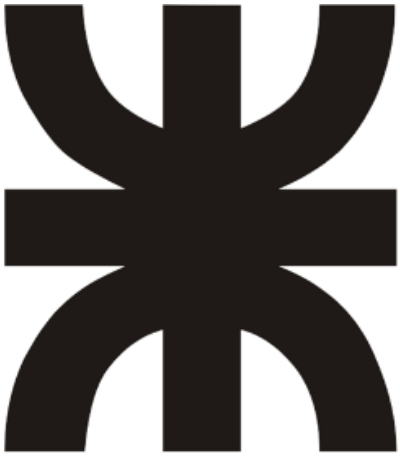


# UNIVERSIDAD TECNOLÓGICA NACIONAL

# FACULTAD REGIONAL VILLA MARÍA

## Ingeniería en sistemas de información

## Inteligencia Artificial

**Curso:** 5to Año

**Año:** 2021


**Integrantes:**

- Pramparo, Damián Legajo: 11939 damian_pramparo@hotmail.com

**Docentes:**

- Dr. Jorge A. Palombarini
- Ing. Juan C. Barsce


**Fecha de entrega: 09/07/2021**

# Contenido

# Desarrollo de trabajos prácticos – Inteligencia Artificial 2021

## 1 – Politicas

Las políticas a tomar comenzando en H son:

Izquierda (pasar a G)

Arriba (pasar a D)

Arriba (pasar a A)

Derecha (pasar a B)

Derecha (pasar a C)

¿Cuál sería el valor de v(D) bajo la Política 1 (asumiendo γ=1 )? Política 1: El agente elige la primera acción posible de la siguiente lista: [Derecha, Arriba, Izquierda, Abajo]

D -> A = -1 (Arriba)

D -> E = -5 (Derecha)

D -> G = -1 (Abajo)

Izquierda no aplica, ya que no es una posibilidad a dar estando en D

¿Cuál es el retorno Rt de un agente que, de acuerdo a su política de actuación, realiza el recorrido [D, E, B, C] desde St=D , asumiendo un descuento γ=0.1 ?

D -> E = -5 * 0.1 = -0.5

E -> B = -1 * 0.1 = -0.1

B -> C = +50 * 0.1 = 5

Retorno de Rt = 4.4

¿Cuál sería el valor de q(B,abajo) bajo la Política 1, asumiendo γ=1 ?

El valor de reward seria: -5

## 2 - SARSA con The Cliff

Como primera instancia, armo toda la estructura que existe en el notebook de la catedra, teniendo asi, la función *run, learn y chose_action*, donde debería construir la función de learn en base al libro de RL.

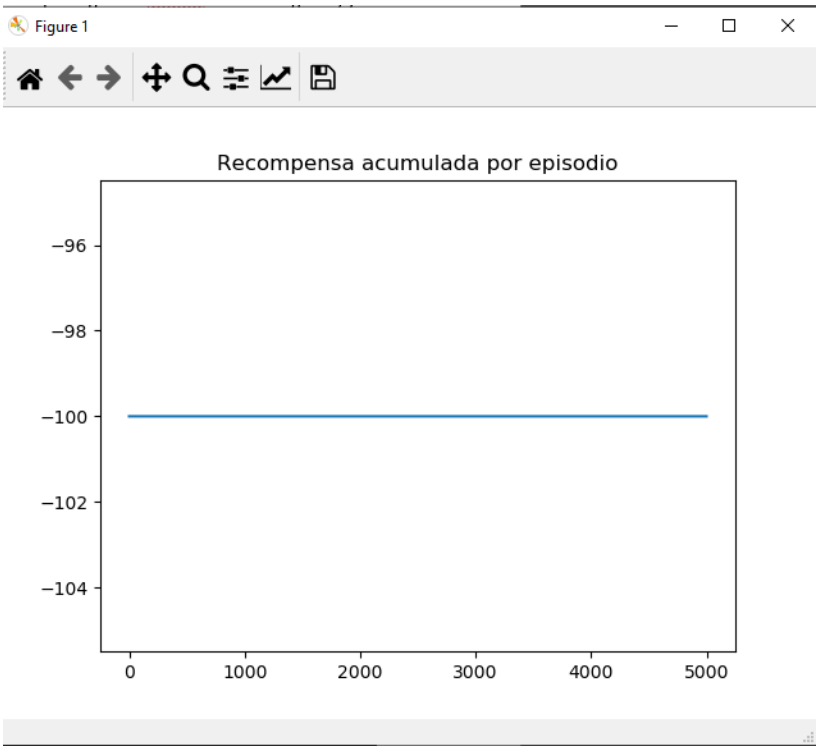
Cuando realizo la implementación, me doy cuenta que llega a un reward máximo negativo, y esto no baja, es decir, no se resetea el reward.

```python
####Funcion de aprendizaje
def learn(state, action, reward, next_state, next_action):
    """
    Dado un (estado, acción, recompensa, estado siguiente),
    realiza una actualización SARSA de Q(s,a)
    """

    q_back = q.get(state,action)
    q_next = q.get(next_state, next_action)
    q[(state,action)] = q_back + alpha * ( reward + gamma  * q_next - q_back)

    #q[(state, action)] = 0  # TODO - completa con tu código aquí


#Corro el script para ver su funcionamiento
```

TP 3 - RL - Damian_Pramparo

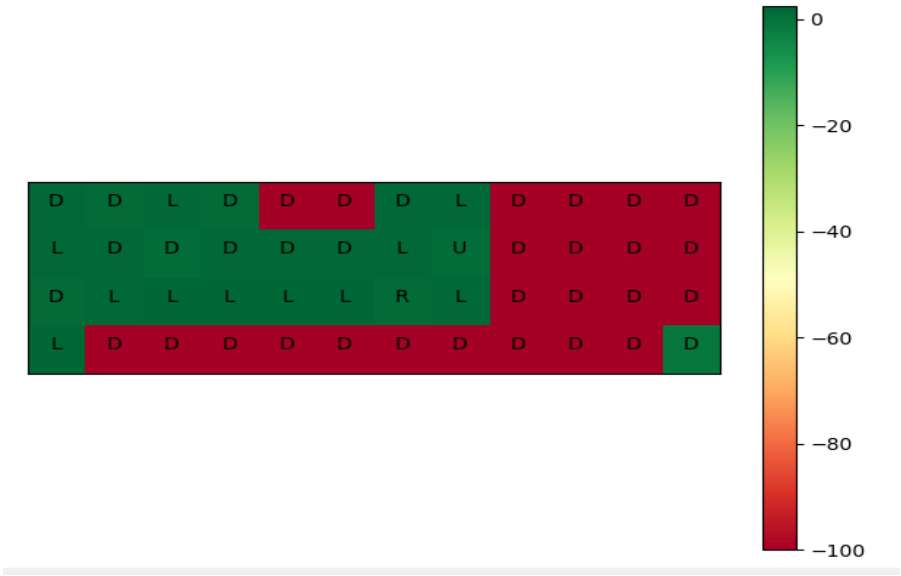
```
[Episode 1800] - Mean rewards -100.0.
[Episode 1800] - Mean rewards -100.0.
[Episode 1800] - Mean rewards -100.0.
[Episode 1800] - Mean rewards -100.0.
[Episode 1800] - Mean rewards -100.0.
[Episode 1800] - Mean rewards -100.0.
[Episode 1800] - Mean rewards -100.0.
[Episode 1800] - Mean rewards -100.0.
[Episode 1800] - Mean rewards -100.0.
[Episode 1800] - Mean rewards -100.0.
[Episode 1800] - Mean rewards -100.0.
[Episode 1800] - Mean rewards -100.0.
```

En donde realizar el grafico, obtengo que



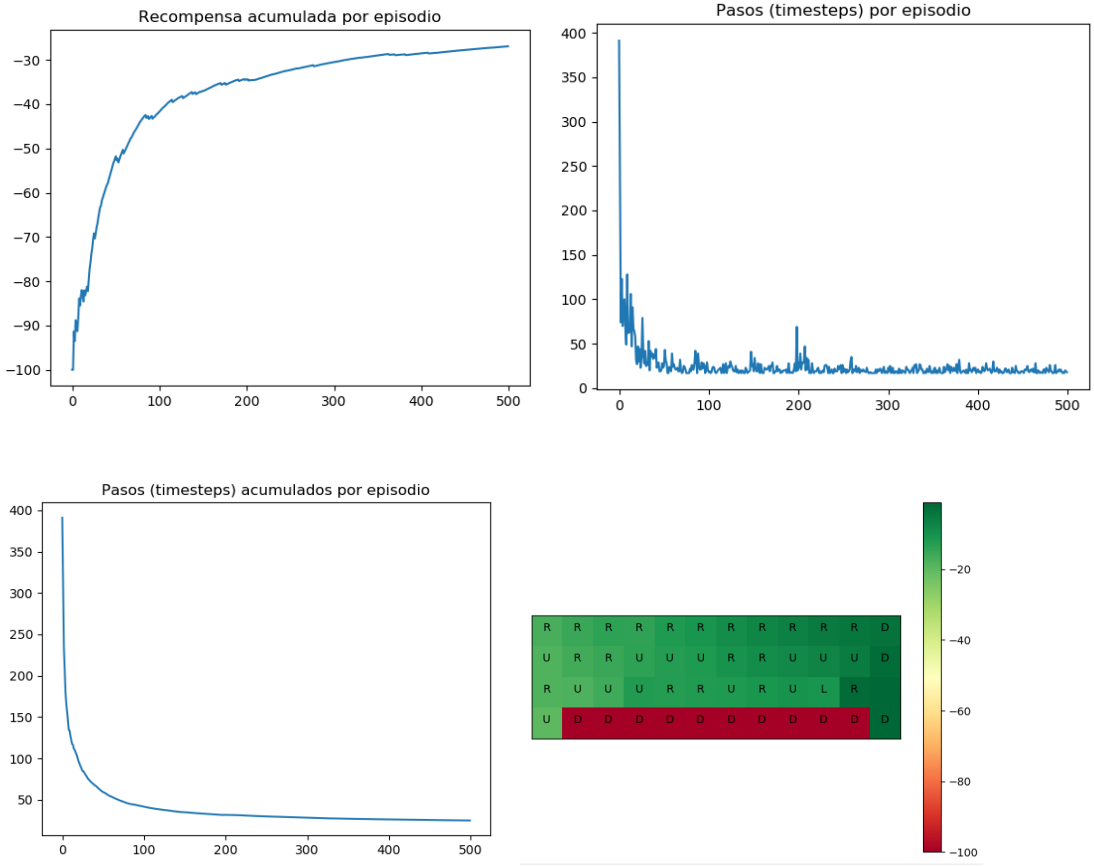

Pero, ya con 500 episodios, es suficiente, le puse 5000 para probar su comportamiento a más episodios.

Al revisar la matriz de política optima, no obtengo un buen resultado




El error estaba, que no obtenía el estado y acción actual, sino que siempre me lo iba acumulando. Luego de eso, vuelvo a cero lo que obtengo y los resultados mejoraron mucho

```
def learn(state, action, reward, next_state, next_action):
    """
    Dado un (estado, acción, recompensa, estado siguiente),
    realiza una actualización SARSA de Q(s,a)
    """

    q_back = q.get((state,action), 0.0)
    q_next = q.get((next_state, next_action),0.0)
    q[(state,action)] = q_back + alpha * (reward + gamma * q_next - q_back)
```



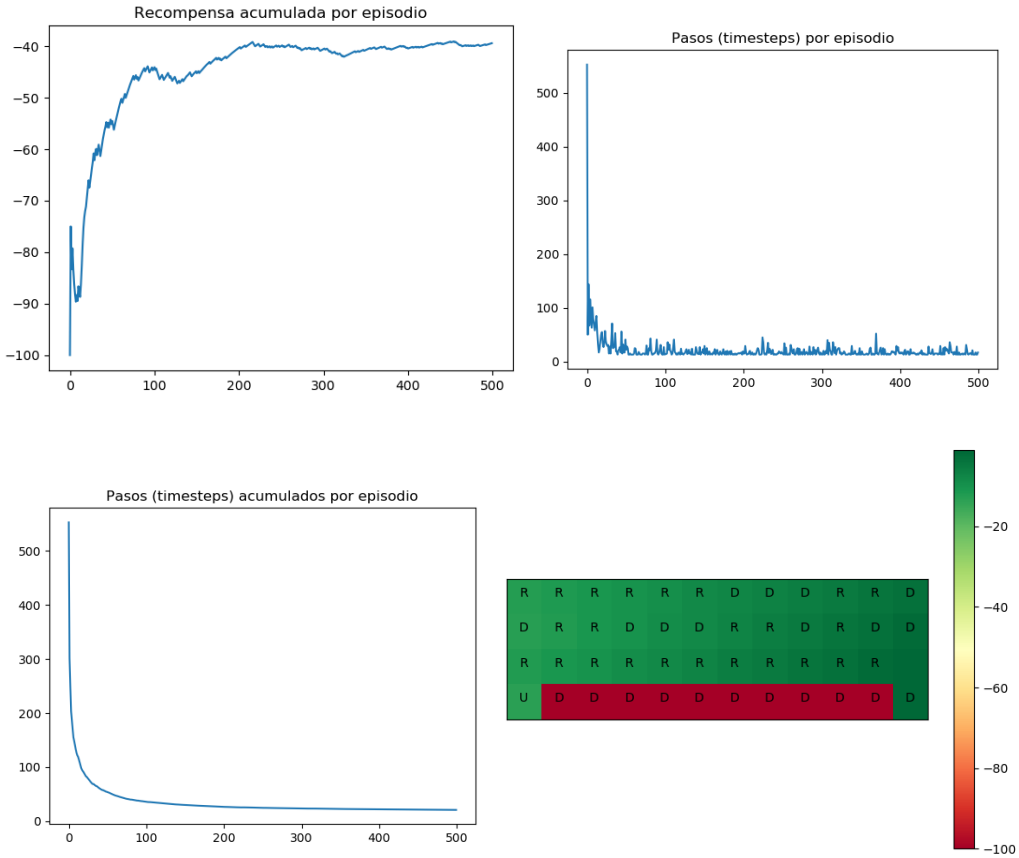

Como conclusión, puedo decir que SARSA, en estos entornos básicos, funciona demasiado bien, y hasta ahora no ha tenido problemas en The Cliff , con tan solo 500 episodios., Donde se ve que si aumento la cantidad por episodio, este mejora, pero ya con 300, obtendríamos algo optimo.

## 3 – Q Learning con The Cliff

Luego de realizar el primer punto, tomo la misma estructura de ejecución y realizo la función de **learn** para q learning, en donde teniendo en cuenta que este algoritmo al ser off policy y lo que hace es tomar la acción que dé más reward en el conjunto de estados.

```
def learn(state, action, reward, next_state, next_action):
    """
    Dado un (estado, acción, recompensa, estado siguiente),
    realiza una actualización SARSA de Q(s,a)
    """

    q_back = q.get((state,action), 0.0)

    q[(state,action)] = q_back + alpha * (reward + gamma * (max([q.get((next_state, a), 0.0) for a in
                            range(env.action_space.n)])) - q_back)
```

En donde, tras correrla, se puede ver que:




Este algoritmo, logra converger mucho más rápido que SARSA ya que trabaja de la forma mencionada anteriormente, utilizando la política máxima.

## 4 – Prueba de diferentes valores con SARSA

Utilizando:

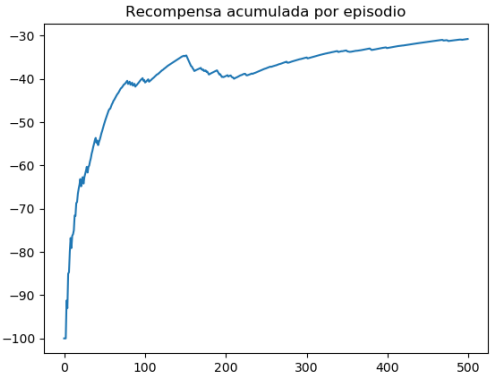
Alpha = 0.5

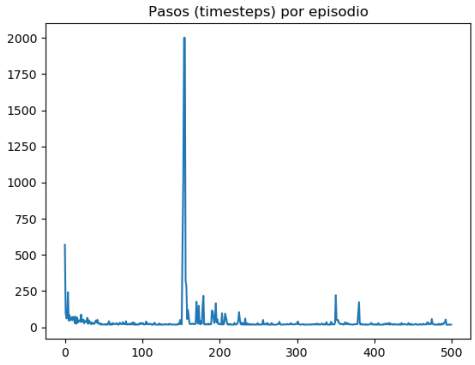
Gamma = 0.9

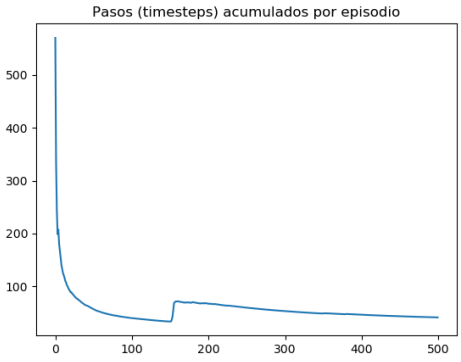
Épsilon = 0.1

Episodios = 500

Se obtiene





En donde vemos que, con estos valores, se logra una mejor convergencia de los diferentes con respecto a la versión original de SARSA, pero con un poco de inestabilidad.

Ahora, utilizando:

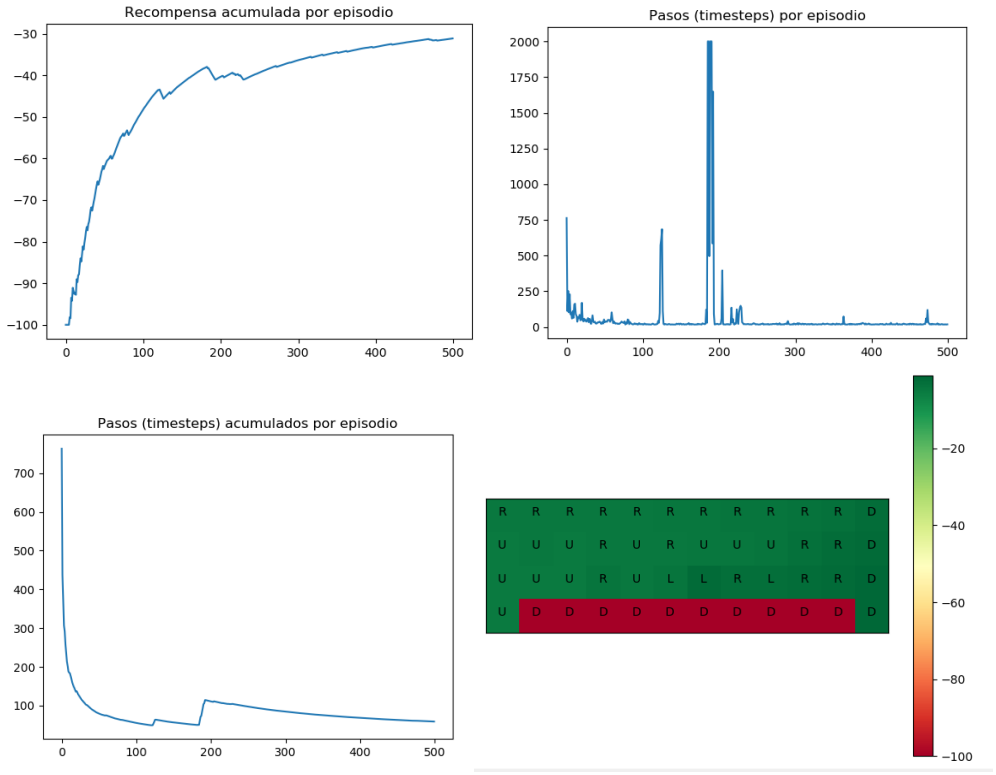
Alpha = 0.3

Gamma = 0.8

Épsilon = 0.1

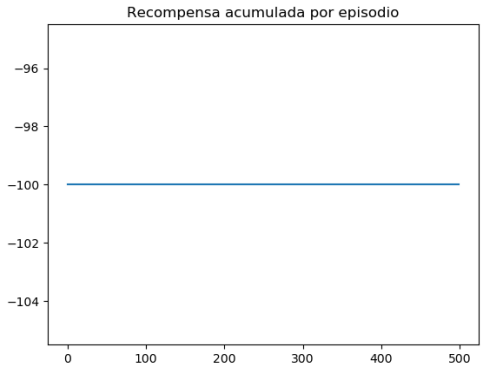
Episodios = 500

Vemos que

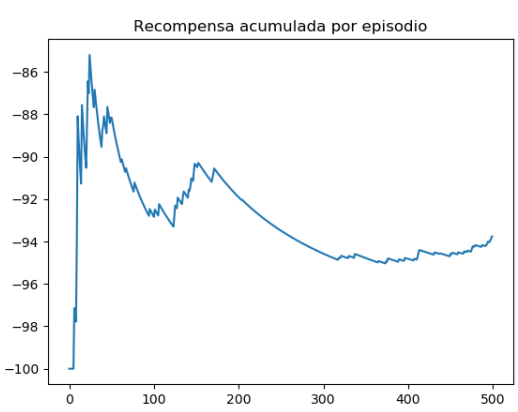


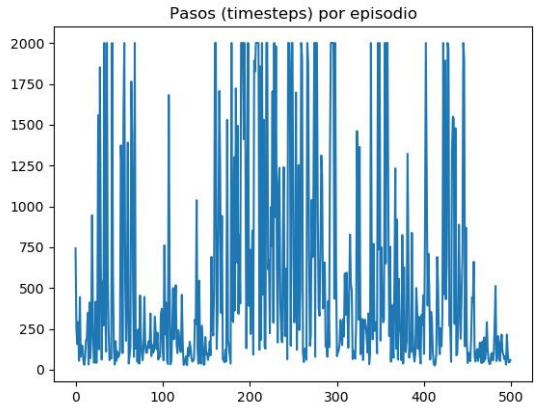

Hubo algunos cambios, pero esta configuración de hiperparametros no mejoro el algoritmo, sino que le generó mayor inestabilidad.

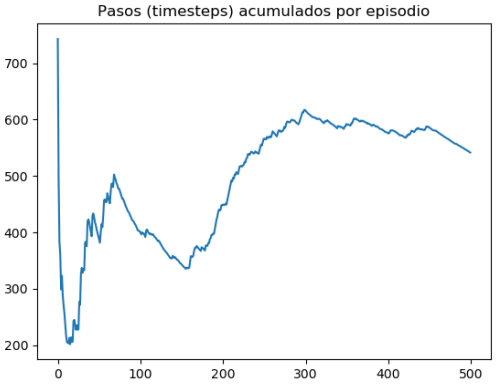
Cambiando épsilon = 1, el agente no explora nunca, siendo el entorno utilizado SARSA no es bueno poner el épsilon en 1, ya que nunca explora, sino que explota el conocimiento que nunca adquirio. Entonces, se obtiene esto:

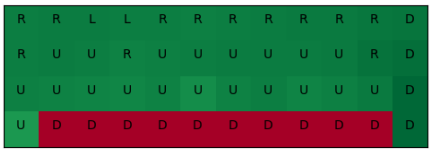


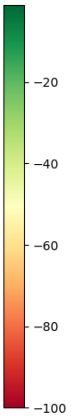

Probando con épsilon = 0.5, podremos ver que hay cambios en el reward obtenido:



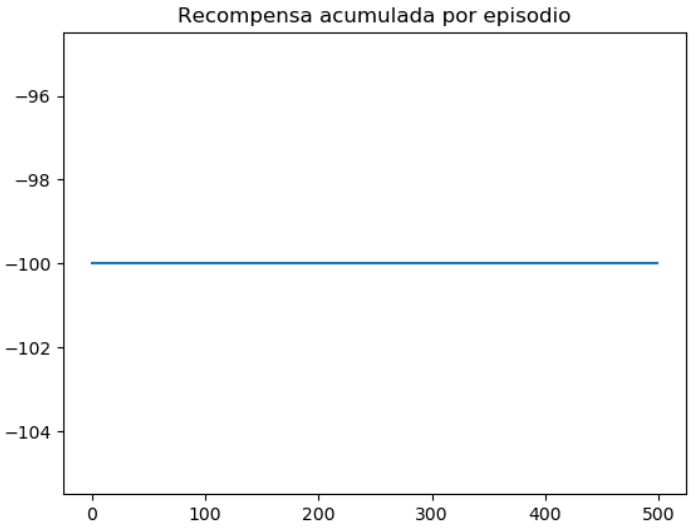




Vemos que con un 50% de exploracion-explotacion, tenemos cambios para mal, en donde el algoritmo necesitaria mas episodios para poder seguir, pero es mejor que utilizar e = 1.

Haciendo uso de la funcion SoftMax, con un tau de 50, no se obtienen mejoras, el algoritmo no converge nunca.

```
def choose_action_softmax(state):
    """
    Elige una acción según el aprendizaje realizado, usando una
    política de exploración softmax
    """

    q_values = [q.get((state, a), 0.0) for a in actions]

    softmax_q = np.exp(np.array(q_values) / tau)/ np.sum(np.exp(np.array(q_values)/tau))
    estado_random = random_state.uniform()
    for i in (range(len(softmax_q))):
        if estado_random < softmax_q[i]:
            return actions[i]
    return actions[i]
```




## 5 – Creación de entornos personalizados

Para realizar este punto, realice un entorno de grilla, en el cual es parametrizado para poder ver cómo funciona, con unas 4 acciones, una pared y un goal.

Lo corro en el archivo *5-a.py* en donde importa el entorno creado en el archivo *EnvGoLeft*.

En el primer archivo mencionado, quedo de la siguiente manera, haciendo uso de PPO.

```
from EnvGoLeft import GoLeftEnv, GoGrid
from stable_baselines3.common.env_util import make_vec_env
from stable_baselines3 import DQN, PPO

env = GoGrid(width=4, height=4)
env = make_vec_env(lambda: env, n_envs=1)

model = PPO('MlpPolicy', env, verbose=1).learn(20000)


obs = env.reset()
for i in range(10):
    action, _states = model.predict(obs)
    obs, rewards, dones, info = env.step(action)
    env.render(mode='console')
    if dones[0]:
        obs = env.reset()
```

En el archivo EnvGoLeft, se puede ver la clase GoGrid, que es creada por mí, tomando como referencia el ejemplo dado en clase para poder probar un poco como hacer un entorno propio, y la verdad que ha sido una muy buena experiencia, ya que me ayuda por si alguna vez, necesito entrenar un agente en algún entorno que no se encuentra creado por la comunidad. Obviamente, hay muchos entornos mucho mas sofisticados para poder probar, pero en casos especiales, esta es una buena opción.

## 6 – Entrenamiento de entornos mas complejos con PPO y DQN

Al querer realizar stable-baseline-zoo en Windows, me encontré con un problema de directorios, ya que la estructura es diferente a la utilizada en Linux.

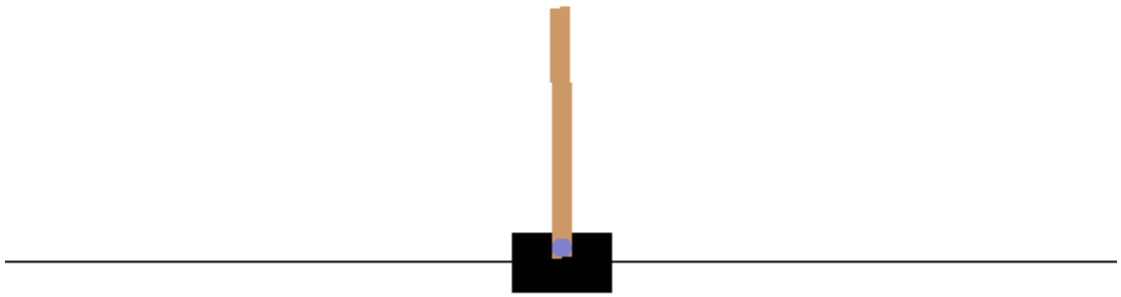
Realizare las pruebas con algoritmos tabulares para comprobar el funcionamiento de algoritmos más complejos.

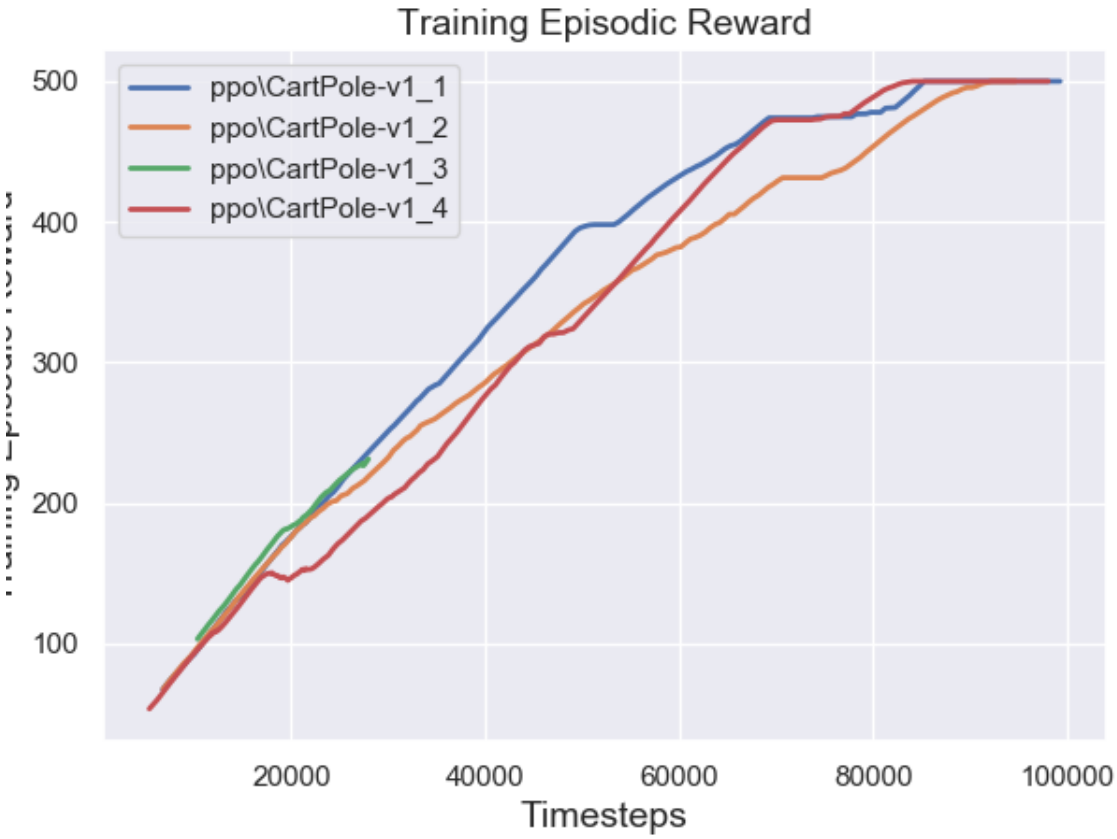
Al probar entornos más complejos en algoritmos tabulares, no funcionan, ya que hay que adaptarlos para que estos puedan correr de manera correcta.

Al regresar y debuggear el error, me encuentro que era una librería que me faltaba instalar.

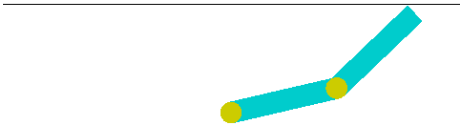
En primera prueba, realizo con las siguientes características:

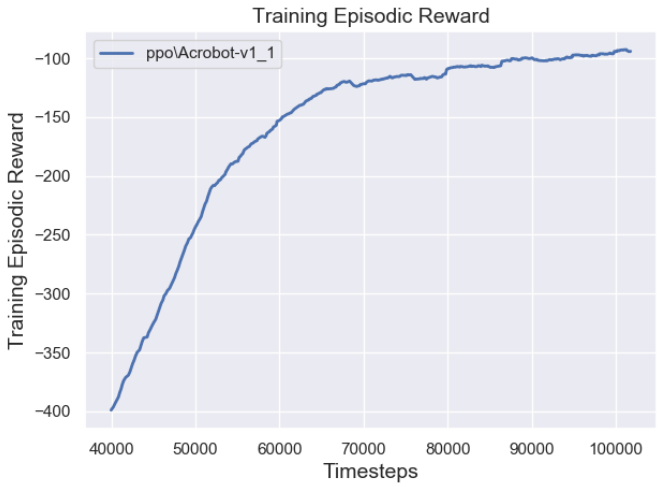
- Stablebaseline3-zoo
- **Algoritmo**: PPO
- **Entorno**: CartPole-v1



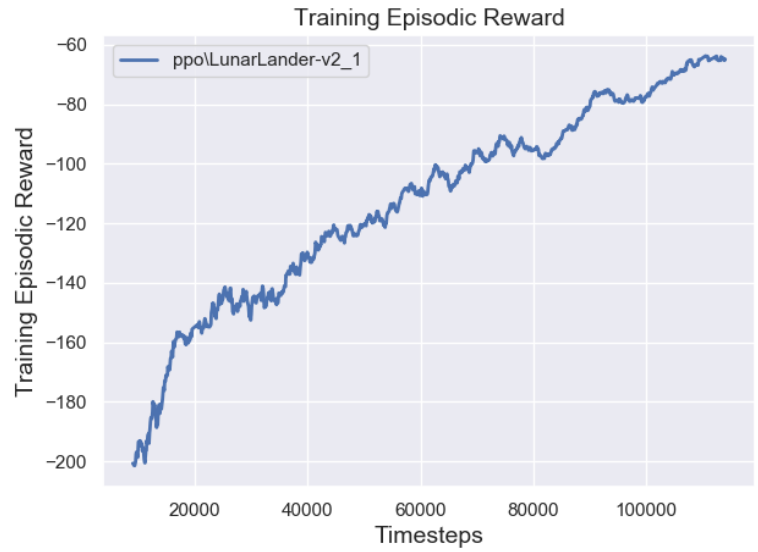


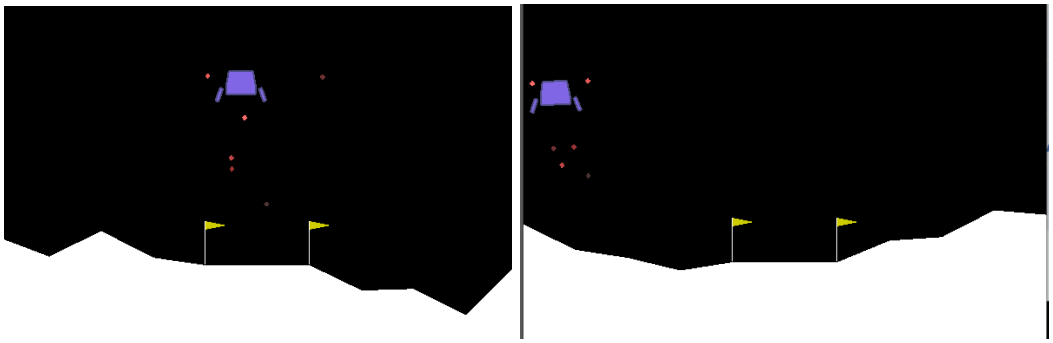

- Stablebaseline3-zoo
- **Algoritmo**: PPO
- **Entorno**: Acrobot-v1





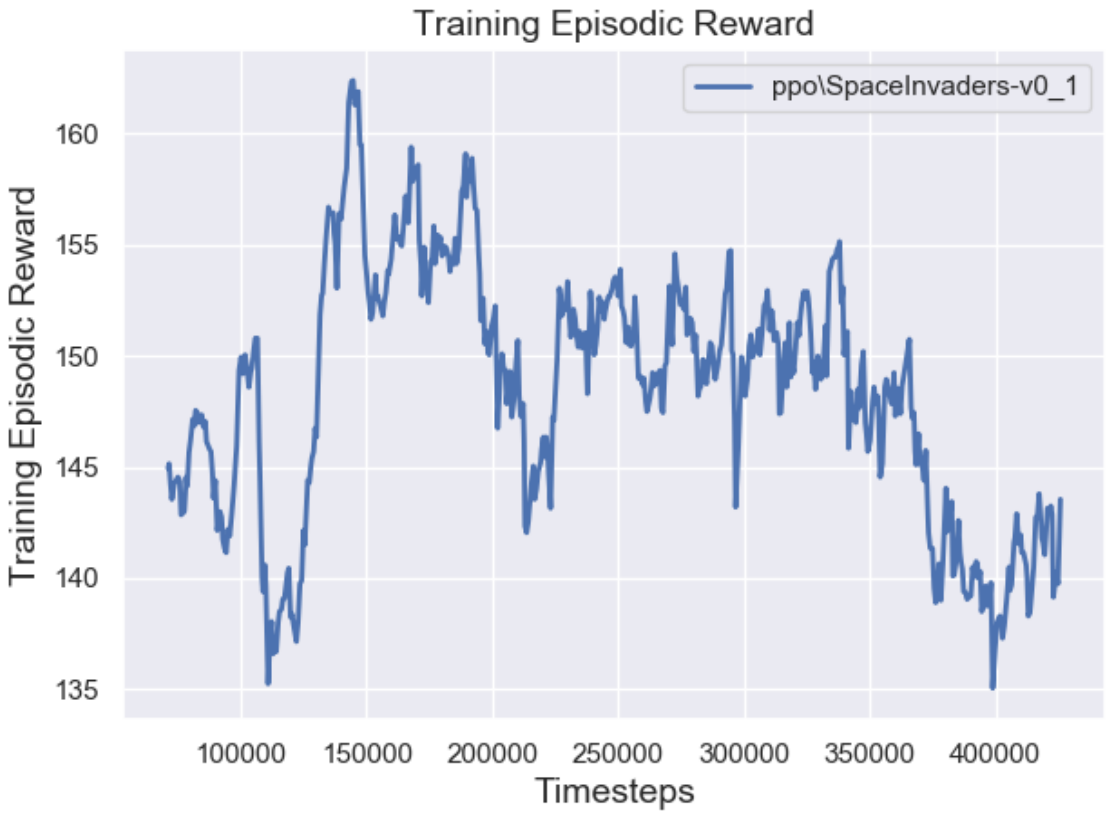

- Stablebaseline3-zoo
- **Algoritmo**: PPO
- **Entorno**: LunarLander-v2



Por ultimo, la prueba de fuego:

- Stablebaseline3-zoo
- **Algoritmo**: PPO
- **Entorno**: SpaceInvader-v0




A modo de aclaración, he corrido todos estos entornos en mi PC sin contar con GPU para poder correr entornos muchos mas complejos, como por ejemplo *SpaceInvader-v4* que era mi meta, pero ya con ver que el v0 tardo dos horas en terminar, no quise realizar uno de mayor complejidad y solo me concentré en ver y probar los que realicé.

Ha sido una experiencia gratificante, al ver como un algoritmo logra controlar un juego, como por ejemplo *LunarLander-v0,* que se puede ver tranquilamente en modo de renderización.