Reinforcement learning Analysis with Open AI Gym

Diego VIllarreal De La Cerda

Universidad de las Americas Puebla UDLAP

Artificial Intelligence LIS3082

Abstract—*The following Lab report makes a comparison hon how different approaches to Reinforcement learning makes a difference into the result an intelligent agent will have, using the library "gymnasium", from open AI, the approaches we are going to follow are* ***hard-code****,* ***Neural Network****,* ***policy gradients*** *and* ***Q-learning***

Keywords—*LaTeX class, lab report, academic paper, Artificial inteligence, intelligent agents*

1. Introduction

In the field of Artificial Intelligence one of the most basic concepts is the Reinforcement learning whose objective is to teach an intelligent agent by a "Reward" system assigned by a supervisor, the challenge is to determine what is the best approach where the intelligent agent learns better and faster, also this analysis is based on the code done by Aurélien Geron [3]

1.1. Cart pole Environment

The *"Cart Pole Environment"* is a simple game where a pole is placed in the top a cart that is capable to move sideways, the main goal here is to balance the pole over the cart as long as possible, the simulation is over when the either the agent reach the goal (that means it reach a certain amount of steps, for this experiment is set to 200 steps) or fail (that means that the pole falls down before)

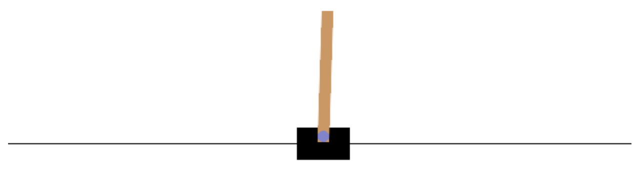

Figure 1. *Cart pole Environment*

1.2. Setting the Environment

For this simulation we are going to use Python as the main programming language and TensorFlow as our AI trainer, with the Gymnasium library as you can see in 1

```python
import gymnasium as gym
import sys
assert sys.version_info >= (3, 7)
from packaging import version
import tensorflow as tf
import matplotlib.animation
import matplotlib.pyplot as plt

assert version.parse(tf.__version__) >= version.
    parse("2.8.0")
env = gym.make("CartPole-v1", render_mode="
    rgb_array")

plt.rc('font', size=14)
plt.rc('axes', labelsize=14, titlesize=14)
plt.rc('legend', fontsize=14)
plt.rc('xtick', labelsize=10)
plt.rc('ytick', labelsize=10)
plt.rc('animation', html='jshtml')
```

Code 1. *dependencies load*

once initialized the environment each observation of the agent returns a 1D Numpy array that represents the state of the x coordinate of the cart, its velocity, the angle of the pole and the angular velocity of the pole Also the agent has a "Action Space" in this case the agent is the cart with 2 possible Actions move left (0) and move right(1)

2. Hard Coded policy

The hard Coded policy is a simple manner of making the pole stable, if the angle of the pole angle is positive then push the cart to the right and if its negative push to the left.

```python
def basic_code(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1
```

Code 2. *Hard Coded policy*

by simulating 500 episodes the results are as expected not as good a we want to, analyzing the return of the mean, the standard deviation, minimum and maximum is the following

$$\begin{bmatrix}41.698 & 8.389445 & 24 & 63\end{bmatrix}$$

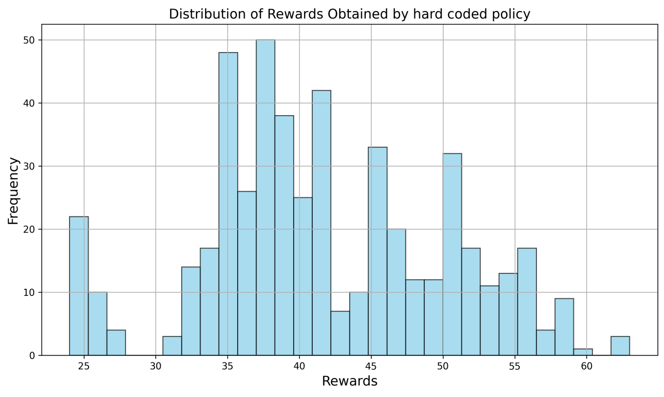

Figure 2. *Hard-coded policy distribution*

the best our agent could do is to get 63 steps so far away from the 200 steps, nonetheless the execution time is so much faster than the next methods.

3. Neural Networks

3.1. Defining the Neural Network

The neural network will take observations as inputs and estimate a probability to take either action 0 or action 1, since the neural network is a black box i cannot explain detail what is doing

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(5, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid
    "),
])

def pg_policy(obs):
    left_proba = model.predict(obs[np.newaxis],
    verbose=0)[0][0]
    return int(np.random.rand() > left_proba)
```

Code 3. *Neural network definition*

the idea is Simple, by a given input of the observation the network will estimate a Bernoulli distribution for the environment, representing with the probability **P** to take action 0 and probability **1-p** the action 1

$$P[X = x] = p^x(1-p)^{1-x} x = 0, 1$$

For this particular case each observation is completely independent from the other since a result of the fundamental theorem of probability each random variable so this can be interpreted as that the current observation is complete environment full state

3.2. Exploration-exploitation

as you may see in the code 3 we chose between a random probability and the network prediction instead of choosing the one with the highest probability given by the model, this opens up to talk about the concept of ***"Exploration-exploitation dilemma"*** the sthocastical method is limits the results a network could get, the exploratory factor is given by the randomness of the choice between moving Left or right

3.3. Network Results

By simulating the same model 500 hundred times just as the Hard Coded Policy to compare it in equal conditions we get the following

$$\begin{bmatrix} 22.58 & 12.527393 & 9 & 99 \end{bmatrix}$$

Explaining part by part, the mean of the 500 simulations is 22.58 steps lower than the hard coded with a Standard deviation of 12.52 greater than the hard coded, this result in less accurate and a high variance model, in addition it that the min-max interval is bigger than the the hard coded this result is a agent that is not reliable, due sometimes can get a result above the hard coded, or sometimes fail and getting even less additionally the calculations done exceed the time the hard coded policy get, so this random approximation became inefficient and imprecise, both characteristics valued in an AI model

4. Gradient Policy

4.1. Adjusting the Network

in the previous section the given approximation to the network was not as reliable as we desire so we medicate the policy in where we define the target probability so if an action is good increase the probability and if an action is bad we reduce the probability the problem in this particular case is that most actions does not have visible consequences in the next step or in the next 10 and is not clear to know what actions contributed to the result this is called the *"Credit assignment problem"*

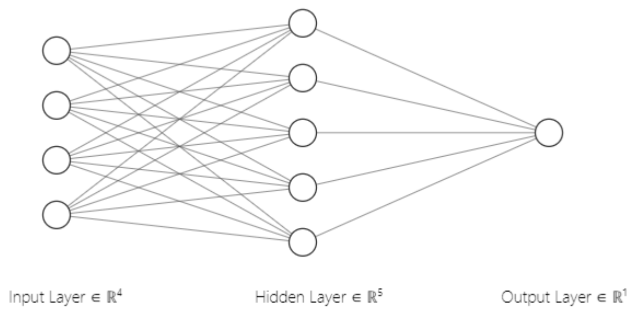

Figure 3. *basic Neural Network architecture*

besides this modification the network architecture still the same with the full environment observation input and one probability output

4.2. Credit Assignment

the problems is tackle with a method that come from a mathematical result called gradient descent.[2]

$$a_{n+1} = a_n + \gamma \nabla F(a_n)$$

the function defined is a recursive call that takes the previous state and adjust a γ that is the step of the gradient by simulating many scenarios and penalizing those whose rewards are better making them more probable (γ greater the 1) in future instances and penalizing those who are worst (γ less than 1)this way the agent will take better decisions without losing the *"exploration - Exploitation"* factor

```
def discount_rewards(rewards, discount_factor):
    discounted = np.array(rewards)
    for step in range(len(rewards) - 2, -1, -1):
        discounted[step] += discounted[step + 1]
     * discount_factor
    return discounted

def discount_and_normalize_rewards(all_rewards,
    discount_factor):
    all_discounted_rewards = [discount_rewards(
    rewards, discount_factor)
                              for rewards in
    all_rewards]
    flat_rewards = np.concatenate(
    all_discounted_rewards)
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean) /
     reward_std
```

```
                for discounted_rewards in all_discounted_rewards]

optimizer = tf.keras.optimizers.Nadam(learning_rate=0.01)
loss_fn = tf.keras.losses.binary_crossentropy

for iteration in range(n_iterations):
    all_rewards, all_grads = play_multiple_episodes(
        env, n_episodes_per_update, n_max_steps, model, loss_fn)

    total_rewards = sum(map(sum, all_rewards))
    print(f"\rIteration: {iteration + 1}/{n_iterations},"
          f" mean rewards: {total_rewards / n_episodes_per_update:.1f}", end="")

    all_final_rewards = discount_and_normalize_rewards(all_rewards,
                                                       discount_factor)
    all_mean_grads = []
    for var_index in range(len(model.trainable_variables)):
        mean_grads = tf.reduce_mean(
            [final_reward * all_grads[episode_index][step][var_index]
             for episode_index, final_rewards in enumerate(all_final_rewards)
                 for step, final_reward in enumerate(final_rewards)], axis=0)
        all_mean_grads.append(mean_grads)

    optimizer.apply_gradients(zip(all_mean_grads, model.trainable_variables))
```

Code 4. *Gradient Policy*

by simulating 150 iterations of this learning policy we get a mean of 187.9 rewards a much better result than the previous but with a very high time cost even simulating less than the half the compute time goes similar to the random neural network but with better level of proficiency

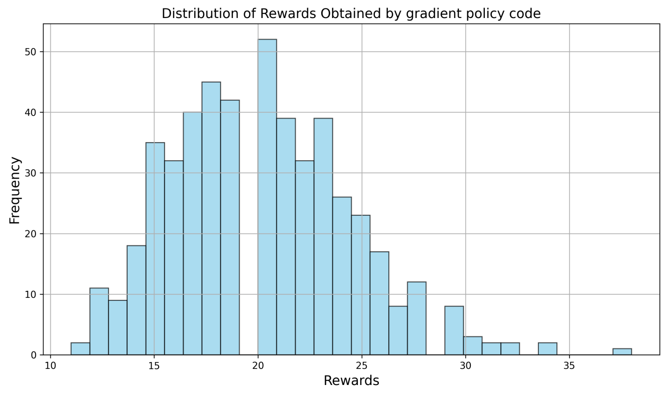

Figure 4. *gradient policy neural Network distribution*

5. Q-Learning

Q-learning or Quality Learning is a way an intelligent agent learn a series of rules on what to do in particular situations, Q-learning finds an optimal policy where it maximize the results, the way of optimizing is to find the Q-values "Quality actions" to a state action pair $Q(a, s)$ [1]

5.1. Q-Learning model

Mathematically the model is described as follow

$$Q(a, s) = (1 - \alpha)Q(a, s) + \alpha(R * \gamma * max_{a'}Q(s', a'))$$

where

- $Q(a, s)$ is the Q-value for a state action pair
- α is the learning factor (controls the weight of the new information)
- R is the immediate reward
- γ is the discount factor (controls the importance of future rewards)
- s' is the next state
- a' is the next action

this way the intelligent agent explore the environment and adjust the values for α and γ until converge to an optimal policy that maximize the expected cumulative reward

5.2. Building a Deep Q-Network

Now the approach is to take all the observations of the environment and transform them in to an output that's either move left or right

```
tf.random.set_seed(42)  # extra code   ensures reproducibility on the CPU

input_shape = [4]  # == env.observation_space.shape
n_outputs = 2  # == env.action_space.n

model = tf.keras.Sequential([
    tf.keras.layers.Dense(32, activation="elu", input_shape=input_shape),
    tf.keras.layers.Dense(32, activation="elu"),
    tf.keras.layers.Dense(n_outputs)
])
```

Code 5. *Q-Learning*

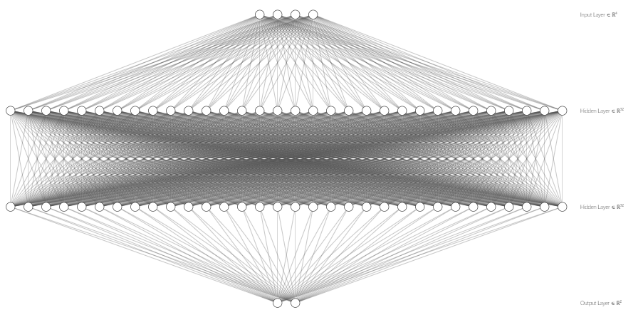

Figure 5. *Deep Q-Learning neural Network visual Representation*

To select an action the Network will pick the action with the largest Q-Value, however keeping the Exploration-Exploitation the agent will choose a random action with probability epsilon

```python
def epsilon_greedy_policy(state, epsilon=0):
    if np.random.rand() < epsilon:
        return np.random.randint(n_outputs)  # random action
    else:
        Q_values = model.predict(state[np.newaxis], verbose=0)[0]
        return Q_values.argmax()
```

Code 6. *Q-Learning*

Now training the model

```python
DQNRewards = []
for episode in range(600):
    DQNeps_rewards = 0
    obs, info = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, truncated, info = play_one_step(env, obs, epsilon)
        DQNeps_rewards += reward
        if done or truncated:
            break

    DQNRewards.append(DQNeps_rewards)

    # extra code    displays debug info, stores data for the next figure, and
    #               keeps track of the best model weights so far
    print(f"\rEpisode: {episode + 1}, Steps: {step + 1}, eps: {epsilon:.3f}",
          end="")
    rewards.append(step)
    if step >= best_score:
        best_weights = model.get_weights()
        best_score = step

    if episode > 50:
        training_step(batch_size)

model.set_weights(best_weights)
```

Code 7. *Q-Learning training code*

the results are the following

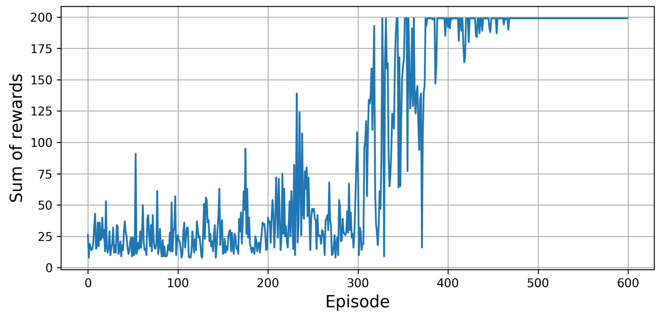

Figure 6. *Deep Q Network of rewards over iteration*

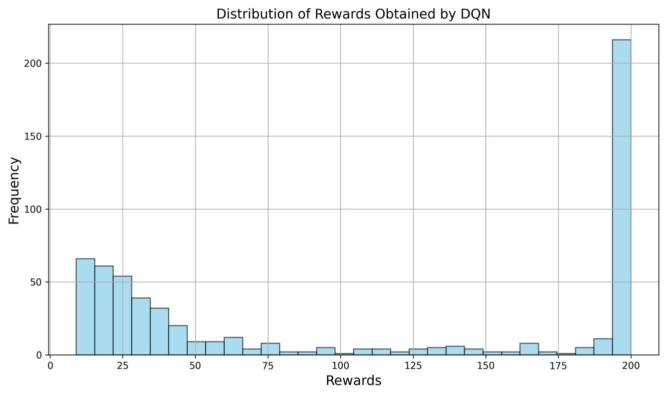

Figure 7. *Distribution of rewards*

as you can see in the figure the agent begin learning how to balance the pole at the episode 300, before that, many of the episodes played were to adjust the Q-values, and before the episode 400 the agent figures a way to keep the pole at least 150 steps, finally after the episode 500 we get perfect balance of 200 steps here we can say that the agent learned how to play the game

additionally the figure shows how the rewards are distributed closer to the objective than the previous policies the training time is longer due the multiple calculations done by the network

6. Conclusions

In conclusion, reinforcement learning stands as a vast and continually evolving field within artificial intelligence. Throughout our exploration, we have witnessed significant advancements propelled by innovative methodologies, leading to increasingly impressive outcomes. However, it is crucial to acknowledge that these achievements often come at the cost of escalating training times and computational demands.

As RL algorithms become more sophisticated and capable of handling complex tasks, the computational resources required for training have also seen a notable surge. The exponential growth in both the complexity of tasks tackled by RL agents and the corresponding computational requirements raises pertinent questions about scalability and feasibility, particularly for real-world applications.

6.1. GitHub

You can download the original notebook with a few modifications i did and the images i generated in the link bellow

Link:https://github.com/DiegoVilla03/Artificial-intelligence

References

[1] C. J. W. P. Dayan, "Q-learning", *https://doi.org/10.1007/BF00992698*, 1992.

[2] L. C. B. III, "Reinforcement learning through gradient descent", Ph.D. dissertation, School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213, 1999.

[3] A. Geron, "Chapter 18 – reinforcement learning", https://github.com/ageron/handson-ml3/blob/main/18$_reinforcement_learning.ipynb$, Tech. Rep., 2023.