Deep Reinforcement Learning: How to teach a machine to play PacMan

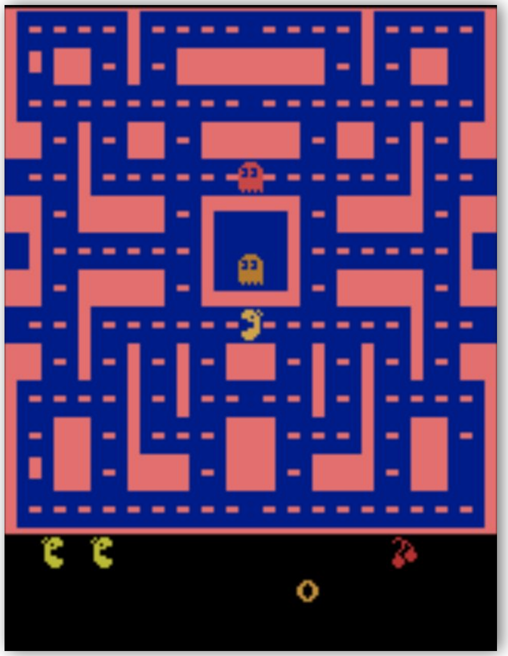
CECS 406: Machine Learning
California State University Long Beach

May 10, 2018

Léo Poulin

Source code:
github.com/Kyalma/dql-pacman

Introduction


MsPacman-v0 Environment

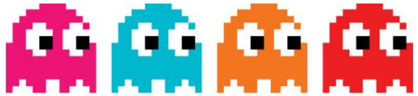
What drove me to start this project:

- Creating its own Artificial Intelligence is tedious and not very accurate
- Reinforcement Learning appears more likely to be used in the industry and research fields

Environment:

- Needed to be as close as possible to the original PacMan game's AI behaviour
- OpenAI Gym
  - ALE + Stella (Atari 2600 Emulator)

ALE: MG Bellemare, Y Naddaf, J Veness, and M Bowling. "The arcade learning environment: An evaluation platform for general agents." Journal of Artificial Intelligence Research (2012).
Stella: A Multi-Platform Atari 2600 VCS emulator http://stella.sourceforge.net/



Q-Learning

- Model-free Reinforcement Learning
- Predict the utility of the available actions for a given state *s*
- The AI can use value-iteration or policy-iteration algorithms to find an optimal policy directly from its interactions with the environment without building a model
    - Temporal Difference Learning: Update the value of the previous action based on the value of the current action
- Can handle problems with stochastics transitions and rewards

The Q-Learning approach is often referred as “Deep Reinforcement Learning” or “Deep Q-Learning”, in our study case.

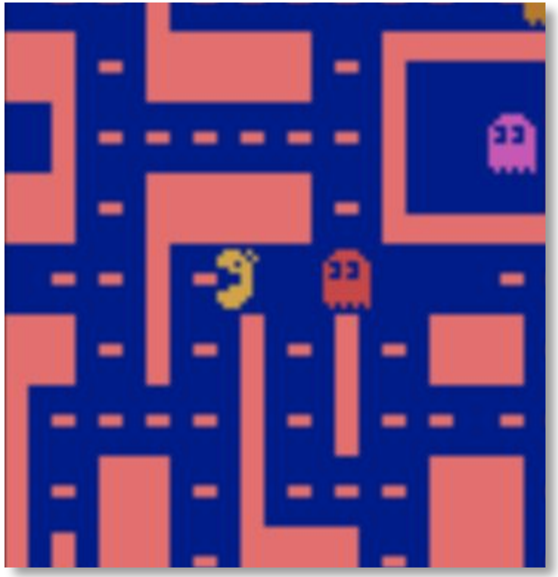
Quality equation:

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

Bellman equation:

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

Q function


MsPacman environment in a State λ

Define a function Q that represent the maximum discounted future reward when performing an action *a* in a state *s*

Q(state=λ, action=NOOP) = 7409396.0

Q(state=λ, action=UP) = 8664897.0

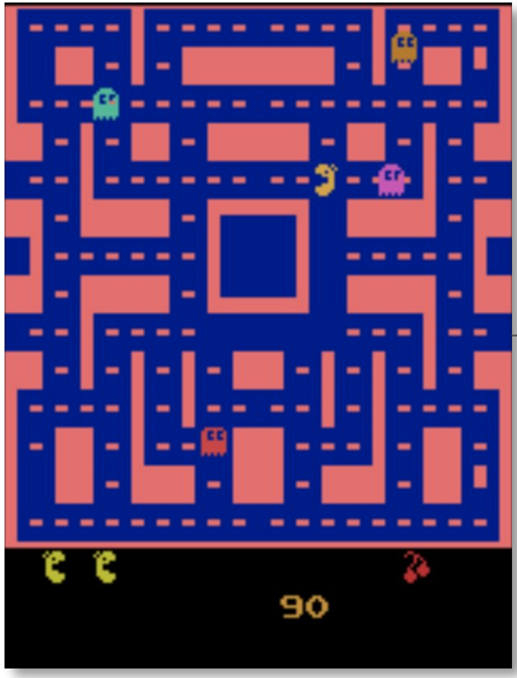
Q(state=λ, action=RIGHT) = 9259114.0

Q(state=λ, action=LEFT) = 9347670.0

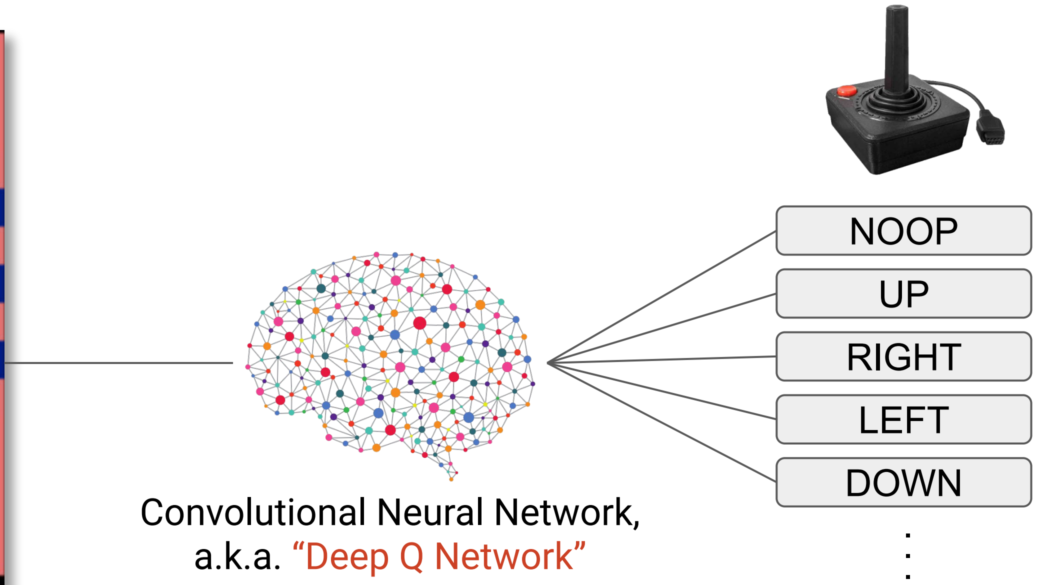
Q(state=λ, action=DOWN) = 6671756.5

The AI will pick the action with the highest Q-value and continue optimally from that point on. Thus we can estimate the score at the end of the game without knowing futures actions and rewards

Input and Output spaces



Input: Full RGB Image 210x160x3



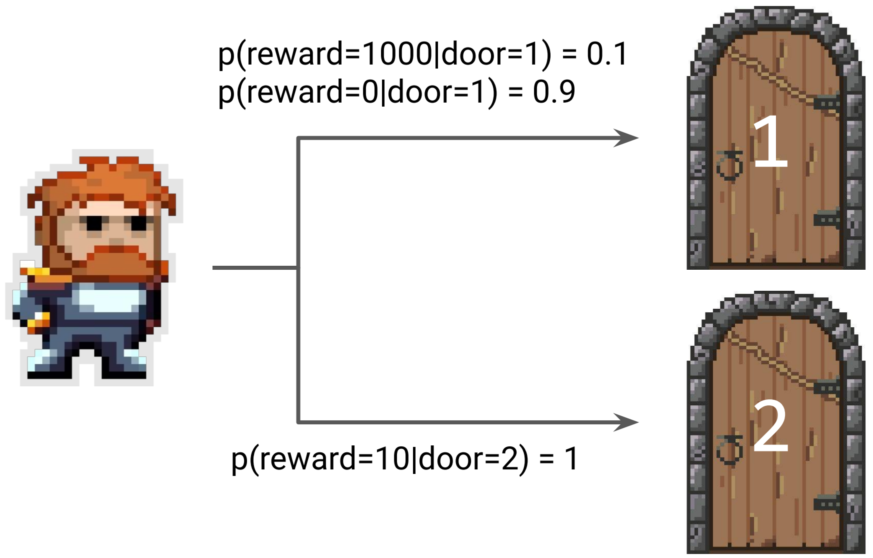
Convolutional Neural Network,
a.k.a. "Deep Q Network"

Output space: (1, 5)
Originally (1, 18) but
reduced to fit the game

Picture from: https://medium.com/@startuphackers/building-a-deep-learning-neural-network-startup-7032932e09c

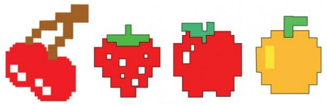
Exploration VS Exploitation

Since the Q-table is initialized randomly, we can say that our model is not explored.
Consider the following situation:



Our agent can get "unlucky" and earn the lowest reward by using door number 1. And because our algorithm is "greedy", it will choose door number 2 from that point on.

So we introduce the concept of "Exploration": The agent will be given a probability to perform a random action instead of the optimal one. Thus allowing it to eventually figure out some higher hidden reward.



Pictures from:

- http://pixeldungeon.wikia.com/wiki/Category:Characters
- https://www.pinterest.fr/pin/228768856055666541/

Exploration VS Exploitation

However, we want our agent to eventually converge to an optimal policy. So we are going to lower that probability of performing a random action over time as it confident more confident with its understanding of the model (Q-values estimations).

Pictures from:

- https://pixelartus.com/tagged/evolution-of-lara

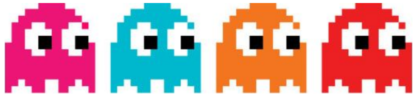
Training the network: Observation

Once we have defined our Random Agent and our DQL Agent, we can start the observation phase.

We store the agent's experience at each step $e_t=(s_t, a_t, r_t, s_{t+1})$ in our memory dataset $\mathcal{D} = e_0, e_1, \dots, e_n$.

Pseudo-code for the observation function:

```
state = environment.reset()
done = False
while not done:
     if random.rand() > exploration_threshold:
          action = random.choice(environment.action_space)
     else:
          action = argmax(model.predict(state))
     next_state, reward, done = environment.step(action)
     memory.append([state, action, reward, next_state, done])
     state = next_state
```



Training the network: Learn by replay

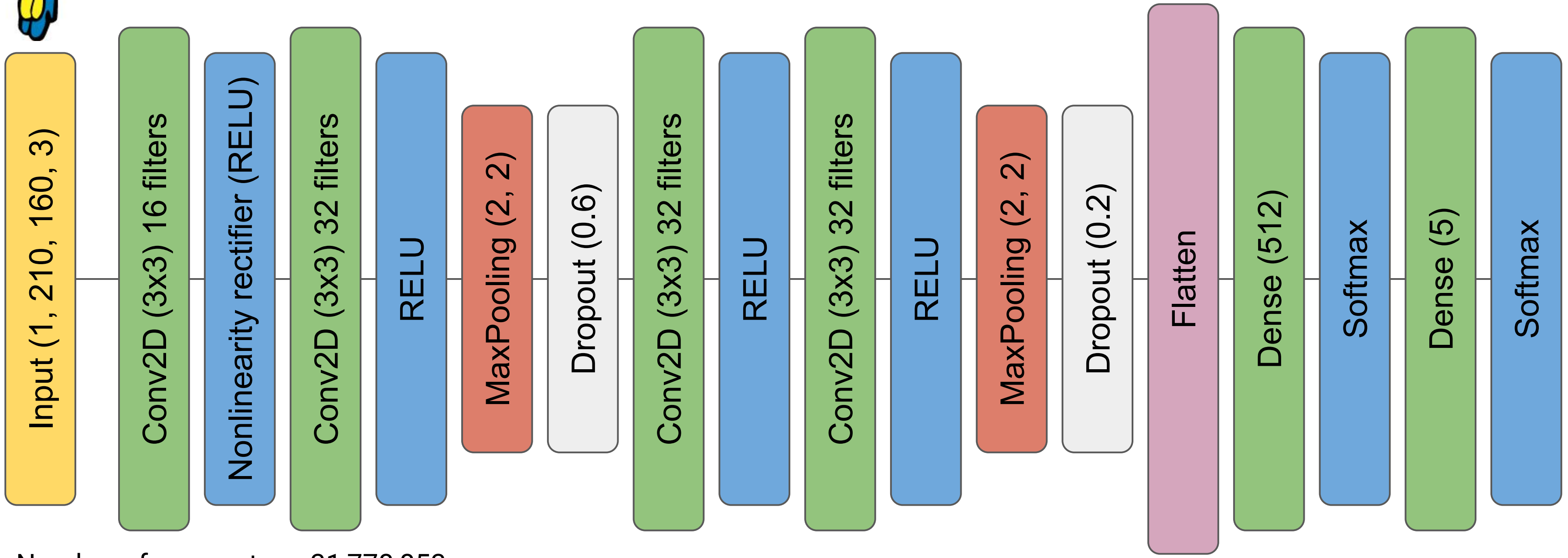
Then we perform Q-Learning updates, or mini-batch updates, using samples of experiences drawn at random in our memory data-set.
The reason why we choose random samples is because the correlation between consecutives experiences. Randomizing breaks this correlation and thus reduce the variance of the updates.

Pseudo-code for the learn by replay function:

```
batch = random.sample(memory, batch_size)
for state, action, reward, next_state, done in batch:
      if done:
            target = reward
      else:
            target = reward + discount_rate * max(model.predict(next_state))
      target_function = model.predict(state)
      target_function[action] = target
      model.fit(state, target_function, epochs=1)
if exploration_rate > min_exploration_rate:
      exploration_rate *= exploration_decay
```

The batch size is always equal to 32 to avoid overfitting.

Building the Neural Network: Alexnet-like

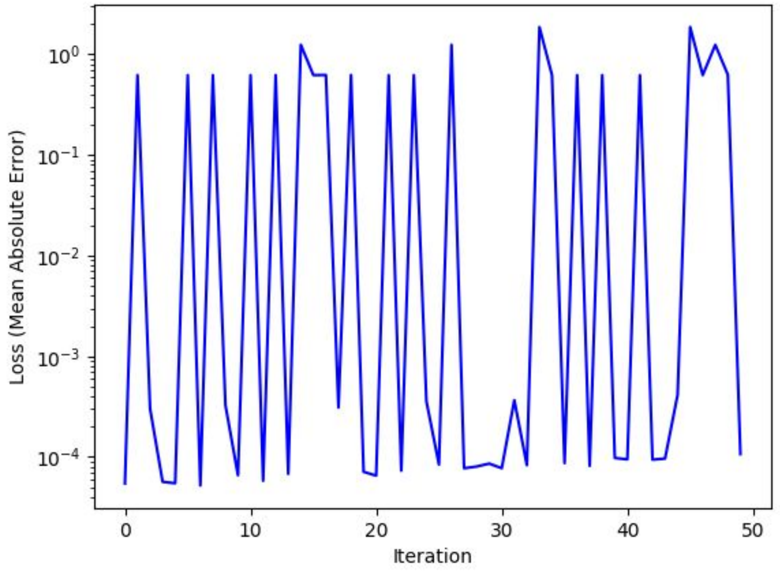


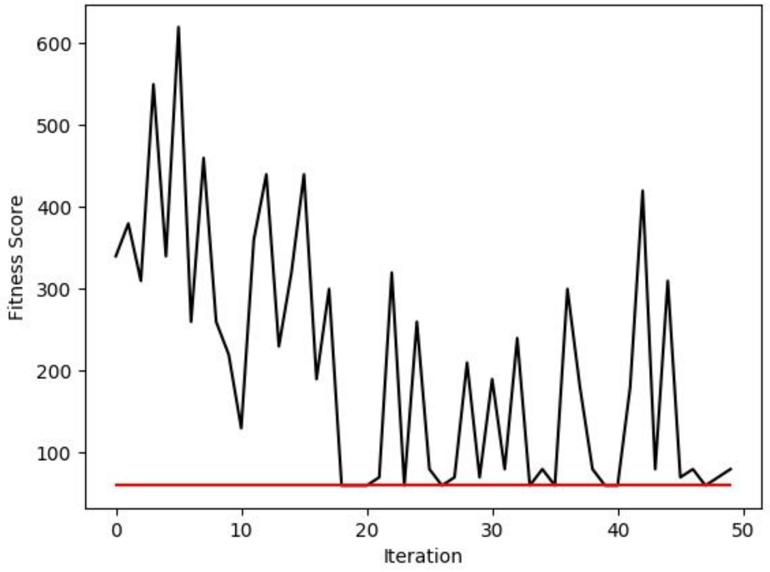
Number of parameters: 31,778,853
Iteration time: 2:07.2
Optimizer: Adam

- Adaptive learning rates
- Store an exponentially decaying average of past squared gradients
- keeps an exponentially decaying average of past gradients (similar to momentum)

■ Best score during observation time (RandomAgent)

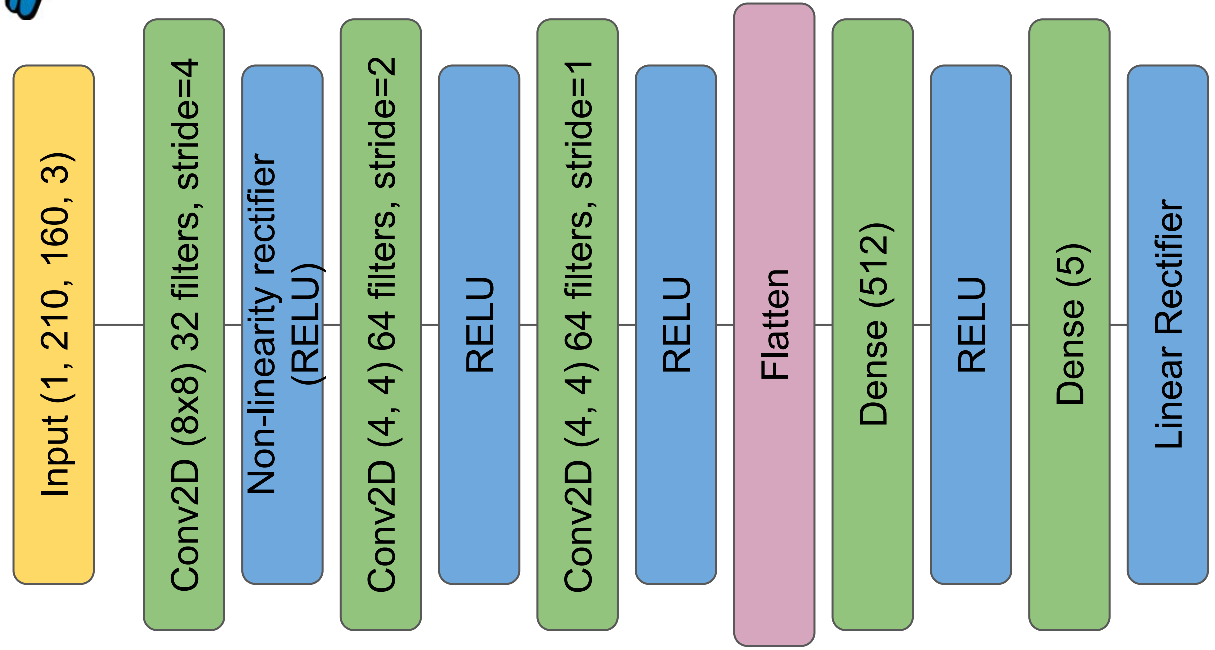
🟥 Score of the trained model (DQLAgent)





Observation: Loss stays between 10 and 0.0001, which is relatively low, but the trained model never "learn", the score never increase over time.

Building the Neural Network: Deepmind

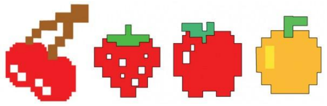

Input (1, 210, 160, 3)
Conv2D (8x8) 32 filters, stride=4
Non-linearity rectifier (RELU)
Conv2D (4, 4) 64 filters, stride=2
RELU
Conv2D (4, 4) 64 filters, stride=1
RELU
Flatten
Dense (512)
RELU
Dense (5)
Linear Rectifier

Number of parameters: 10,429,605

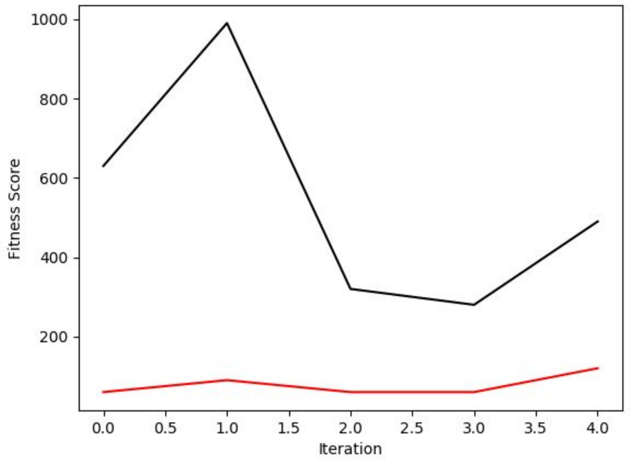
Iteration time: 1:11.94

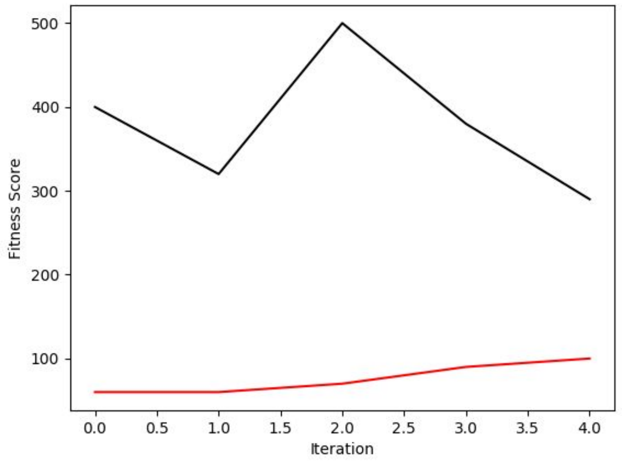
Optimizer: RMSprop

- Adaptive learning rate method
- Divides the learning rate by an exponentially decaying average of squared gradients.



Choosing the right memory size





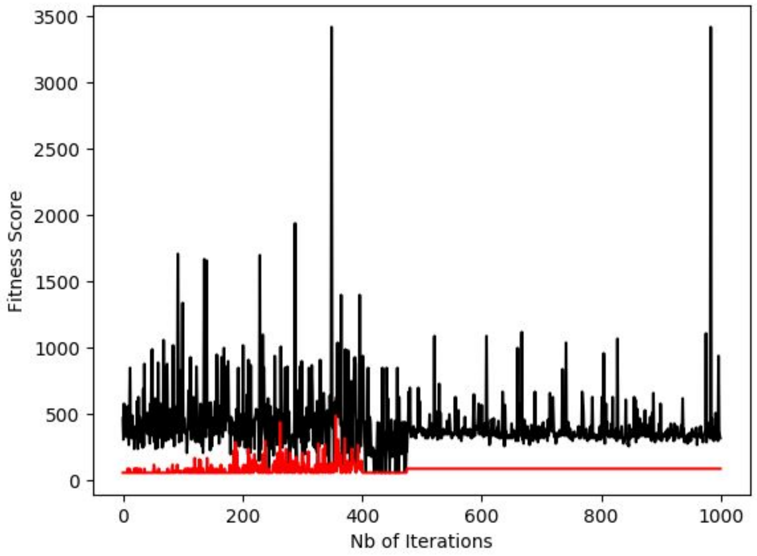
“Forgetting” the previous interactions as soon as the learn on replay was done (roughly 7500 frames)

VS

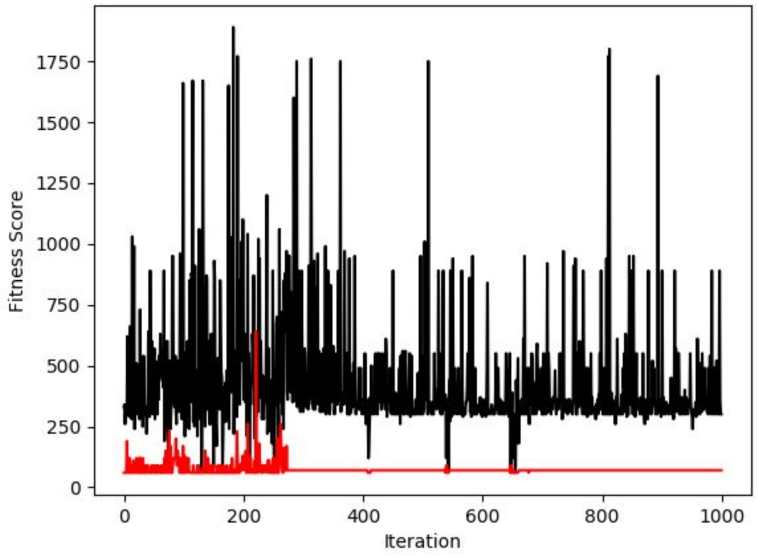
Setting the memory to fixed size of 100000 frames

By choosing a memory size that include previous iterations, we are no longer influenced by the performance of the current observation time.

Impact of the memory size on the performance



With “short-term” memory

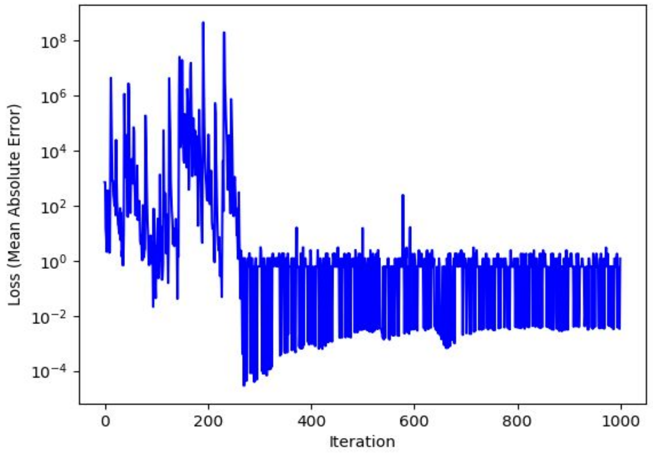


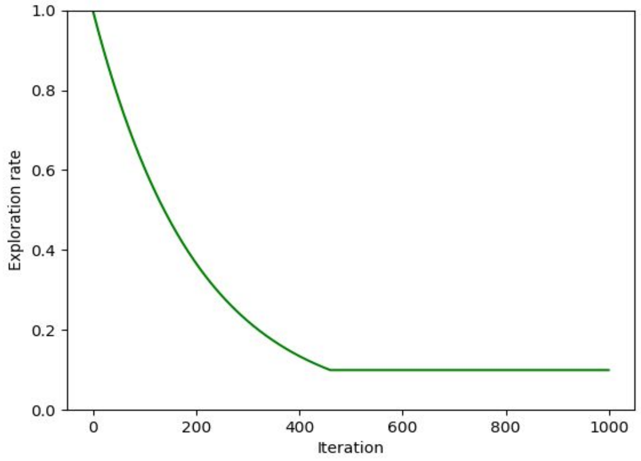
With “long-term” memory

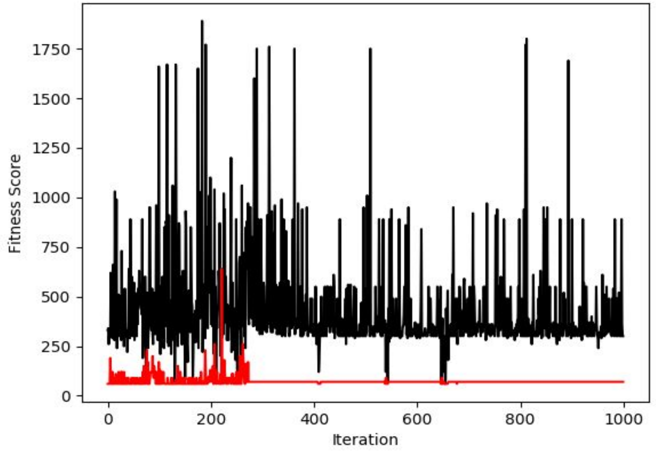
Assumption: Training on more data would increase the chance of training on “qualitative” data and significantly increase the AI performance.

Result: Average fitness score is practically not impacted.
Best performance only increased by ~20%. But computation time was doubled.

Results







Model: Deepmind
Exploration rate initial value: 1
Exploration rate min value: 0.1
Exploration decay per each step: 0.995
Learning rate: 0.001

Issue: We can observe that after ~250 iterations, the score of the DQL agent stays relatively the same and the loss drop, like we observed with the alexnet-like model.

Related works, Ressources, and Conclusion

DQN Nature Paper "Human-level control though deep reinforcement learning":
https://deepmind.com/research/publications/human-level-control-through-deep-reinforcement-learning/
https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

- Fine tuning of the algorithm, frame-skipping, AI performance compared to a human

Youtube channels that helped me to understand the basic of Deep Reinforcement Learning:

Jabrils

Siraj Raval

Thank you everyone who believed in me for this project

Header and Footer pictures from:

- https://www.teepublic.com/t-shirt/458352-8-bit-pacman-ghosts
- https://www.pinterest.com/pin/560627853606320058/
- http://pacman.wikia.com/wiki/Ms._Pac-Man