Deep Q-Learning for BattleZone

Karray Elyes, Khodabin Misha, Kobrosly Lotfi

March 2019

1 Introduction

In this paper, we present our work based on an artificially intelligent agent for the INF581 course. We used tools from the course to build the agent on a game called *BattleZone*. The goal of this game is to control a tank, which can move back and forward, rotate around its center and shoot rounds, in order to destroy enemies and get the best possible score.
We chose to use a Deep Q-Network agent that was customized for the game. The reason behind this is that the state space (possible frames) is too big, actions change the state in an unintuitive manner and we want our agent to learn the important features by itself. We implemented various techniques to optimize the training and get the best possible score.

The code can be run by downloading the contents of this github repository and running `main.py`. TensorFlow, Gym and Pickle must be installed to be able to launch it. (You can find the same files in the submission as well).

2 Background and Related Work

2.1 Choice of hyperparameters

For this project, we tried to replicate the ideas of the paper [1] by DeepMind and to adapt them to the particular environment we have. We relied on it for the choice of the graph for the model as well as for the learning hyperparameters. Here are the most important ones:

1. $\gamma = 0.99$ is the discount factor for updating the policy.
2. $\epsilon = 10^{-6}$ is the rate at which the exploration rate changes. It means that this rate takes 10^6 iterations to change from its initial to final value
3. 1 and 0.1, the start and end rates of exploration rates.
4. $n_{frames} = 2..4$, the number of frames used as an input to the network (to define the state). We set it to 2 for memory efficiency, but we may need to take it up as the model may require additional information.

2.2 Deep Q-Learning

Our problem is an instance of a Markov Decision Process, where we must teach a deterministic agent to maximize its reward in a stochastic environment. Given what was said previously, the model we use is based on Deep Q-Network (DQN). In our case and given what we previously said, our model will try to approximate the function:

$$Q^*(s,a) = max_\pi\{\mathbb{E}[\sum_{i=0}^{\infty}\gamma^i r_{t+i}, \quad s_t = s, a_t = a, \pi]\}$$

where s_t is the state, a_t is the action taken, r_t is the reward obtained by taking action a_t at the state s_t all at time t, π is the policy function indicating what actions to take given the states and is the discount factor for the future gains. This means that Q is the highest expected sum of discounted future rewards achievable by following a fixed policy.

According to the Bellman equations, we have thus:

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

where s' is the state resulting from taking action a from state s. We then update our function through iteration and aim for it converging to the following function:

$$Q(s,a) = r + \gamma \times max_{a'}(Q(s',a'))$$

where $Q(s,a)$ is the value obtained when performing the action a to obtain state s' from state s, r is the reward obtained following the transition and $max'_a(Q(s',a'))$ is the optimal value following the state a'. γ is a discount factor that will be close to but inferior to 1. The main reason for this is that we want to avoid having infinite values for the Q-function. Another reason is that we want to favor an agent that maximizes both short-term and long-term rewards.

2.3 ϵ-greedy exploration

To train our agent, we first need to make it perform random actions and get rewards later. As it becomes more "mature", we let it perform the actions that maximize the value according to its prediction. This is the ϵ-greedy exploration, where we choose the action predicted by the model with an increasing probability through time.

Practically, we do such a thing during the exploration phase by choosing a random action with the probability $p(i) = max(1 - \epsilon_i, 0.1)$ and choosing the action that gives the best Q-value according to the model with probability $1 - p(i)$, where i is the iteration number.

3 The Environment

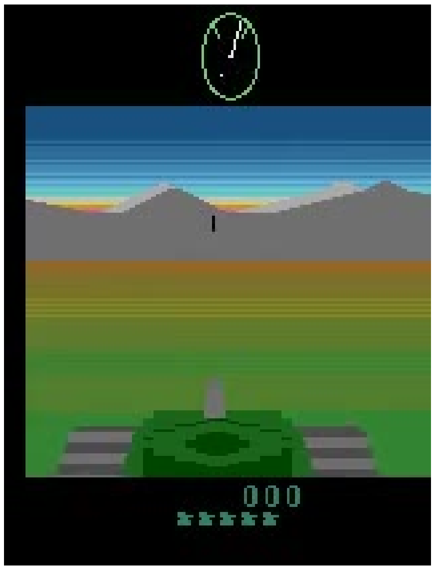
The BattleZone environment is already implemented in the OpenAI gym and has 12 versions with slight differences.We have chosen the `BattleZone-v0` environment:

- The state is defined by the frame. It contains a radar system that indicates the position of nearby enemies in a certain radius. It also contains a visual representation of fired rounds and enemies in a reduced vision field, as well as the remaining lives. We will see later that we need more than one frame to define the state.
- The reward signal will be the reward obtained by destroying an enemy. The total reward for a run will be the total score when the player dies.
- The action space is of size 18, corresponding to the joystick + button combinations for an Atari. Here is a list of actions and corresponding numbers:

Movement	No shooting	With shooting
Stay still	0	1
Forward	2	10
Right	3	11
Left	4	12
Back	5	13
Forward+right	6	14
Forward+left	7	15
Back+left	8	16
Back+right	9	17

We should keep in mind that shots are not fired every frame if we keep pressing the button, and we will aim to make our model learn how long it takes to fire another round.

- Firing rounds and moving are deterministic actions. However, spawning of enemies is random and stochastic, and enemies spawn more densely and frequently as we advance through the game. This makes it so the agent learns how to play the game in different settings and not in one where enemies spawn following the same sequence.
- This environment is more interesting than most classic 2D games, as actions change not only the position but also the orientation of the player (tank). The main challenges that the agent needs to overcome is to understand how its movements affect its position but also the positions of its enemies. It needs to understand where the enemies are, where they are facing and where they are moving from the observations. Finally, it needs to learn when is the best time to shoot, as it cannot shoot at every frame.
- The first real-world application that we can think of is a self-driving tank that makes decisions based on a visual and a radar input. While we do not wish for such a thing, we believe our agent could be used for a navigating self-driving vehicle that uses two types of inputs to understand its environment.



An observation with the various elements: the radar, the view, score and lives

4 The Agent

The aim of our agent is to choose the action for the tank to take according to the observed state in order to maximize the reward.

4.1 Preprocessing

The agent, as specified, takes the frames as input. Actually, it is not exactly like that. Each frame is divided and we take two components to provide our network with:

- The *subframe* of the radar

- The *subframe* representing the vision field (minus the mountains and the sky on top as it is not informative, and its movement may cause the agent to false conclusions and waste computational power)

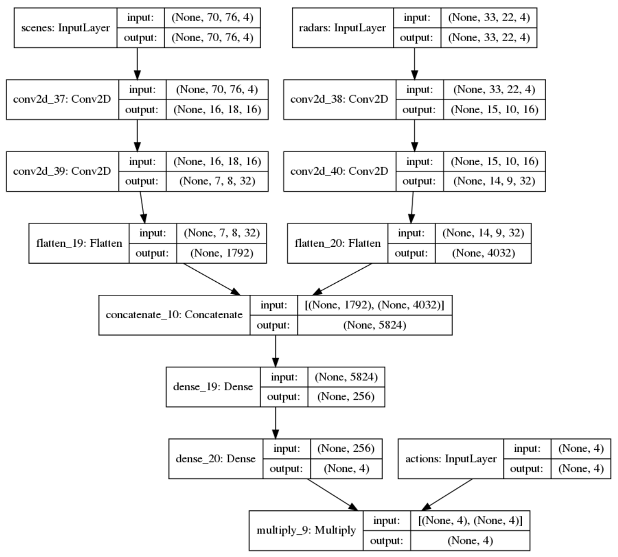
These two subframes are then transformed to grayscale, and the vision field subframe is downsampled to improve memory usage (we did not downsample the radar subframe as the resolution is already quite low, we feared it might result in a significant information loss).

One might think that all necessary information may come from the radar alone (as it contains information on the position of the tank and its enemies). However, it does not provide anything related to firing (especially if the tank had just sent a projectile, and how much time has passed since then). This is the reason that the preprocessing part gives the two subframes as output.

4.2 Architecture

Since the state space is too big (we must enumerate all possible frames), rather than a Q-table, we will use a double-pipelined Deep Q-network. The idea behind this is inspired from the DeepMind paper.

The input of our model will be the preprocessed radar and front view combined with the actions for which we want to compute the Q-values (encoded as one-hot vectors). The double-pipeline concerns the radar and front view subframes, as the first will go through fully-connected layers (represented as a vector) and the second will go through convolutional layers before fully-connected ones.



Graph of our model

4.3 Predicting with the model

To choose the next action, we input the current state as well as a few previous ones (these frames will essentially provide the model with information regarding the last time a projectile was fired) and a vector of ones for the actions in order to compute the Q-values for all the actions and choose the one that yields the highest target.

4.4 Fitting and updating

For each action a that takes us from state s to s' and gives the reward $R(s, a, s')$, we update our Q-values accordingly by setting the targets $R(s, a, s') + \gamma \ max'_a(Q(s', a'))$ for the output of our Q-network $Q(s, a)$ and performing a gradient descent (RMSProp) optimization. We summarize the above points in the following algorithm:

```
Result: Write here the result
Initialization of the Q-network with random weights;
for $i$ in $n_{iterations}$ do
    Sample $X \in \{0,1\}$ with $p(X=0) = max(1-\epsilon i, 0.1)$;
    if $X$ = 1 then
        Predict next action $a$ with the Q-function;
    else
        sample the next action randomly;
    end
    Execute the action in the emulator and observe the reward;
    Perform a gradient descent step on $Q(s,a)$ with target $r + \gamma max'_a(Q(s',a')$ (or $r$ if the action is terminal);
end
```

Algorithm 1: Exploration and learning algorithm

We made the choice of fitting using only one epoch for two main reasons:

- Making computations lighter and faster, which will enable a faster decision-making process as well as gaining more experience from real-time playing.

- Avoiding over-fitting: with the choice of a mini-batch compared to the rest of the set, there is a risk of over-fitting if the model trains too long on the same mini-batch. The goal is to learn the most experience from the largest number of memories.

4.5 Improvements performed

We encountered a few possibilities that may help our model perform better and especially prevent some convergence and undesired behaviour issues.

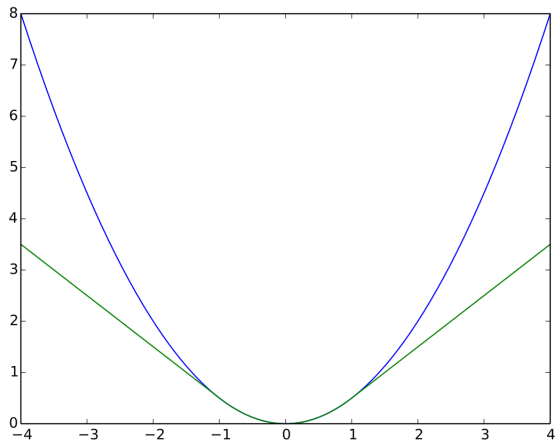
4.5.1 Establishing a target model

One of the main concerns of our approach is the possibility that our model does not converge. In fact, we are asking the network to converge to the Q-function while at the same time, the model itself is changing. This is what can be labeled as *"chasing its own tail"*.

To solve this issue, we will attempt to save a copy of the model every certain number of iterations, as to separate the evolving model (which will be oscillating in order to reach a stable value) from its target, and use this model as the desired target. This is a heuristic approach to stabilize the process and converge to a solution in a quicker way.

4.5.2 Huber loss function

The function we first used in our model is the *Mean Squared Error*. Despite being quite popular and puts emphasis on large errors in an attempt to minimize them, is not appropriate for our case. In fact, using MSE will cause the network to quickly and greatly vary while trying to converge. However, since it is trying to predict its own output, the model changing at the same time as its target might have the reverse effect. This means that this approach is also helpful in securing the goal of our first improvement.



Huber loss (green) and MSE (blue)

4.5.3 Experience Replay

The following approach can also translate into what we can call a *memory*. In fact, we provide our agent with a memory that will feed to him his past actions and states as well as the results to the actions it undertook from rewards to resulting states.

To implement this approach, we used a data structure called *Ring Buffer*: the theoretical idea is to have a ring of slots (thus with a maximum capacity, and the ring structure suggesting an ordered filling of these slots, and when a new element is to be inserted, it is either inserted in an empty place of takes the spot of the oldest memory) in which the model will store its *experience* or past plays and try to fit on that set so that it improves its decision making process for the coming rounds. This circular structure, leaving only the most recent memories, somewhat ensures that the model will only fit on what should be an improved version of itself compared to its earlier stranger and this is based on the idea that the older the memory the less likely it is to be a memory the model should remember.

4.5.4 Rewarding policy

The reward variable's goal is to incite the model to increase it as a maximization problem. However, since there are no *penalization* per Se, as "dying" will only end the game (and reset the reward to 0) but being an occasional event, its effect may be not very influential. We thus introduced another reward (negligible to the real reward though) that will try to guide the agent towards staying alive: every time the agent does an action that prevents it from dying, they get +0.1. The idea might be misleading to the agent (in an extreme case, running away from enemies will maximize the reward) but it can help him avoid risky situations and the game being over.

4.5.5 Same action for a number of frames

This idea stems from imitating the human behavior. In fact, when a human plays on atari, it is really difficult to decide an action and apply it for only one round (time of latency for human response, latency in the joystick, its sensitivity, etc.). This will get the agent a little closer from the human behavior, will make its decisions more stable and associate more coherent resulting states and rewards for certain actions starting from some other states.

5 Results and Discussion

Our model did not train for a long time, and the results are still not satisfying. It performs a certain action all the time, which does not make it effective at all.

5.1 Weaknesses

Let us try to understand the main weaknesses of our model.

- The agent seems to fail to link the reward to a certain action, since the change in score occurs many frames after it shoots. Besides, it fails to avoid terminal actions because when it receives the information, it is too late. We will aim to tackle these two issues by introducing a higher n_{frames} and making the agent perform the same action over a bigger number of frames.

- For the moment, our agent forgets the results of previous experience when playing, which means the improvement of its behaviour is still not sufficient.

- The agent still does not turn to face enemies that appear on the radar. We hope that the last two issues will be tackled by experience replay and longer training.

- Random actions rarely give rewards, so it may be interesting to play the game ourselves to give the model better episodes to train on: we could perform imitation learning.

5.2 Possible improvements and future work

- Choosing a higher batch-size for the fitting step (beware of the accuracy-efficiency ratio).

- Exploring other possibilities for the model's architecture as well as the learning approach (there may be a different and more suitable approach that the Deep Q-learning one).

- Testing other methods of optimization for the loss function

- Using already available codes to compare, analyze and draw conclusions about the issues that may face the agent and how to overcome them

- Using the RAM environment to feed the agent and improve its decision making speed (may have some other drawbacks)

- Revisiting the details of the model's architecture, whether it as the number of layers, their type, the number of filters/ hidden nodes in an attempt to improve the efficiency and probably increase the size of the input.

6 Conclusion

Our agent still needs much improvement. Its behaviour seems to have slightly improved but training needs a large amount of time and computational power to reach satisfying results. We could not afford to have the right amount of time to do this, but we are confident that our model may bear good results.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin A. Riedmiller (2013) *Playing Atari with Deep Reinforcement Learning*

`http://arxiv.org/abs/1312.5602`

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, Demis Hassabis (2015) *Human-level control through deep reinforcement learning*

`https://doi.org/10.1038/nature14236`