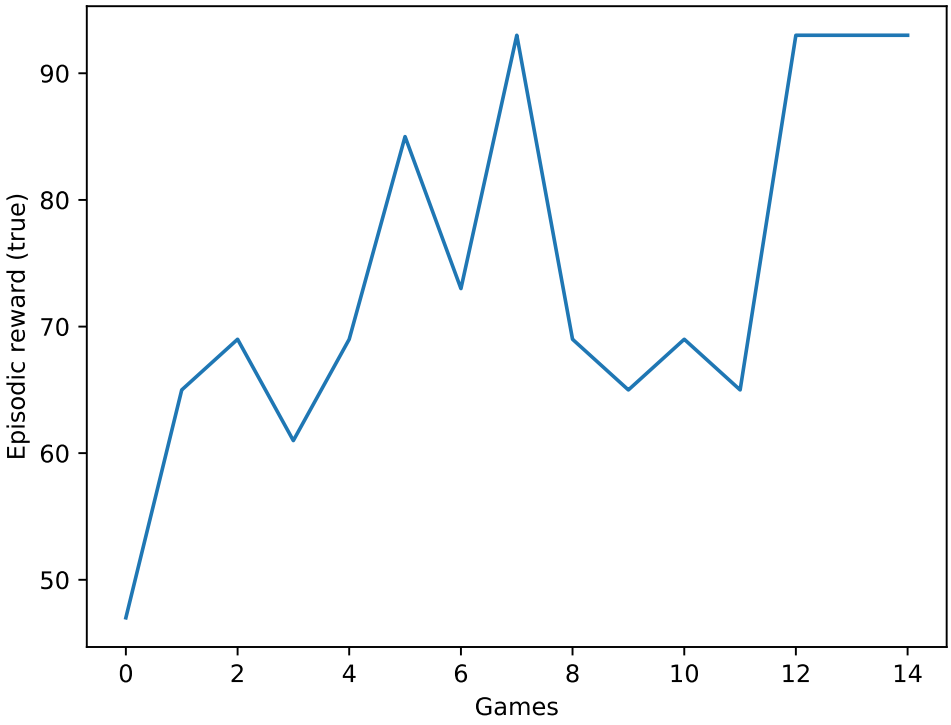
1.1) Environment “Simpler basic”, training (modified and vanilla reward, respectively):



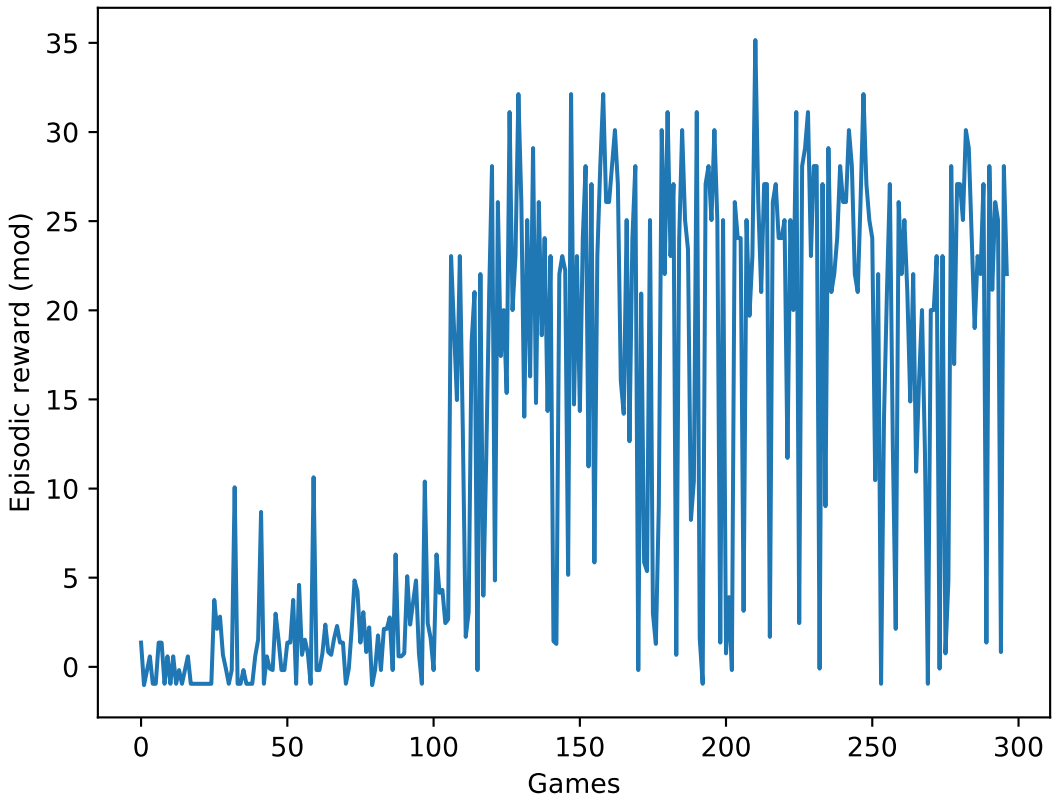


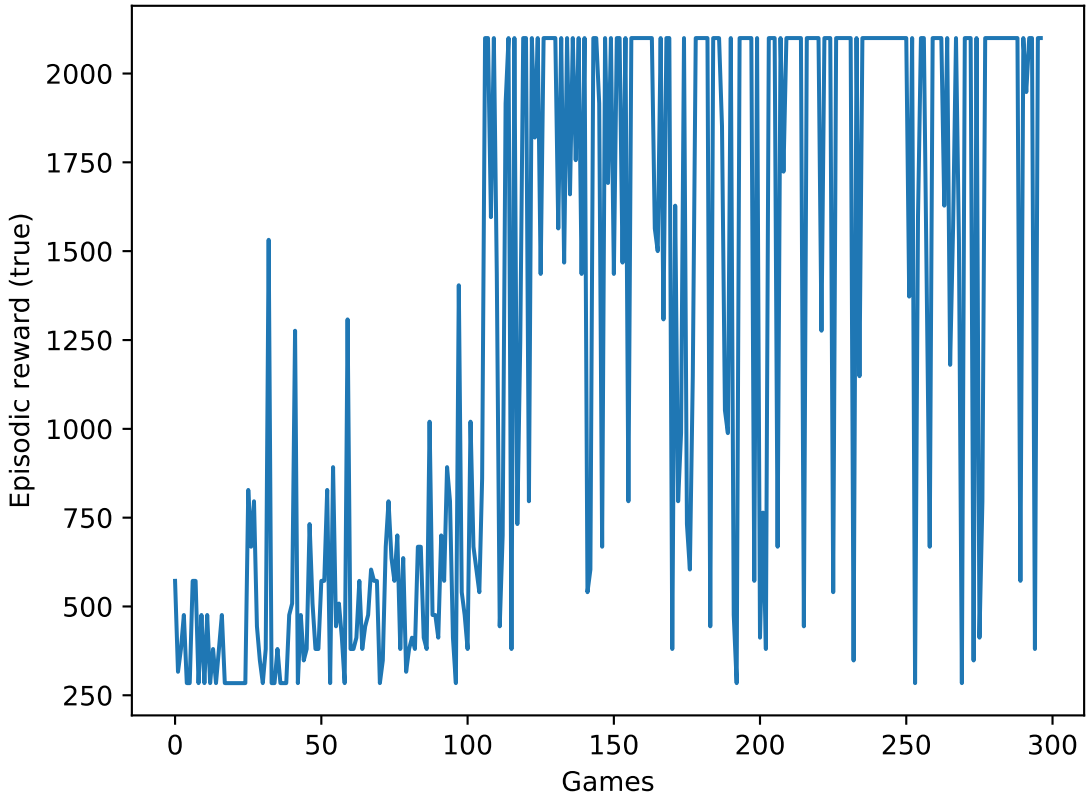
1.2) Environment “Simpler basic”, evaluation (modified and vanilla reward, respectively):



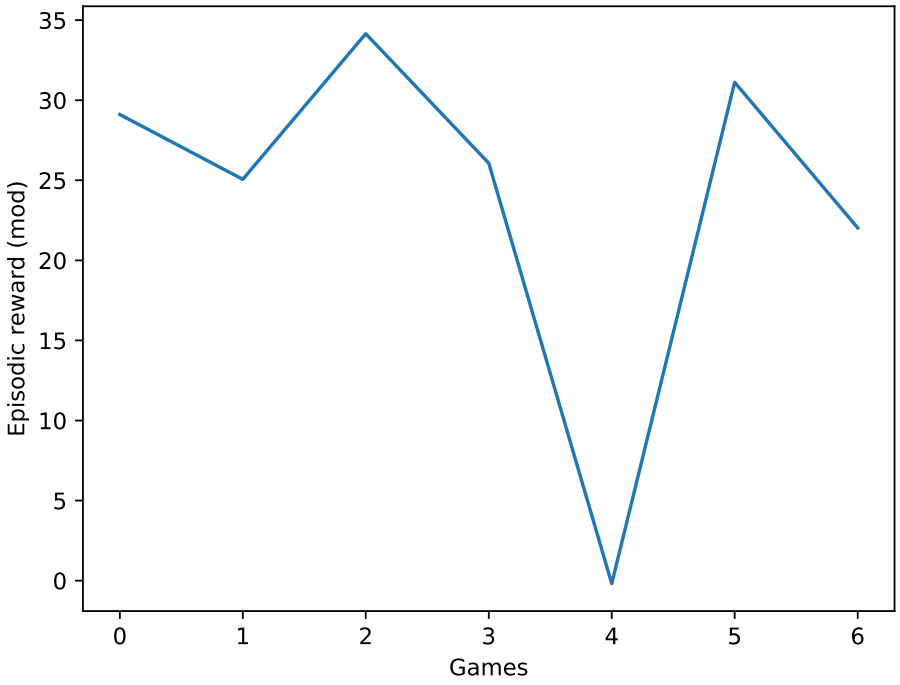


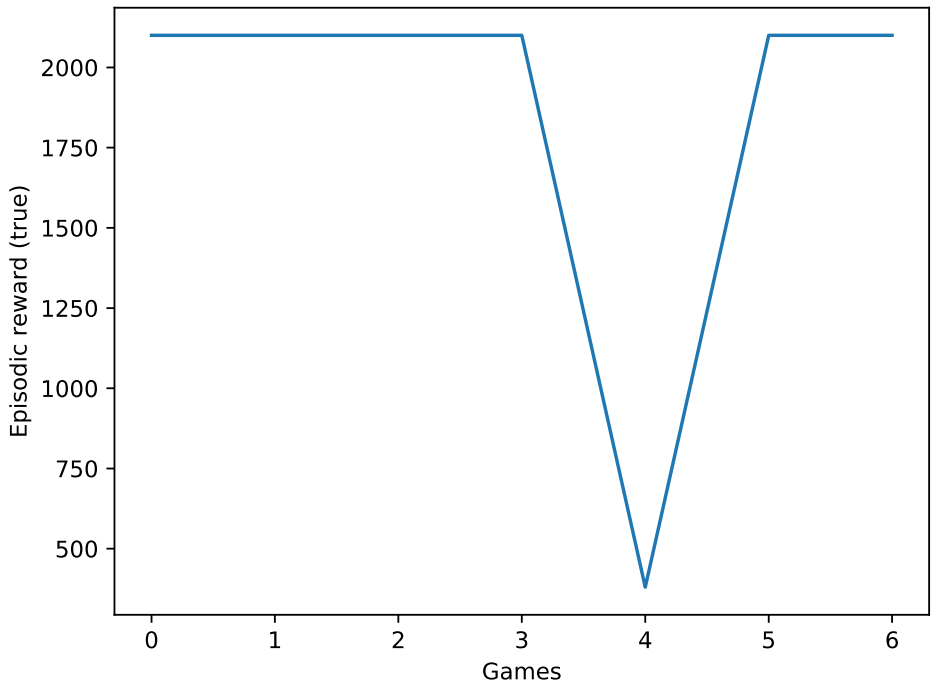
2.1) Environment “Health gathering”, training (modified and vanilla reward, respectively):


Episodic reward (mod)
35
30
25
20
15
10
5
0
0
50
100
150
200
250
300
Games

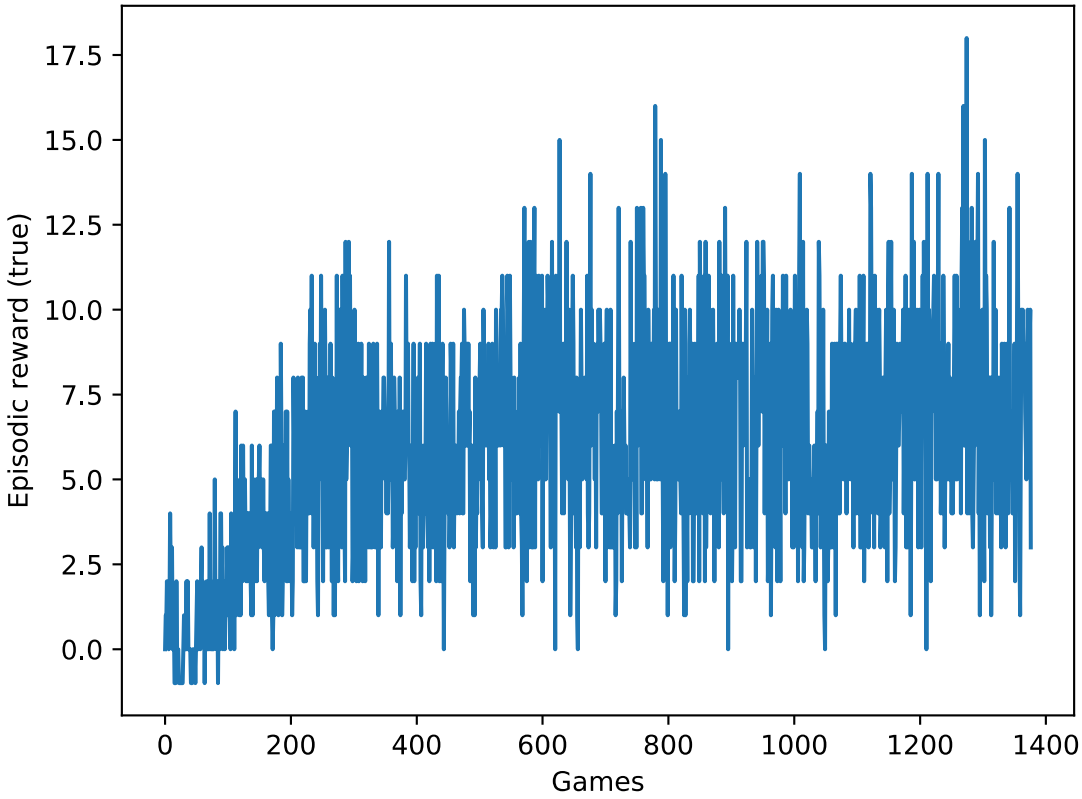


2.2) Environment “Health gathering”, evaluation (modified and vanilla reward, respectively):

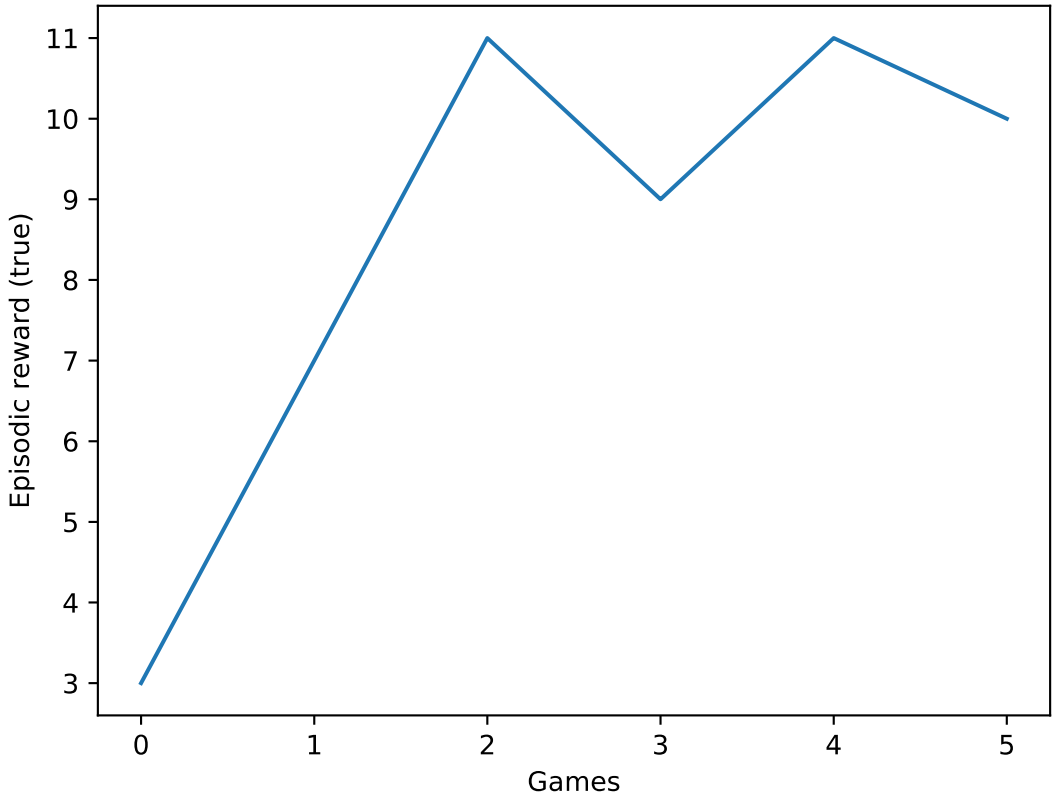




3.1) Environment “DTC”, training (vanilla reward):



3.2) Environment “DTC”, evaluation (vanilla reward):



Average final agent evaluation rewards:

1) “Simpler basic”: 0.94 (“vanilla” 73.93)
2) “Health gathering”: 23.9 (“vanilla” 1854.2857)
3) “DTC”: 8.5

Conclusion:

1) Model trained for “Simpler basic” scenario sometimes has some troubles with walls (tendency to stuck at the borders) on a rare occasion.
2) Model trained for “Health gathering” can sometimes stuck between health packs at equal distance (Buridan’s ass paradox) or in the wall corner when there’s no health packs at sight.
3) Model trained for “DTC” demands a lot more time to properly train than for other investigated scenarios.
3) Double DQN (excluded from this report, yet presented in the code archive) does not improve the situation much. Sometimes the agent become more responsive in the moments where it got stuck before, but altogether less precise in more common tasks (misfires, inaccurate touching of the health pack, etc) that at normal DQN.

4) Nonetheless, the DQN model helps training an agent, albeit at the cost of the considerable time consumption. Some papers recommend to train the model a day or two to avoid some of described problems, but my notebook becomes way too hot, so I wake up once in the middle of the night to turn it off and never actually tried to do that ever again.

What to do next:

1) I would like to learn more techniques at freeCodeCamp (https://simoninithomas.github.io/Deep_reinforcement_learning_Course/) to implement them; for example, check out the Dueling Double DQN and see if that would make the model work better without excessive (as in "time-consuming, PC-burning") training.

2) One more interesting extension to the problem: make my own map (via Doom Builder, maybe) with different places for different goals and strategies (for example, one chamber has some health packs, another is a corridor of enemies, the third one demands interaction to get certain key, etc); then, implement the hypermodel that is trying to learn, which chamber demands which strategy; and train neural networks separately for each cluster, but with one common or similar reward function (e.g., maximizing the amount of stats (health, ammo, etc) per episode). Apply the transfer learning, so these networks would train separately first, combine the result in one multi-neural network model.

Folder structure description:

./report.pdf – this file;
./init_scenarios.py – file updating the default scenarios in the "scenarios" folder to the compatible and unified (same) format representation (frame color scheme);
./CONFIG.py – file with model hyperparameters;
./DQN_DOOM.py – file describing the DQN model, the training loop and evaluation parameters;
*./*_reward.py* – file containing corresponding reward function;
./main_basic.py – entry point for "Simpler basic" scenario;
./main_center.py – entry point for "Health gathering" scenario;
./main_health.py – entry point for "DTC" scenario; each entry point contains the name of scenario, the name of related model, the training and evaluation parameters (steps, window visibility, etc) – the basic code to "click-and-play" the corresponding scenario. If the model file doesn't exist, it's generated by training; otherwise, the system runs in evaluation mode.
./run.sh – shortcut executable to launch the model easily. Usage: run.sh *model_shortname log_postfix* . "model_shortname" is the name of the launcher after

“main_” part (“basic”/”center”/”health”). “log_postfix” is used to make different log files (for example, command “./run.sh basic train” will produce “basic_train.log” file).
./graph_and_models – results (DQN and Double DQN graphics, logs and trained models).
./scenarios – updated model-compatible scenarios from ViZDoom.
./modifications/DDQN_DOOM.py – replace DQN_DOOM.py with this file to try Double DQN implementation instead of default DQN one.
./modifications/vanilla_reward – launchers with vanilla reward only, no reward modifications.