Documentación - Mountain Car

Santiago Irazoqui - 234730

Link al proyecto de wandb
Hacer público el proyecto de wandb es medio raro (no lo logre), asi que si quieren ver las runs y eso creo que la única manera es que me pasen el mail de la cuenta de algún profesor/ayudante y que lo invite al team. Ahí pueden ver todos mis proyectos.

Link al repo
Los colores de las runs intentan simbolizar un poco qué ocurrió: verde es que salió todo bien, se cumplió el propósito de la run, o logre un Q mejor. Rojo es que cometí un error y la run ya no sirve tanto. Amarillo puede ser por resultados raros o objetivos no cumplidos, y azul como extra.

Nota: hay una notebook general (no en la subcarpeta de notebooks) con algunos algoritmos en general. El resto de las notebook estan medio desordenadas entre todos los tests, wandb, etc.

Análisis previos

Límites de las discretizaciones

Para saber qué límite de las discretizaciones usar, primero vamos a ir a buscar los datos a Gym:

Action Space	Discrete(3)
Observation Shape	(2,)
Observation High	[0.6 0.07]
Observation Low	[-1.2 -0.07]
Import	`gym.make("MountainCar-v0")`

Posición

Luego en clase comenzamos a probar, dada una discretización, sobre qué partición cae cada posición del carro (nos interesan las posiciones límite). Por ejemplo, si el carro se posiciona en la cota superior, o más adelante, siempre cae en la última partición. Así sabemos que la división es: $[max, \ +\infty)$. Seguimos probando y llegamos a que:
$(-\infty, min)[min, \ldots)\ldots\ldots [max, \ +\infty)$

Esto es importante ya que con esto nos hacemos una idea mejor de cómo podríamos discretizar. Por ejemplo, si nosotros utilizamos la partición con cota superior 0.6, en realidad el carro nunca va a llegar hasta esa zona. El high es 0.6, y la partición va de $[0.6, \ +\infty)$, por lo que todo ese estado va a ser siempre 0. Por lo tanto, bajamos el límite a 0.5 para que de allí en adelante sea todo el mismo estado.

Esto también lo hicimos para la cota inferior.

```
pos_space = np.linspace(-1.1, 0.5, 10)
vel_space = np.linspace(-0.07, 0.07, 6)
```

Elegir un estado para chequear su valor y así construir una gráfica

Para esto, en cada iteración de qLearning yo tomo el estado inicial y guardo el q.

Primeros intentos, exploración

Para los primeros intentos utilicé la discretización (no hay razón particular para la elección de estos números.):

```
pos_space = np.linspace(-1.1, 0.5, 33)
vel_space = np.linspace(-0.07, 0.07, 3)
```

Ninguno de estos intentos llegó a un resultado decente. El problema es que para que Q comienza a mejorar y propagar los cambios, tiene que llegar por lo menos una vez. Yo en estos intentos le estuve dando muy poco tiempo a Q para que llegara, al rededor de 4000 iteraciones de qLearning cada vez que corría train (y unas 10 meta iteraciones, por lo que solo 40000 iteraciones en total).

El hecho de que nunca llegue también se puede deber a que la discretización no es buena, pero es difícil saberlo solo con estos tests.

Lograr que termine

Por lo tanto el plan es:
→ Hacer suficientes iteraciones

→ Utilizar un epsilon alto, para favorecer la exploración
→ El gamma reduce exponencialmente. Por lo tanto se necesita un valor muy grande para que el lookahead sea suficiente.
→ Alpha también va a ser grande, ya que quiero darle importancia a los nuevos estados junto con la exploración.
→ una vez que termine, guardo el Q en un pickle para seguir con los estudios

Por ejemplo:

```
iterations = 100000
all_iterations.append(iterations)

alpha = 0.99 #variacion permitida
all_alphas.append(alpha)

epsilon = 0.999 # -> más grande mas exploracion
all_epsilons.append(epsilon)

gamma = 0.999999  #futuro - learning rate - más bajo más greedy
all_gammas.append(gamma)
```

Probar distintas discretizaciones

La configuración anterior no terminó ni después de 1 millón y más de repeticiones de pura exploración, por lo que el problema está seguramente en la discretización.

La primera que logre que terminará en poco tiempo fue la siguiente:

```
divPos = 33
divVel = 11
self.pos_space = np.linspace(-1.1, 0.5, divPos)
self.vel_space = np.linspace(-0.07, 0.07, divVel)
```

Luego de cambiarla, llegó a ganar por lo menos una vez en menos de 400000.

Primera discretización

Decay de los valores

Epsilon

El primer estudio que consideré hacer es manejar el epsilon o exploration rate. El exploration rate define la distribución de probabilidad que la técnica epsilon-greedy va a utilizar para elegir acciones random (explorar) o acciones conocidas (explotar). Básicamente, se toma:
$P(explorar) = \varepsilon$
$P(explotar) = 1 - \varepsilon$

Por lo tanto, tiene sentido que baje el epsilon. Si lo pensamos como si fuera una persona, también tiene sentido: cada episodio, cada vez que jugamos de nuevo), exploramos un poco al principio, pero conforme avanza el juego recaemos en lo que ya conocemos para intentar ganar.

La única duda es cómo hacerlo bajar. Al principio lo bajaba manualmente, pero esto tenía malos resultados.

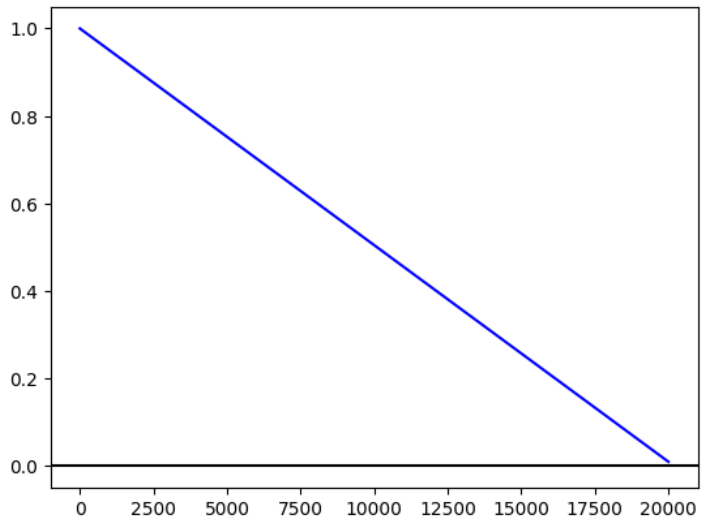
Por lo tanto busque una función que hiciera decaer el epsilon con el paso del tiempo:

Primera forma de bajar el epsilon:

```
final_epsilon = 0.01
epsilonDecreaseRate = max(((max_iterations - count) / max_iterations), 0)
epsilon = ((initial_epsilon - final_epsilon) * epsilonDecreaseRate) + final_epsilon
```

Esto lo estuve bajando en cada meta iteración de test (cada vez que corro el algoritmo de Q-Learning).

La gráfica del decay de epsilon queda de la siguiente manera:



Alpha, gamma

Teniendo esa función para decaer epsilon, intente también hacer decaer alpha y gamma (learnign rate y decay factor). Eso tiene un poco de sentido. A medida que avanzo baja el alpha, haciendo que los nuevos estados valgan menos. Lo mismo ocurre con gamma, que haría que al final del episodio el algoritmo sea más greedy, disminuyendo el lookahead y dejándolo local.

Hice dos runs de esta manera:
Primera run
Segunda run

No pude lograr que llegaran a algo decente. Siempre bajo el resultado de la policy optima.

Nueva discretización

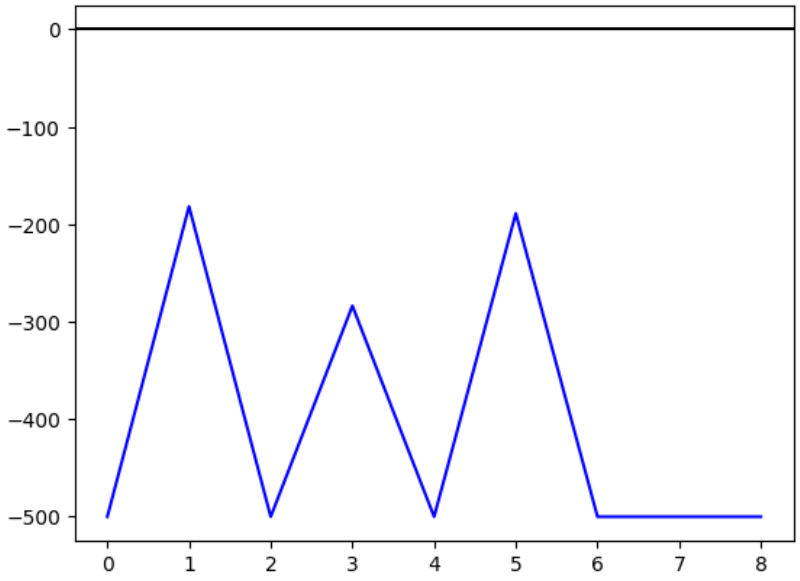
Viendo que no estaba pudiendo mejorar, decidí buscar una nueva discretización. Llegue a ver lo siguiente:

```
divPos = qLearningCar.env.observation_space.shape[0] #2
divVel = 10
```

Cuando lo probé, rápidamente me dieron buenos resultados. De nuevo, comencé solo explorando para hacer llegar al autito. Utilicé los siguientes parámetros:

```
alpha = 0.1
epsilon = 1
gamma = 0.99
```

Rewards promedio:



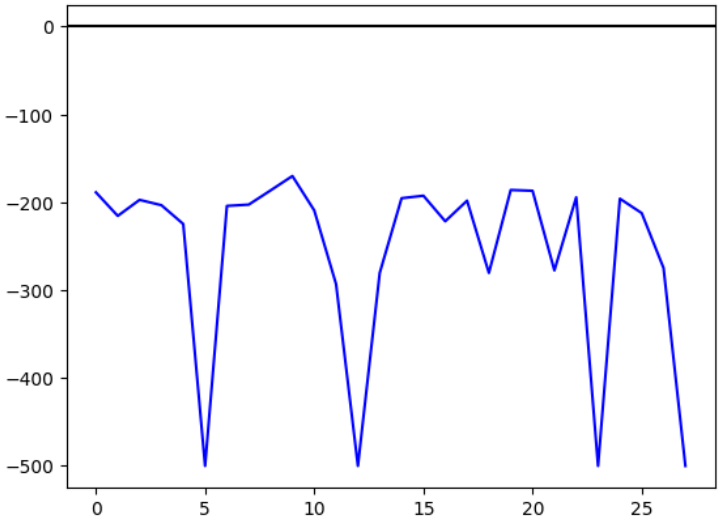
Run con la que encontré este Q

Decay epsilon

De nuevo, ahora que tengo un Q que llegó, voy a empezar a manejar los parámetros. Esta vez sí decidí ir modificando los parámetros de a uno.

Utilizando el mismo método que antes, ejecute muchas iteraciones, y termine llegando a un nuevo mejor Q con valor de -170.

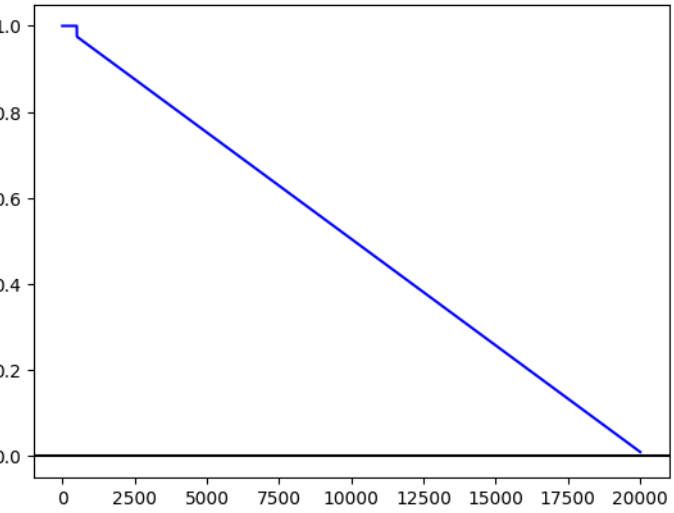
Rewards promedio:

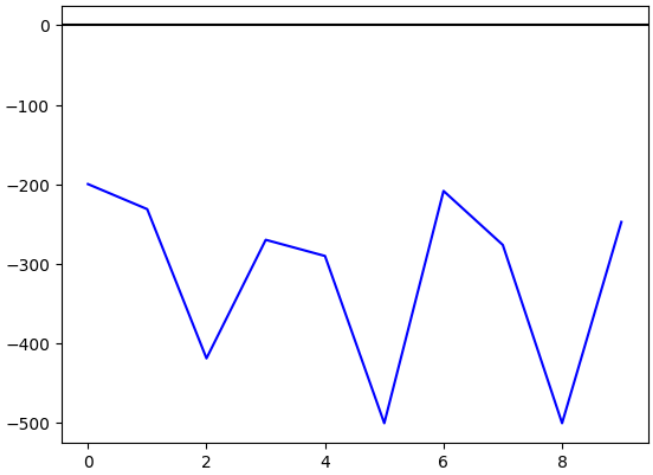


Probar dandole un mínimo de iteraciones

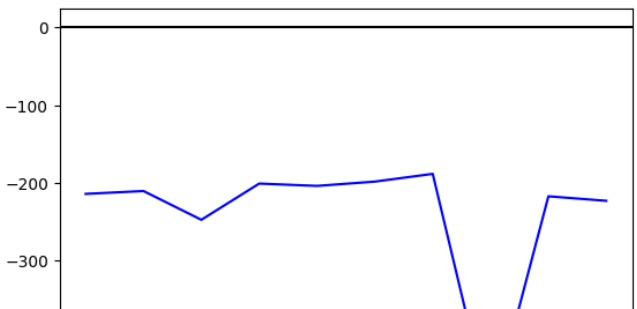
[También probé]() dando un mínimo de iteraciones al principio antes de que epsilon empezara a decaer.

Primero lo corrí con un mínimo de 500, pero no note mucha mejora





Luego lo corri con 1000 iteraciones mínimas, y pude ver que la variación fue menor, aunque puede haber sido de casualidad.



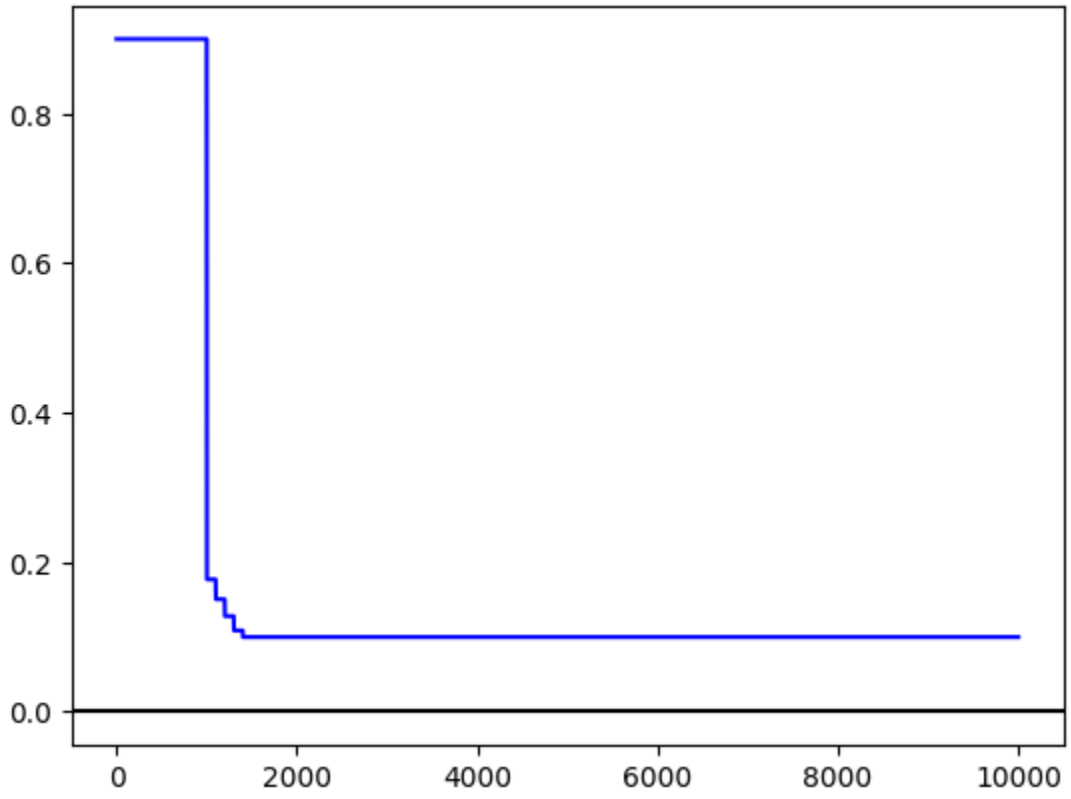
Decay Learning rate

Manejar el learning rate es un poco más difícil, ya que los efectos son menos claros. En definitiva define que variación se puede dar con los nuevos valores. Un learning rate muy grande haría que nuestro Q quede como un serrucho, ya que la alta varianza haría que en diferentes casos tenga valores más distanciados

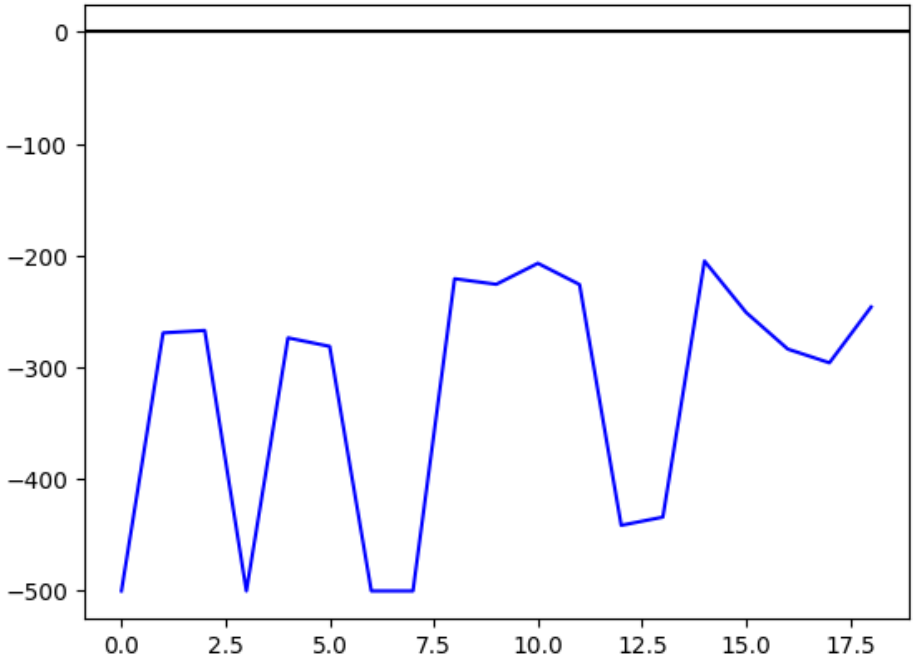
Por otro lado, un learning rate muy bajo antes de encontrar la policy óptima no tiene mucho sentido, ya que estaríamos dándole poca importancia a los nuevos valores, y por lo tanto demorando encontrar la policy óptima.

La idea es que me ayude a investigar nuevos caminos. El epsilon al principio hace que se explore siempre. A estas nuevas exploraciones se les da más importancia con el alpha alto al principio. De esta manera quizás pueda salir de los mínimos locales si es que caigo en ellos, y acelero encontrar la policy óptima.

Primera forma de decaer epsilon:

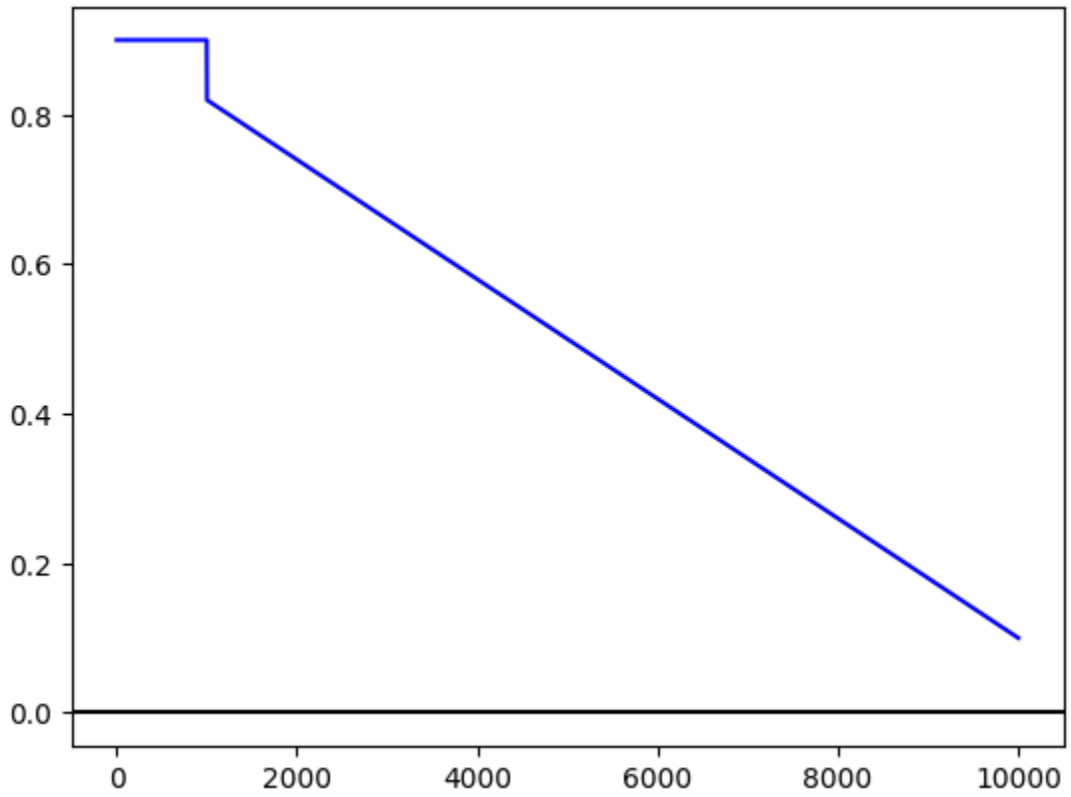


Fue demasiado empinada. Mirando los rewards, en mi opinión, fueron peores.

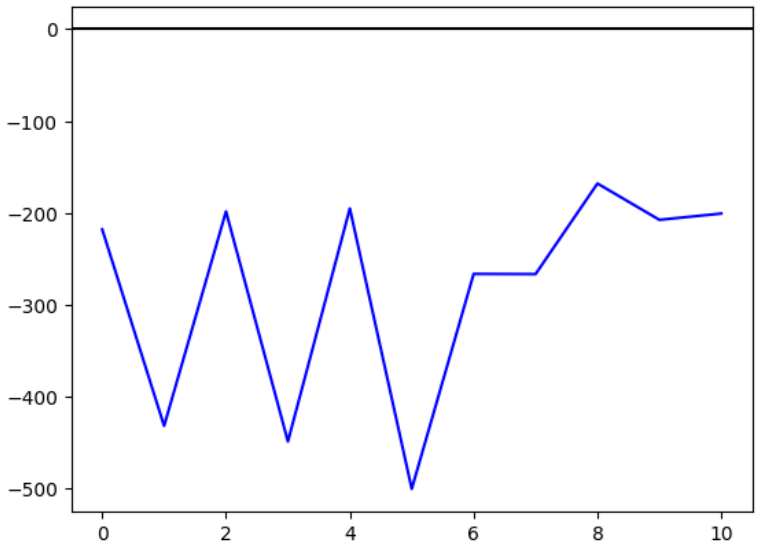


Segundo método de decaer alpha para learning rate:

Opte por un método lineal, con iteraciones mínimas al igual que con epsilon para exploration rate.



Viendo los resultados me fue dificil saber que está ocurriendo.



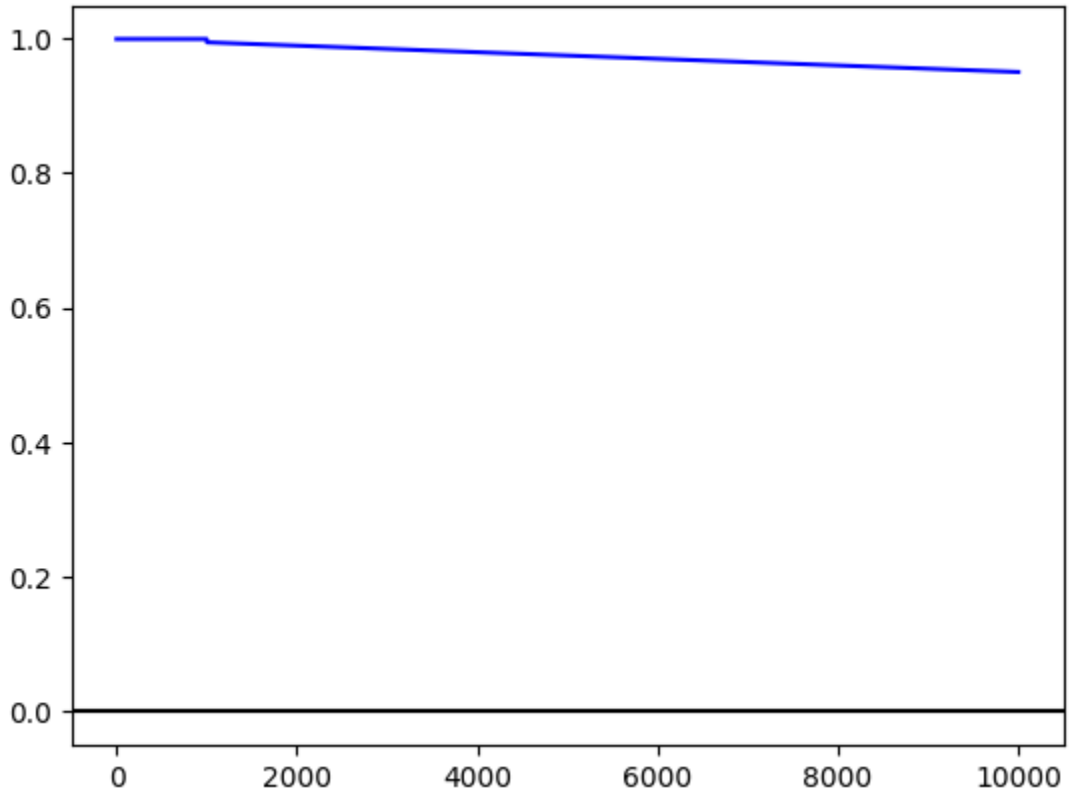
Por un lado, está aumentando la varianza bastante.

Pero también encontré el mejor Q hasta ese momento en esa run. Q = -167

Empeora la vdd - o no. conseguí el mejor hasta ahora con este (-167)

Decay Gamma

Intente variar el gamma de a muy poco, intentando que al final el algoritmo fuera más greedy. No tuve muchos resultados, y por temas de tiempo tampoco pude explorar mucho más esta opción.



Ver run en wandb

Test extra

He hecho otros tests que no me parece necesario mencionar, pero están en wandb si se quieren ver. Por ejemplo, cosas como variar el tamaño del batch size.

Best Q

El mejor resultado promedio que conseguí al final a partir de la policy óptima es: -129.8. El pkl se llama “Best_Q_-129.8”, en la carpeta de Best pickles

Lo conseguí con una discretización de (20, 20), iterando con un decay de epsilon.