Léo Poulin - 017465433

CECS 406: Introduction to Machine Learning

Semester Project Final Report - Reinforcement learning on MsPacman

In a some case where learning is required without a complete understanding of the model or without the time to build a defined set of parameters and correct outputs, reinforcement learning comes in handy because it can eventually surpass problems with “real-world” complexity. It is very similar to the human cognition, it uses pasts experiences to determine an optimal policy to use in future similar situation. However, to reach that , the agent needs to deal with large multi-dimensional inputs and derive accurate representation of the environment from thoses. This project tries to demonstrate the efficiency and adaptation (learning) capability of a deep Q-learning agent on the Atari 2600’s MsPacman, using OpenAI Gym environment emulator, where it will progress through the game only by analysing full RGB pixel arrays as inputs.

Q-Learning

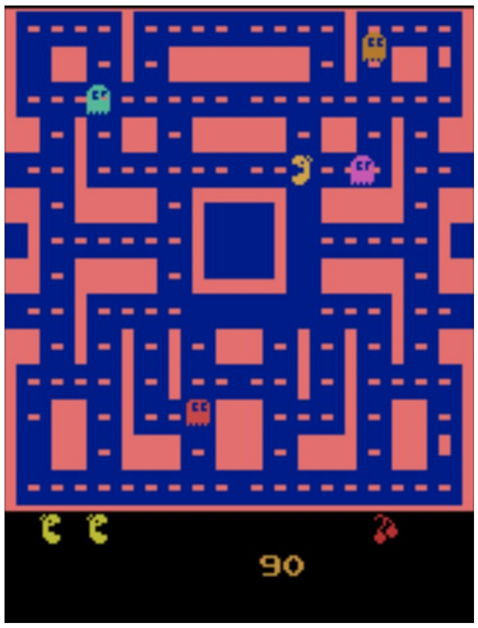
Q-Learning is a model-free learning where the agent doesn’t know the environment model, but will attempt to derive an optimal policy through trial and error using its history of interactions with this environment. Q-Learning deals with stochastics transitions and rewards; its goal is to approximate the state-action value function Q, in other terms, predict the utility of an action *a* in a state *s*. This is defined by the quality equation:

$$Q : s \times a \rightarrow \mathbb{R}$$

Our algorithm will use value-iteration to converge to the optimal state-value function; the optimal policy can be derived from the optimal value function. These iteration, or updates, relies on the Bellman equation, as follow:

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

The Q function of an action *a* in a state *s* is equal to the sum of the immediate reward and expected future reward after the transition to the next state.



Parameters and output size

The Gym library allows us to render a complete MsPacman environment, which returns us a full RGB image of size 210 (height) x 160 (width) x 3 (color depth) that represent the actual image that would be displayed by the Atari 2600 system. The environment generate a new image each time it is reset or when an action (step) in performed. The environment defines by default an action space, that represents the actual controls that are valid on this game and that could be used as action to step to the next state. The default action space presents 18 possible controls but was reduced to only 5 to avoid confusion and actually use meaningful actions, the final action space will be ['NOOP', 'UP', 'RIGHT', 'LEFT', 'DOWN']. Our neural network will process the input and estimate the quality of each of those actions in the current state by resulting an array of shape *(1, action_space)* containing the Q-values of each of those actions..

Exploration

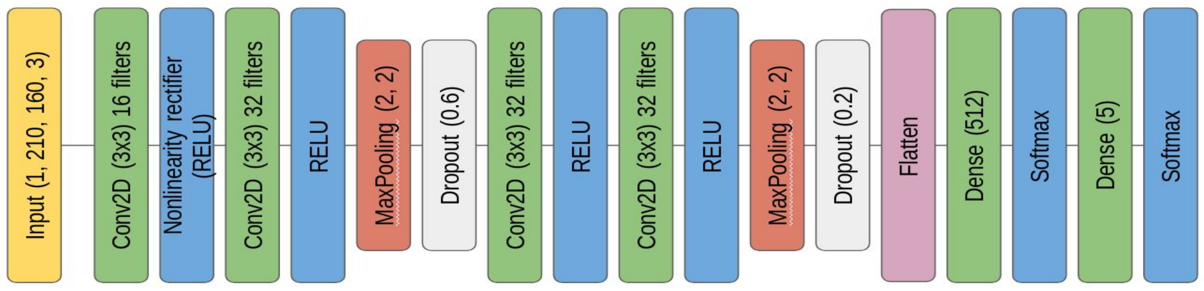
Because the network is initialized at random values and actually doesn't know to behave in the environment, there is a huge probability that its estimations will be inaccurate. Our algorithm is considered as 'greedy', it will choose to perform the action with the highest Q-value, but we can't guarantee the accuracy of that value because of its lack of exploration with the environment. In most of the cases, the agent will always choose the action from that point on and ignore the other possibilities. So we introduce an exploration factor, where our agent will choose to perform a random action instead of the one predicted by our neural network. This will allow our agent to eventually figure out hidden rewards in the environment. However, we want our agent to eventually converge to the optimal policy using the predicted values, so we are defining an exploration factor ε, where $1 \leq \varepsilon < 0.1$, that will decrease over time as the agent becomes more confident with its estimate of Q-values.

Dataset and Training

For this project, we are using a learning technique known as experience replay. First, we enter in a observation phase, in which the agent is going to 'remember' its interactions with the environment by storing experiences, an experience at the state s_t is defined as $e_t = (s_t, a_t, r_t, s_{t+1}, d)$ where *a* is the chosen action, *r* the immediate reward, and *d* is a boolean representing if the action was terminal (meaning that it's game over) or not, in a data set $D = [e_0, e_1, \dots, e_N]$. During the training phase, we will sample some of those experiences out of our dataset to perform Q-Learning updates (mini-batch updates) on our neural network. The batch size is fixed to 32 to avoid overfitting. Random sampling is used to reduce the correlation between consecutives experiences and thus reducing the variance and the loss.

Choosing the right neural network

A couple of neural networks were tested during this project. The first one is a variant from the Alexnet convolutional neural network w less layers and smaller filters.



The total number of trainable parameters on this model is 31778853. Like all the models used in this project, it take as an input an array of images, allowing to also interpret a “moving” environment, but because of technical constraints, we are only feeding him an array of one image.

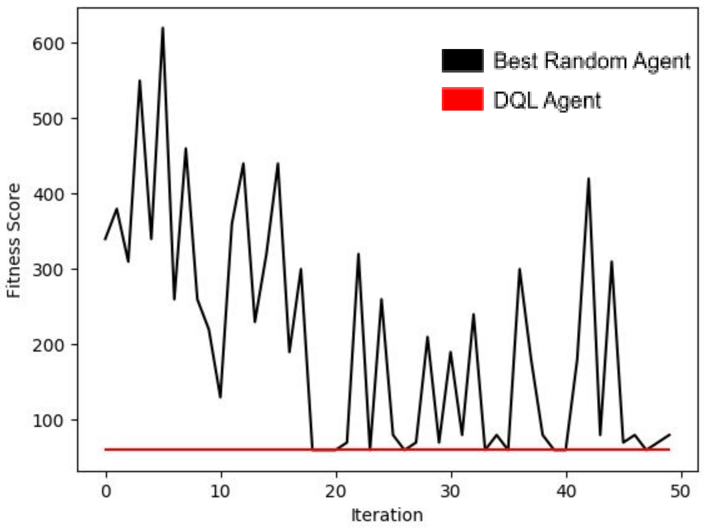
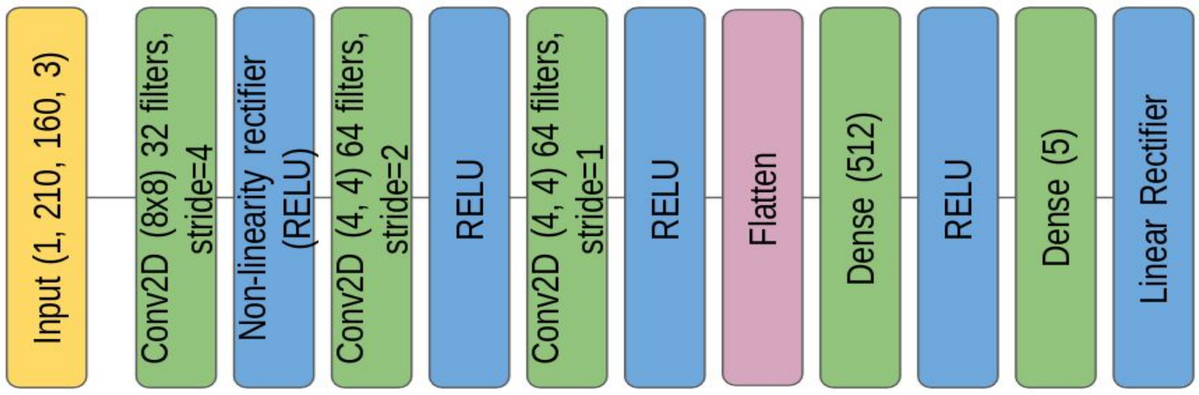


Figure 1: Observation of the performance of the different agents over time using the Alexnet-like model

Although that the Alexnet model was proved to work in many image-related problems, it doesn’t seem to be efficient in our case. The DQL agent would not improve its performance over time. In Figure 1, we can see clearly that the observation phase, using the random factor, can score relatively high inside the environment, but the DQL agent doesn’t seem to learn from it and stays at the same performance. The fitness score represent the total amount of reward earned at the end of the game.

To actually know if the issue came from our model architecture, we implement the Deepmind ‘Deep Q Network’, which is significantly smaller than the previous one and doesn’t rely on Dropouts and MaxPooling layers.


Input (1, 210, 160, 3)
Conv2D (8x8) 32 filters, stride=4
Non-linearity rectifier (RELU)
Conv2D (4, 4) 64 filters, stride=2
RELU
Conv2D (4, 4) 64 filters, stride=1
RELU
Flatten
Dense (512)
RELU
Dense (5)
Linear Rectifier

This network only consisted in 10429605 trainable parameters, which divided the training time by 2, from 2m07.2s for the Alexnet-like to 1m11.94s for this model on CPU power. Using the same parameters as the previous model (learning rate, exploration rate, decay), we could observe significant changes in the policy of the DQL agent.

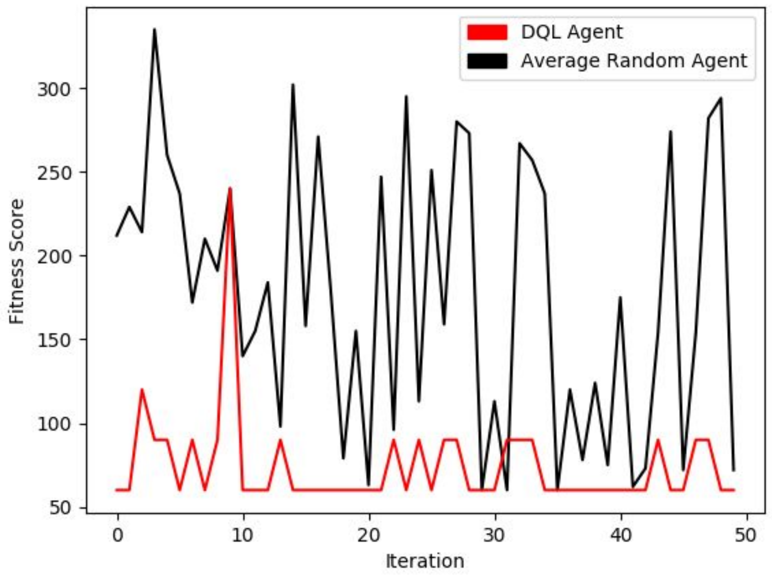


Figure 2: Observation of the performance of the different agents over time using the deepmind model

Since this model allowed us to actually train our model, the rest of the project is mainly using that one. The alexnet-like model will only be used to comparison purposes.

Experimentation

In the scope to try to improve our learning capability, some parameters and methods were altered to observes how the agent policy was impacted. First, we noticed strong correlation between the DQL agent performance and the global performance during the observation phase (Figure 3). The assumption of this effect was the fact that the replay memory was cleared at the end of each iteration, thus each learning phase will replay whatever performance was given at the current iteration, if that performance was poor, the DQL agent will be negatively impacted. To confirm this assumption and resolve the issue, the memory size was limited at the 100000 most recent frames, allowing us to replay the performances of several previous iteration. We can immediately see, in Figure 4, that the correlation between observation and learning performances was broken.

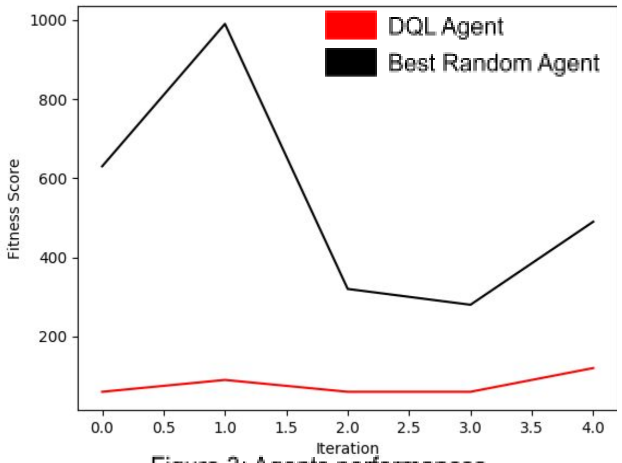


Figure 3: Agents performances with "short-term" memory

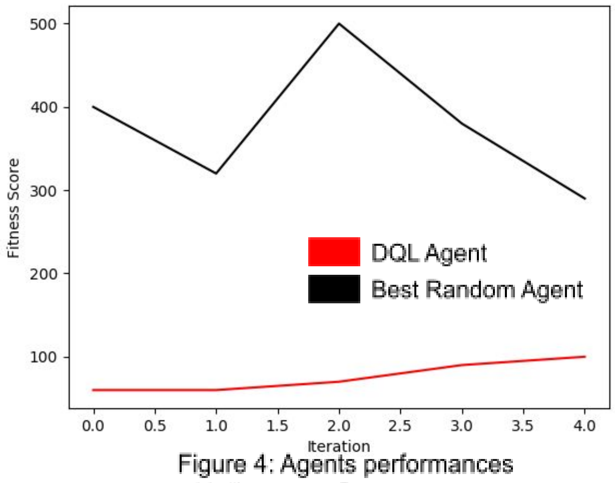
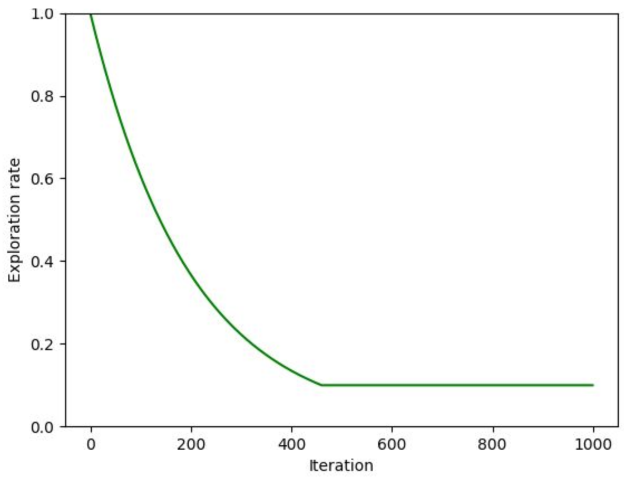


Figure 4: Agents performances with "long-term" memory

Even though we freed our agent to one constrain, the over all performance was not improved. On one observation, the DQL agent would score 650, where previously it will score a maximum of 520.

Results obtained

We fully monitored our program for a thousand iterations using the deepmind model, an exploration rate starting at 1 decaying to 0.1 by a factor of 0.995 per iteration, and a learning rate of 0.001. The results obtained were quite unstables but it demonstrate the learning capability of our algorithm:



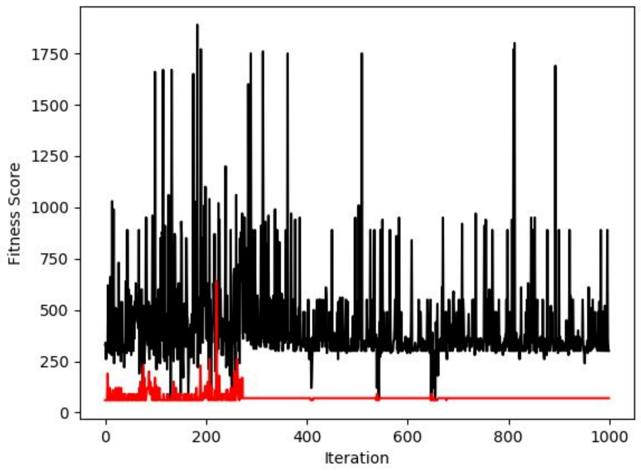
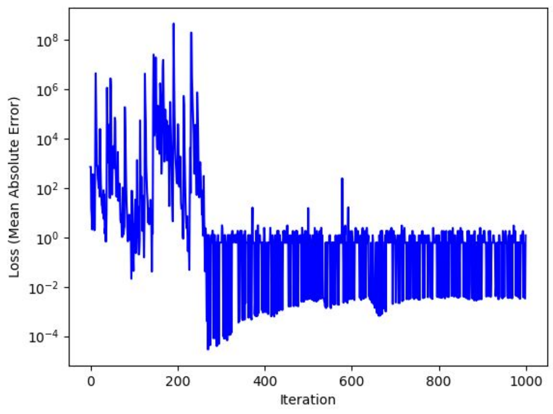


Figure 5: Learning rate

Figure 6: Fitness score

Figure 7: Loss



Although the DQL could learn how to play better, it would generally 'forget' what it learned, it will converge towards a new policy that will potentially fail in other situation than the one observed. This is the most challenging part of Q-Learning, it is very complex to build a complete understanding of the model from scratch. Smallest changes in the policy could turn the tables.

Unknown issues

After a few hundred iterations, the agent would stop progressing, or even regress. The cause of this particular phenomenon is not yet determined. Our first assumption is our decaying learning rate used by the RMSprop optimizer in use in the deepmind model. This optimizer uses an adaptive learning rate method that divides the learning rate by an exponentially decaying average of squared gradients. Because the learning rate is relatively small at the beginning (usually between 0.001 and 0.00025) and the decaying rate around 10^{-6} it is almost impossible to estimate the point were the algorithm is not assimilating the updates. We can see in the last graphs that as soon as the agent 'flatline', the loss stays between 1 and 0,0001. Which is pretty similar to the alexnet-like network, in which there was no progress either (Figure 8).

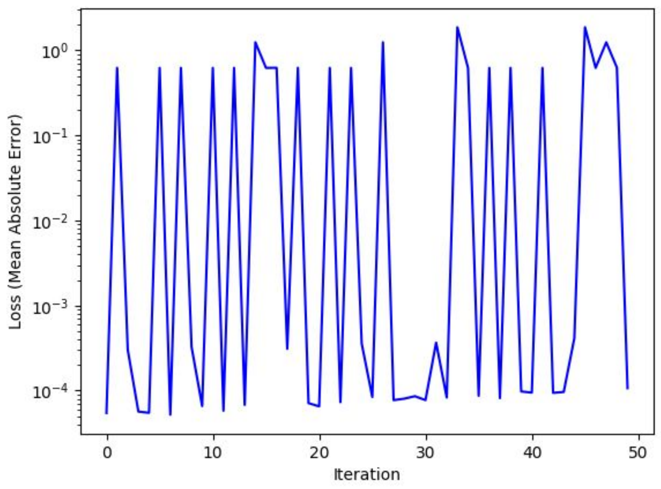


Figure 8: Loss on Alexnet-like model

References

GitHub project: https://github.com/Kyalma/dql-pacman

DQN Nature Paper: https://bit.ly/2ruP9CT