Reinforcement Learning: DQN vs PG

ROSSETTO FEDERICO

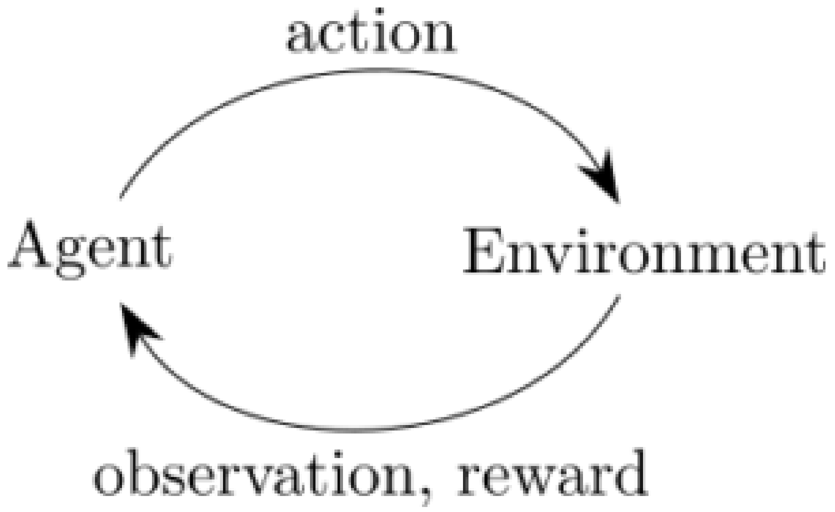
INTELLIGENT SYSTEMS FOR PATTERN RECOGNITION

2017/2018

Contents

Introduction – The problem

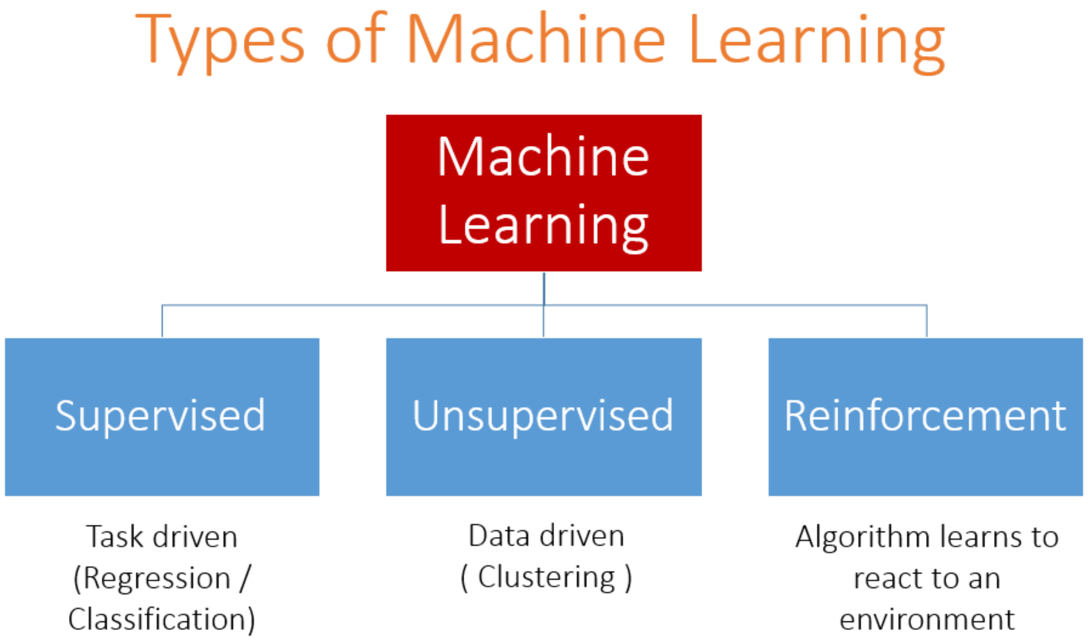
The goal is to develop an agent that learns to play a game by playing it



The goal is to **learn the game**
<u>without having any prior knowledge</u>

Introduction – Where's the difference

What is the key difference from what we have seen so far?



We are trying to learn how an agent should act, based on a **Reward Function** that gives us the outcome of our actions

Introduction – Markov Decision Process

A **Markov Decision Process** is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ where:

$\mathcal{S}$ is a finite set of states

$\mathcal{A}$ is a finite set of actions

$\mathcal{P}$ is a state transition probability matrix

$\mathcal{R}$ is a **Reward Function**, where $R_s^a = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$

γ is a discount factor between 0 and 1

Introduction - Notations

Return: The return is the sum of all future rewards discounted exponentially by the time

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Policy: A probability distribution of the actions over the states

$$\pi(a|s) = P(A_t = a | S_t = s)$$

State-Value Function: Expected return from a state given a policy

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

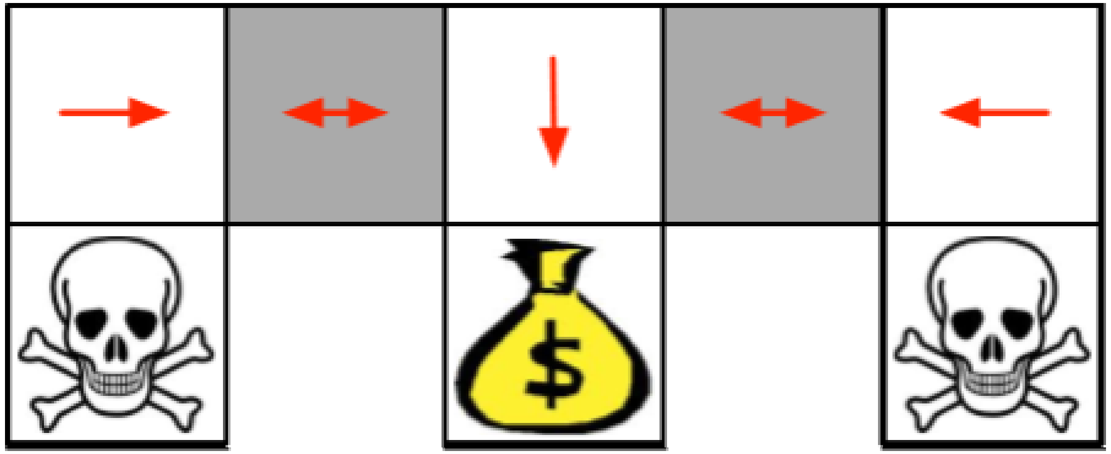
Action-Value Function: Expected return from a state, making an action and then following a policy

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

Introduction – Stochastic vs Deterministic

Deterministic Policy: given a state, the agent performs always the same action

Stochastic Policy: given a state, the agent outputs a probability distribution over the actions



The Agent **cannot differentiate** between the gray areas

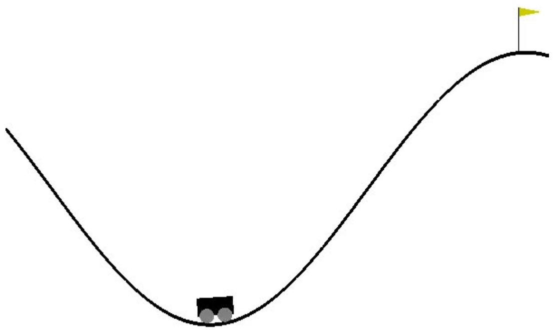
- A **Deterministic Policy** <u>cannot</u> solve this from every possible starting point
- A **Stochastic Policy** <u>can</u> solve this from every possible starting point

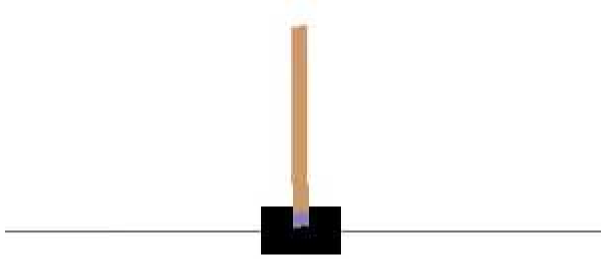
Introduction – Environments

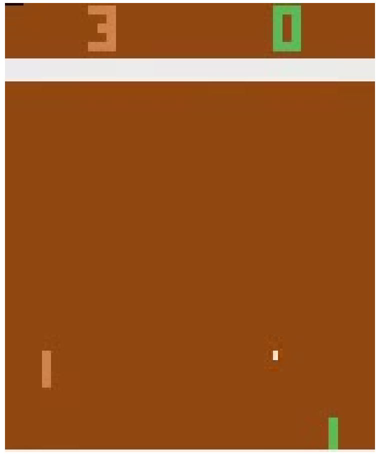
For the experiments it has been used the OpenAI Gym Library

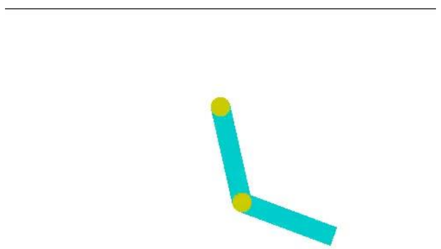
The games that I have tested are:

- CartPole
- AcroBot
- MountainCar
- Pong









DQN Agent - Overview

DQN (Deep Q Network): Deep Neural Network used to estimate the Action-Value Function

A DQN Agent Works with Deterministic Policies

Off-Policy: Learns about the optimal policy while following an Exploration Policy

Q-Learning Equation (Bellman Optimality Equation):

$$Q^*(s_t, a_t) = r_t + \gamma * \max_a Q^*(s_{t+1}, a)$$

We are not actually learning a policy, but we can derive the **optimal policy** by knowing the Action-Value Function:

$$\pi(s) = \arg\max_a Q(s, a)$$

DQN Agent – Training (Online)

$$L(\theta) = \frac{1}{L}\sum_{i=1}^{L}\left(r + \gamma * \max_{a'} Q'(s', a') - Q(s, a)\right)^2$$

Where s' is the successor state of s after performing the action a

The implementation of this Loss as been done using **Keras**, using the Mean Square Error, and computing the target $y = \left[r + \gamma * \max_{a'} Q'(s', a')\right]$

DQN Agent - Improvements

- **Remember – Replay Strategy:** Memorize experience and sample from it at each step
- **Frozen Model:** The target model Q' is not updated online, but synchronized after C steps
- **Huber Loss:** Loss function that is less sensitive to outliers

$$L(x) = \begin{cases} \frac{1}{2}x^2 & |x| < 1 \\ |x| - \frac{1}{2} & |x| \geq 1 \end{cases}$$

- **Double DQN:** Stabler variation to the computation of the target

$$y = r_t + \gamma \cdot Q'\left(s_{t+1}, \arg\max_a Q(s_{t+1}, a)\right)$$

- **Weights Reinitialization:** If the network is stuck, reinitialize the weights

DQN Agent – Implementation Details

- **Store:** Takes a state, an action, a reward and the next state and stores them in memory
- **Act:** Takes a state and predicts the next action to perform
- **Train:** Samples from memory some experience and trains the model on them

```python
if np.random.uniform(0,1) < self.exploration_rate:
    action = np.random.randint(self.action_size)
else:
    state = np.reshape(state, [1, self.state_size])
    q_values = self.model.predict(state)[0]
    action = np.argmax(q_values)

return action
```

Act Function with Exploration Rate

```python
for x, action, reward, next_x in np.array(self.memory)[minibatch]:

    data_train = np.reshape(x, [1, state_size])
    target = self.model.predict(data_train)[0]

    data_next = np.reshape(next_x, [1, state_size])
    if double:
        target_action = np.argmax(self.model.predict(data_next))
        max_next_target = self.frozen_model.predict(data_next)[0][target_action]
    else:
        max_next_target = np.max(self.frozen_model.predict(data_next)[0])

    target[action] = reward + self.gamma*max_next_target

    x_train.append(data_train)
    y_train.append(target)
```

Train Function – Sampling and Target Computing

PG Agent - Overview

Goal: Given a stochastic policy $\pi_\theta(s, a)$ find the best policy

Since we are in an Episodic Environment, we can use the start value function as a metric to evaluate our policy:

$$J(\theta) = v_\pi(s_1) = \mathbb{E}_\pi(G_1)$$

On-policy: Learns to optimize his own policy

Policy Gradient Theorem

For any differentiable policy $\pi_\theta(s, a)$, the policy gradient is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) \cdot Q^{\pi_\theta}(s, a)]$$

PG Agent – Training (Batch)

$$L(\theta) = -\sum_{s,a,r} r * \log \pi(s,a)$$

The idea is to <u>maximize the probabilities</u> of actions that yields **positive rewards**, and <u>minimize</u> the ones that gives **negative rewards**

Each Reward has been computed following the Return Equation:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Sadly it is much harder to implement this kind of Loss using Keras, so this algorithm has been implemented using **TensorFlow**

PG Agent – Graph Structure

```python
with tf.name_scope("inputs"):
    self.input_net = tf.placeholder(tf.float32, [None, self.state_size], name="input_state")
    self.actions    = tf.placeholder(tf.int32, [None, 1], name="actions_performed")
    self.d_rewards = tf.placeholder(tf.float32, [None, ], name="discounted_epiosde_rewards")

with tf.name_scope("dense_layers_architecture"):
    self.dense1 = self.input_net

    for n_layer in self.layers_architecture:
        self.dense1 = tf.contrib.layers.fully_connected(inputs = self.dense1,
                                                         num_outputs = n_layer,
                                                         activation_fn = tf.nn.relu,
                                                         weights_initializer = tf.contrib.layers.xavier_initializer(uniform=False))

with tf.name_scope("softmax_output"):
    self.output = tf.contrib.layers.fully_connected(inputs = self.dense1,
                                                     num_outputs = self.action_size,
                                                     activation_fn = None,
                                                     weights_initializer = tf.contrib.layers.xavier_initializer(uniform=False))

    self.action_distribution = tf.nn.softmax(self.output)

with tf.name_scope("loss"):
    log_prob = tf.log(self.action_distribution)
    indices = tf.range(0, tf.shape(log_prob)[0]) * tf.shape(log_prob)[1] + self.actions
    act_prob = tf.gather(tf.reshape(log_prob, [-1]), indices)
    self.loss = -tf.reduce_mean(tf.multiply(act_prob, self.d_rewards))

with tf.name_scope("training_operation"):
    self.train_operation = tf.train.AdamOptimizer(self.learning_rate).minimize(self.loss)
```

- Inputs
- Dense Layers
- Soft-max Output
- Log Likelihood Loss
- Adam Training

PG Agent – Implementation Details

- **Store:** Takes a state, an action and the reward and stores them in memory
- **Act:** Takes a state and samples the next action to perform
- **Train:** Samples from memory some experience and trains the model on them

```
prob_distribution = tf_session.run(self.action_distribution,
                    feed_dict={self.input_net: state.reshape([1,self.state_size])})

action = np.random.choice(self.action_size, 1, p=prob_distribution.ravel())[0]

return action
```

Action Sampling

```
states_train = np.reshape(states_train, [current_batch_size, self.state_size])
action_train = np.reshape(action_train, [current_batch_size, 1])

tf_session.run([self.loss, self.train_operation], feed_dict={self.input_net: states_train,
                                                self.actions:   action_train,
                                                self.d_rewards: reward_train
                                                 })
```

Training Operation

```
# Reward Min-Max Normalization
reward_list = np.array(self.memory)[:,2]
reward_max = np.max(reward_list)
reward_min = np.min(reward_list)
if reward_max != reward_min:
    reward_list -= reward_min
    reward_list /= (reward_max - reward_min)/2
    reward_list -= 1
```

Reward Normalization

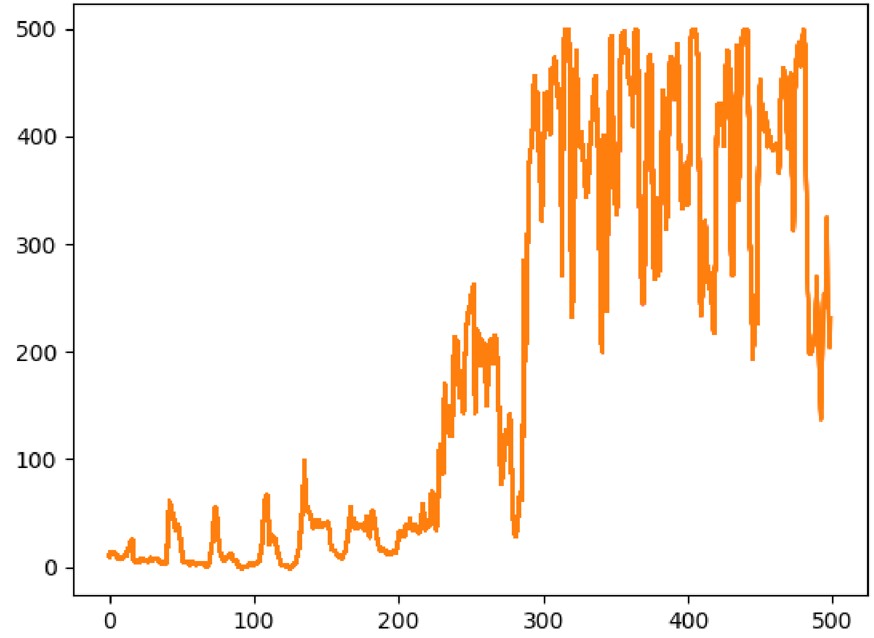
Results – Set up

The Experiments has been structured as:

- **Training Phase:** The algorithms run until the Mean Score of the last 10 episodes reaches a threshold
- **Testing Phase:** The agent plays 100 games without changing the policy obtained

I have also added a reward when the algorithm does not reaches the maximum amount of **timesteps**

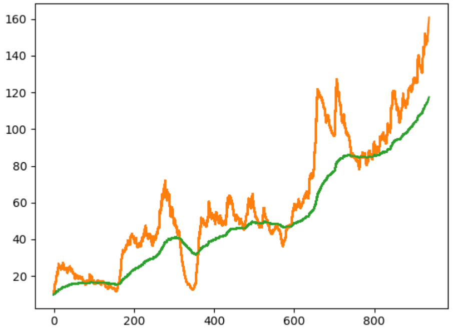
	CartPole	AcroBot	MountainCar
Target Score	+160	-200	-150
Done Reward	-10	+10	+10



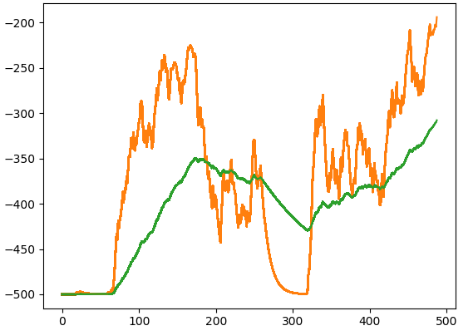
One of the first attempts on CartPole using the DQN Agent

Results – CartPole and AcroBot

DDQN AGENT

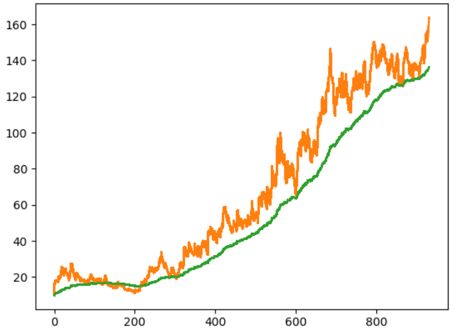


CartPole

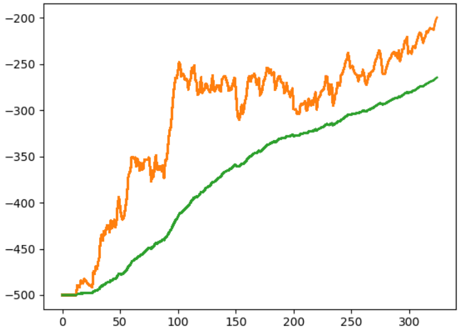


Acrobot

PG AGENT



CartPole

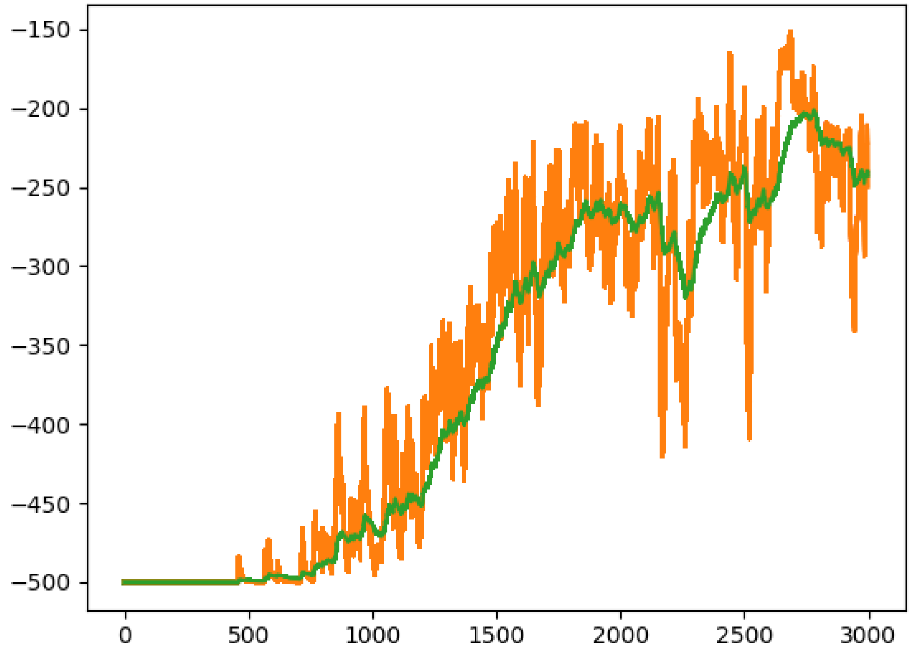


Acrobot

- **Architecture** used in both cases: 2 Layers of 64 H.U. with ReLU activation function
- Both agents are very sensible to the random initialization of the weights

Results – Mountain Car (DDQN Agent)

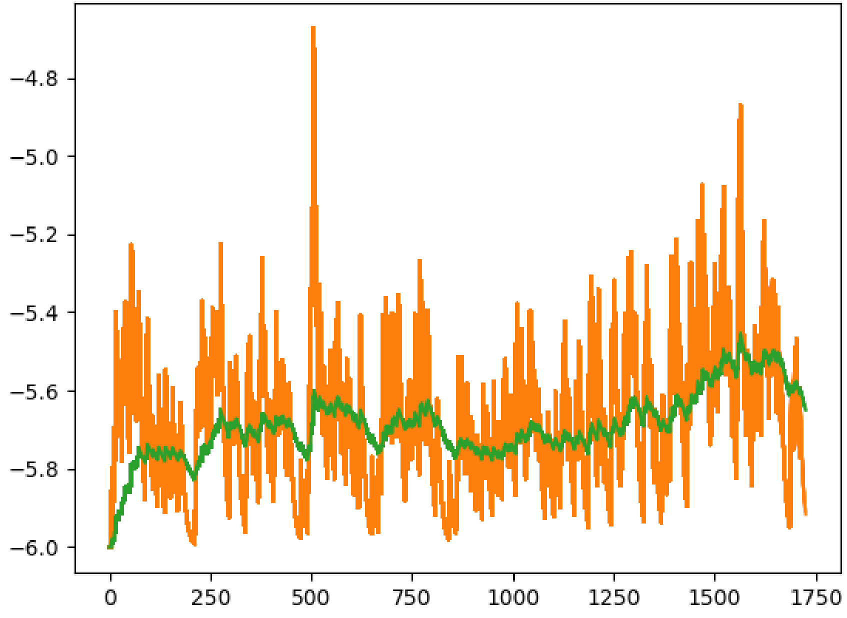
- The game is **quite hard** for the agents that I have developed.
- The main challenge is the small amount of timesteps available to complete the task (200)
- To try and solve them, I have increased the amount of timesteps available (500), and the **DDQN agent** was able to solve this easier version



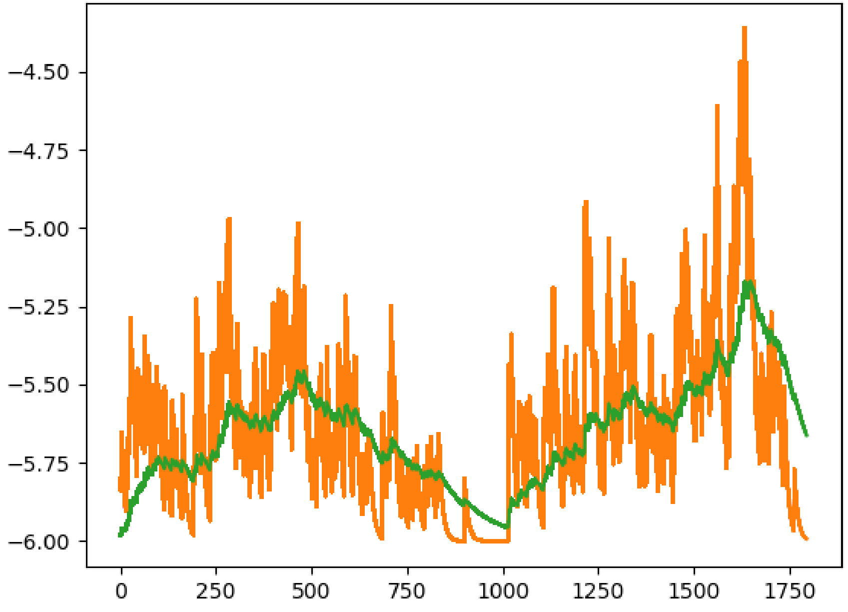
- The **PG agent** wasn’t able to solve this game

Results – Pong (PG Agent)

- I decided to use the Deterministic Pong version, since it is more stable and should be easier to learn
- Since in this case we are using images as input of our **CNN**, the networks are much bigger than before, and requires much more time to train



Update every 5 episodes



Update every episode

Summary – Comparison

- DQN Agents are more unstable and have higher variance
- PG Agents are using stochastic policies, so they can learn more dynamic behaviors

BUT

- In the games that I have tested, when a DQN Agent converges, it usually has higher performances on the test games
- This is probably due to the deterministic type of games that we are dealing with

	DQN Agent	DDQN Agent	PG Agent
AcroBot	-141	-152.43	-206
CartPole	182.58	192.25	131.94

Summary – Conclusions

- Reinforcement Learning algorithms are quite unstable, and can often diverge
- There are a lot of strategies to **improve** the performances and make them more stable
- This means a lot of parameters to tune
- Overall, a really **powerful tool**

THE END

References

- Human-level control through deep reinforcement learning. https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf
- Deep Q-Learning with Keras and Gym. https://keon.io/deep-q-learning/
- Deep Reinforcement Learning: Pong from Pixels. http://karpathy.github.io/2016/05/31/rl/
- An introduction to Policy Gradients with Cartpole and Doom. https://medium.freecodecamp.org/an-introduction-to-policy-gradients-with-cartpole-and-doom-495b5ef2207f
- UCL Course of RL by David Silver. http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html