Máster en Ingeniería Computacional y Matemática

Trabajo Final de Máster

Aplicación de algoritmos de Reinforcement Learning a juegos

Eduard Bermejo
Inteligencia Artificial

Samir Kanaan Izquierdo

31/05/2017

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Aplicación de algoritmos de Reinforcement Learning a juegos*
Nombre del autor:	*Eduard Bermejo Fernández*
Nombre del consultor/a:	*Samir Kanaan Izquierdo*
Nombre del PRA:	
Fecha de entrega (mm/aaaa):	05/2017
Titulación::	*Ingeniería Computacional y Matemática*
Área del Trabajo Final:	*Inteligencia Artificial*
Idioma del trabajo:	*Castellano*
Palabras clave	*Machine learning, reinforcement learning*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*

Este proyecto centra sus esfuerzos en hacer un repaso de la literatura existente sobre Reinforcement Learning así como aplicar los algoritmos de este campo a juegos sencillos. Se ha utilizado el toolkit OpenAI Gym que contiene el emulador de ATARI para realizar los experimentos. Los juegos utilizados son BlackJack, FrozenLake, MountainCar, Breakout y Pong. Desde los algoritmos más sencillos a los más complejos, se observa que cada uno de ellos puede aplicarse sobre diferentes problemas y que dependiendo de la naturaleza y complejidad del problema unos funcionan mejor que otros. Una conclusión importante es la predominancia que demuestran los métodos de Temporal Difference Learning sobre los de Monte Carlo y que se debería obtener la optimalidad en los problemas complejos combinando métodos policy based junto con los value based. Este trabajo ha dejado muchos métodos sin explorar debido a la gran diversidad de enfoques, algoritmos y tricks que se utilizan en esta disciplina y es por ello que para tener una visión más completa se continuará con el análisis de otros algoritmos. Un reto que ha surgido durante el proyecto es también la complejidad que requieren algunos algoritmos para la convergencia hacia un mínimo local junto con el largo tiempo de computación requerido para ejecuar estos experimentos.

Abstract (in English, 250 words or less):

This project focuses its efforts on reviewing the existing literature on Reinforcement Learning as well as applying the algorithms of this field to simple games. The OpenAI Gym toolkit containing the ATARI emulator has been used to perform the experiments. The games used are BlackJack, FrozenLake, MountainCar, Breakout and Pong. From the simplest algorithms to the most complex ones, it's been observed that each of them can be applied on different problems and depending on the nature and complexity of the problem some might work better than others. An important conclusion is the predominance of the methods of Temporal Difference Learning over Monte Carlo and that optimality should be obtained in complex problems combining policy based methods together with value based ones. This work has left many methods unexplored due to the great diversity of approaches, algorithms and tricks that are used in this discipline and that is why in order to have a more complete vision on the topic with the analysis of other algorithms will continue. A challenge that has emerged during the project is also the complexity required by some algorithms for convergence to a local minimum along with the long computing time required to execute these experiments.

Índice

1. Introducción 2
2. Reinforcement Learning 3
  2.1. Métodos para resolver Reinforcement Learning 5
  2.2. Métodos de Monte Carlo 5
    2.2.1. Monte Carlo Policy Evaluation 5
    2.2.2. Estimación de los valores de las acciones con Monte Carlo 6
    2.2.3. Monte Carlo Control 7
    2.2.4. On-policy Monte Carlo Control 8
    2.2.6. Off-policy Monte Carlo Control 9
  2.3 Temporal-Difference Learning 10
    2.3.1. Optimalidad de TD(0) 12
    2.3.2. On-policy TD Control (SARSA) 13
    2.3.3. Off-policy TD Control (Q-Learning) 14
  2.4. Métodos de aproximación de funciones 14
    2.4.1. Métodos Lineales 15
    2.4.2. Métodos no Lineales 15
    2.4.3. Aproximación de la policy 16
3. Juegos utilizados: 17
  3.1. BlackJack 17
  3.2. FrozenLake 17
  3.3. MountainCar 18
  3.4. Pong 18
  3.5. Breakout 19
4. Experimentos 20
  4.1. On-policy Monte Carlo Control para BlackJack 20
  4.2. FrozenLake (Monte Carlo on-policy) 24
  4.3. FrozenLake: Q-Learning 26
  4.4. Function approximation: Mountain Car - Sarsa 27
  4.5. Q-Learning: 29
  4.6. Deep Q learning 31
  4.7. Double Deep Q-Network 34
  4.8. Dueling Deep Q-Network 34
  4.9. Policy Gradient 35
5. Tecnologías utilizadas 38
6. Diagrama de Gantt 39
7. Conclusiones 40
8. Bibliografía 41

Índice de figuras

Diagrama de la interacción entre el agente y el entorno 3
First-visit MC para la estimación de la función de valor de una determinada policy 6
Iteración sobre la policy 7
Algoritmo de control de Monte Carlo asumiendo exploring starts 8
Algoritmo de control de Monte Carlo con una policy ε-greedy 9
Algoritmo de controll off-policy Monte Carlo 10
Algoritmo para TD(0) 12
Algoritmo SARSA 13
Algoritmo Q-Learning 14
Algoritmo de policy gradient 16
Juego BlackJack en un casino 17
Juego de FrozenLake 18
Juego de MountainCar 18
Juego Pong 19
Juego Breakout 19
Imágen se una simulación muestreando acciones de manera aleatoria 21
Gráficos de la función de valor óptima encontrada sin As usable y con As usable 23
Policy óptima sin As usable y con As usable 24
Partidas ganadas por cada 100 episodios repetido 100 veces 25
Partidas ganadas por cada 100 episodios repetido 100 veces 26
Función de valor para MountainCar 27
Función de valor donde se observa que el máximo en la posición se alcanza en -0,5 28
Durada de los episodio en 'steps' de tiempo 28
Reward para cada episodio 29
Función de valor para MountainCar con Q-Learning 29
Reward a lo largo de los episodios 30
Arquitecura de la Deep Q-Netowrk presentada por DeepMind 31
Resultados de las distintas métricas evaluadas en el experimento de Deep Q-Learning 33
Resultados de las distintas métricas evaluadas en el experimento de Double Deep Q-Learning 34
Diagrama Dueling Deep Q-Network 35
Duración del episodio a lo largo del entrenamiento 36
Reward por episodio a lo largo del tiempo 37

1. Introducción

En los últimos años se ha popularizado en la comunidad científica y también en la industrial el concepto de “Reinforcement Learning”. Aunque este concepto no es nuevo, sí que recientemente se han logrado grandes avances. El objetivo de este proyecto es explicar los algoritmos que se utilizan en esta disciplina y probarlos en diferentes juegos con el objetivo de mostrar su aplicación e intentar conseguir resultados óptimos o suficientemente buenos utilizando toda la gama de posibles métodos incluyendo el estado del arte. Para ello se utilizará el entorno creado por OpenAI llamado Gym que permitirá testear los algoritmos desarrollados contra los diferentes juegos que se han seleccionado para este proyecto.

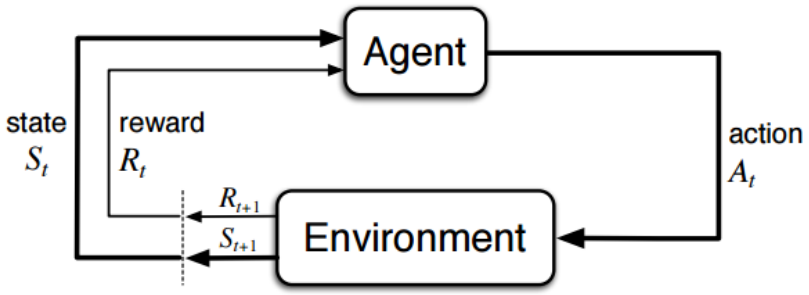
2. Reinforcement Learning

En este partado se definen los conceptos claves de esta disciplina y necesarios para la comprensión del trabajo.

Agente – entorno:

En el framework de RL, el agente es el que aprende y el que toma decisiones mientras que todo lo complementario al agente, y con lo qué éste interactua se llama entorno.

Se trata de una interacción continua en qué el agente selecciona acciones y el entorno responde presentando nuevos estados.



1. Diagrama de la interacción entre el agente y el entorno

Goals y rewards:

El reward es un señal representado por un número real que pasa del entorno al gente.

El goal del agente es maximizar el reward total que recibe, lo que significa maximizar el reward acumulado a largo plazo.

Returns:

En general, lo que se pretende, es maximizar el return esperado y éste se define como:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T ,$$

Donde T es el último paso (step).

Otro concepto a mencionar y utilizado en los algoritmes aplicados a los juegos de este proyecto es el del descuento:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum \gamma^k R_{t+k+1},$$

Propiedad de Markov:

La propiedad de Markov se refiere a la propiedad de ciertos procesos estocásticos por la cual "carecen de memoria", lo que significa que la distribución de probabilidad del valor futuro de una variable aleatoria depende únicamente de su valor presente, siendo independiente de la historia de dicha variable. A los procesos que satisfacen esta condición se les conoce como procesos de Márkov. Esta es una propiedad que se darà por supuesta en todos los juegos que se utilizan.

Markov Decision Process:

Un problema de Reinforcement Learning que satisface la propiedad de Markov se llama proceso de decisión de Markov, o MDP. Si es espacio de los estados y lass acciones es finito entonces se llama MDP finito.

Policy:

Una policy, π, consiste en un mapeo de cada estado, $s \in S$, y acción, $a \in A(s)$, a la probabilidad $\pi(a|s)$ de tomar una acción desde el estado s.

Value functions:

Muchos de los algoritmos de reinforcement learning implican la estimación de funciones de valor- funciones de estados (o de pares de estados-acciones) que estiman cuán bueno es que el agente esté en un estado dado (o lo bueno que es realizar una determinada acción en un determinado estado). La noción de "lo bueno" aquí se define en términos de recompensas futuras que se pueden esperar. Por supuesto, las recompensas que el agente puede esperar recibir en el futuro dependerán de qué acciones éste tomará. Por consiguiente, las funciones de valor se definen con respecto a policies particulares.

Optimal value functions:

Resolver una tarea de reinforcement learning significa, aproximadamente, encontrar una policy que logre una gran cantidad de reward a largo plazo. Para los MDP finitos, se puede definir con precisión una policy óptima de la siguiente manera:

Las funciones de valor definen un orden parcial sobre las policies. Una política π se define como mejor o igual que una política π_0 si su return esperado es mayor o igual que el de π_0 para todos los estados. En otras palabras, $\pi \geq \pi_0$ si y sólo si $v_\pi(s) \geq v_{\pi 0}(s)$ para todo $s \in S$. Siempre hay al menos una policy que es mejor que o igual a todas las demás policies.

Esta es una policy óptima. Aunque puede haber más de una, se denotan todas las policies óptimas por π*.

2.1. Métodos para resolver Reinforcement Learning

Se habla principalmente de tres métodos para resolver el problema de RL:

- Dynamic programing
- Monte carlo methods
- Temporal difference learning

Los métodos de dynamic programming consisten en métodos para calcular políticas óptimas con el supuesto que se tiene un modelo perfecto del entorno como proceso de decisión de Markov. En el caso de estudio del proyecto, nunca se tendrá un completo conocimiento de las dinámicas del entorno y es por ello que no se utilizarán estos métodos.

2.2. Métodos de Monte Carlo

En este caso no se asume conocimiento del entorno en el sentido que no se conocen las dinámicas que gobiernan al juego. Los métodos de Monte Carlo solo requieren de la experiencia. Se trata de muestrear secuencias de estados, acciones y puntuaciones de las interacciones con el entorno.

Aunque sí que se requiere de un modelo para poder aprender, solo se necesita que éste pueda generar transiciones, pero sin la necesidad de que el agente conozca las dinámicas del entorno (éste será el papel de gym).

Éstos métodos se basan en el promedio de las muestras de returns o premios. Para asegurarse de que los returns están ‘bien definidos’, la experiencia debe estar dividida en episodios y éstos deben terminar en algún momento independientemente de las acciones que tome el agente.

Es por tanto al final de un episodio cuando se estima el valor de los estados y se cambia la policy.

2.2.1. Monte Carlo Policy Evaluation

El objetivo es aprender la función de valor de estado para una determinada policy. El valor de un estado es el return esperado empezando desde ese estado.

Por lo tanto, una manera simple de estimar el valor con la experiencia es senzillamente hacer la media de los returns observados después de haber visitado un estado.

Every-visit MC:

Este método estima V como la media de los returns que siguen a todas las visitas a cierto estado en un set de episodios.

First-visit MC:

En este caso solo se tendrá en cuenta la primera visita a un estado para estimar el valor de dicho estado.

En este trabajo se utilizará principalmente First-visit MC.

Initialize:
$\pi \leftarrow$ policy to be evaluated
$V \leftarrow$ an arbitrary state-value function
$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:
(a) Generate an episode using π
(b) For each state s appearing in the episode:
$G \leftarrow$ return following the first occurrence of s
Append G to $Returns(s)$
$V(s) \leftarrow$ average($Returns(s)$)

2. First-visit MC para la estimación de la función de valor de una determinada policy

2.2.2. Estimación de los valores de las acciones con Monte Carlo

Si no se tiene acceso al modelo del juego, es particularmente útil estimar los valores de las acciones en lugar de los valores de los estados. Con un modelo, los valores de estado son suficientes para determinar una estrategia; Uno simplemente mira hacia adelante un paso y elige la acción que conduzca a la mejor combinación de recompensa y próximo estado.

Sin un modelo, sin embargo, los valores de estado por sí solos no son suficientes. Se debe estimar explícitamente el valor de cada acción para que los valores sean útiles en la sugerencia de una policy. Así, uno de los principales objetivos de Monte Carlo es estimar q*.

El problema de evaluación de estrategias para los valores de acción es estimar q_π (s, a), el return esperado al comenzar en el estado s, tomar la acción a, y después seguir la policy π.

La única complicación es que muchos pares de estado-acción relevantes pueden no ser nunca visitados. Si π es una estrategia determinística,

entonces al seguir π se observará una de las acciones para cada estado. Sin returns que promediar, las estimaciones de Monte Carlo de las otras acciones no mejorarán con la experiencia.

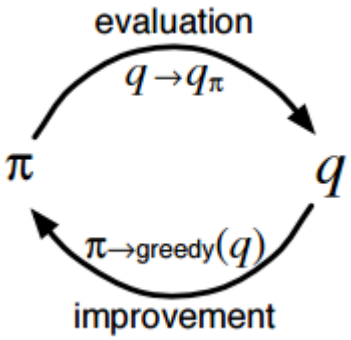
Este es un problema serio porque el propósito de aprender los valores de las acciones es ayudar a elegir entre las acciones disponibles en cada estado. Para comparar las alternativas, se necesita estimar el valor de todas las acciones de cada estado, no sólo el que actualmente se está favoreciendo.
Para que la evaluación de la estrategia funcione, se debe asegurar la exploración continua. Una manera de hacer esto es especificando que el primer step de cada episodio comience en un par estado-acción, y que cada par de éstos tenga una probabilidad no nula de ser seleccionado para empezar el episodio. Esto garantiza que todos los pares de acción-estado serán visitados de manera infinita en el límite de un número infinito de episodios. Esta asunción se llama **exploring starts**.

El enfoque alternativo más común para asegurar que todos los pares de acción-estado son visitados es considerar sólo estrategias que sean estocásticas con una probabilidad no nula de seleccionar todas las acciones.

Éste segundo enfoque serà el que se utilitzará en los experimentos.

2.2.3. Monte Carlo Control

Cuando se habla de control, se habla de aproximar policies óptimas. La idea principal para hacerlo se basa en lo siguiente:



3. Iteración sobre la policy

Se mantiene una policy aproximada y una función de valor aproximada y se va alterando de manera repetida para poder aproximar mejor la función de valor de la policy actual, y ésta se va mejorando con respecto a la función de valor actual.

Estos dos tipos de cambios se ayudan unos a otros en cierta medida, ya que cada uno crea un target dinámico para el otro, pero juntos hacen que tanto la estrategia como la función de valor se acerquen a la optimalidad.

A continuación se muestra una secuencia para ilustrar la idea:

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} q_*,$$

Donde E → denota una evaluación completa de la estrategia y I → denota una mejora de la estrategia.
La mejora de la policy se hace haciendo que ésta sea **greedy**(se elige la acción con más valor) con respecto a la función de valor actual.

$$\pi(s) = \arg\max_a q(s, a).$$

La mejora de la policy puede hacerse construyendo cada π_{k+1} como la estrategia greedy con respecto a $q_{\pi k}$.

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
  $Q(s, a) \leftarrow$ arbitrary
  $\pi(s) \leftarrow$ arbitrary
  $Returns(s, a) \leftarrow$ empty list

Repeat forever:
  (a) Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ s.t. all pairs have probability > 0
    Generate an episode starting from S_0, A_0, following π
  (b) For each pair s, a appearing in the episode:
    $G \leftarrow$ return following the first occurrence of s, a
    Append G to $Returns(s, a)$
    $Q(s, a) \leftarrow$ average$(Returns(s, a))$
  (c) For each s in the episode:
    $\pi(s) \leftarrow \arg\max_a Q(s, a)$

4. Algoritmo de control de Monte Carlo asumiendo exploring starts

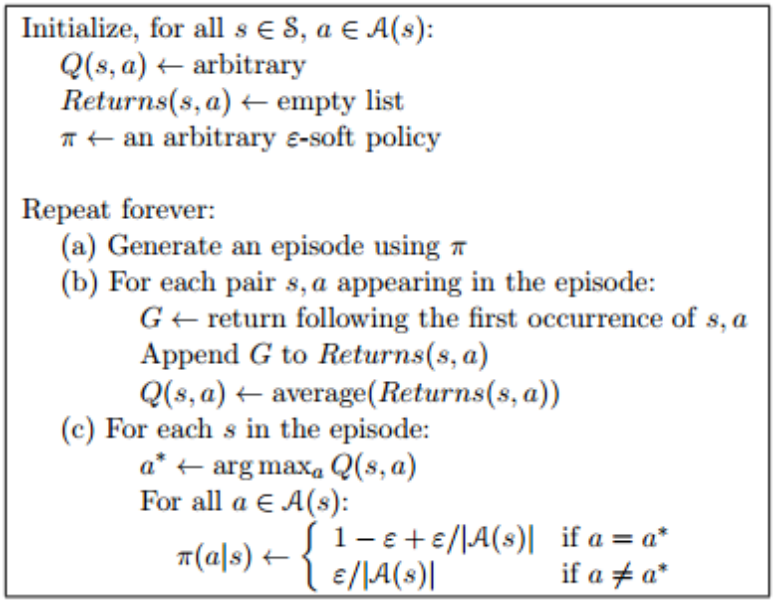
2.2.4. On-policy Monte Carlo Control

La idea es evitar la suposición improbable de los **exploring starts**. La única forma general de asegurar que todas las acciones se seleccionan infinitamente a menudo es que el agente continue seleccionándolas. Hay dos enfoques para asegurar esto, llamados métodos **on-policy** y **off-policy**. En los métodos on-policy se trata de evaluar o mejorar la policy utilizada para tomar decisiones. Los métodos off-policy se comentan en el siguiente punto.

En los métodos de control on-policy, la estrategia es generalmente soft,

siendo π(a|s)>0 para todo s ∈ S y todo a ∈ A(s). Una posibilidad es cambiar gradualmente la estrategia hacia una estrategia óptima determinista. El método que se presenta en este proyecto utiliza estrategias ε-greedy, lo que significa que la mayor parte del tiempo elige una acción que tiene un valor de acción estimado máximo, pero con probabilidad ε selecciona una acción al azar. Es decir, todas las acciones no-greedy tienen una probabilidad mínima de selección, ε/|A(s)|, y el resto de la probabilidad, 1 - ε + ε /|A (s)|, se da a la acción greedy.

En este proyecto también se utiliza una ε variable, de manera que inicialmente fomenta la exploración y a medida que va disminuyendo se acerca a una policy greedy fomentando la explotación de la función de valor.



5. Algoritmo de control de Monte Carlo con una policy ε-greedy

2.2.6. Off-policy Monte Carlo Control

A continuación se presenta la segunda clase de métodos de aprendizaje de control: off-policy. La policy utilizada para generar el comportamiento (la manera en que el agente juega), llamada la policy de comportamiento, puede de hecho no estar relacionada con la policy que se evalúa y mejora, llamada la policy de estimación. Una ventaja de esta separación es que la política de estimación puede ser determinista (por ejemplo, greedy), mientras que la policy de comportamiento puede seguir probando todas las acciones posibles.

En la siguiente figura se muestra un método Monte Carlo off-policy, para estimar q_*. La policy de comportamiento μ se mantiene como una policy

soft arbitraria (soft: todos los pares estado-acción tienen probabilidad no nula de ser seleccionados).

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$N(s,a) \leftarrow 0$; Numerator and
$D(s,a) \leftarrow 0$; Denominator of $Q(s,a)$
$\pi \leftarrow$ an arbitrary deterministic policy

Repeat forever:
(a) Select a policy μ and use it to generate an episode:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
(b) $\tau \leftarrow$ latest time at which $A_\tau \neq \pi(S_\tau)$
(c) For each pair s, a appearing in the episode at time τ or later:
$t \leftarrow$ the time of first occurrence of s, a such that $t \geq \tau$
$W \leftarrow \prod_{k=t+1}^{T-1} \frac{1}{\mu(A_k|S_k)}$
$N(s,a) \leftarrow N(s,a) + WG_t$
$D(s,a) \leftarrow D(s,a) + W$
$Q(s,a) \leftarrow \frac{N(s,a)}{D(s,a)}$
(d) For each $s \in \mathcal{S}$:
$\pi(s) \leftarrow \arg\max_a Q(s,a)$

6. Algoritmo de controll off-policy Monte Carlo

Un problema potencial es que este método sólo aprende de las colas(últimos pasos) de los episodios, después de la última acción no greedy. Si las acciones no greedy son frecuentes, entonces el aprendizaje será lento, particularmente para los estados que aparecen en las partes iniciales de episodios largos. Potencialmente, esto podría retardar mucho el aprendizaje.

Este método ha sido poco probado y se dejará fuera del objeto del proyecto pero se menciona aquí como primer método off-policy comentado.

2.3 Temporal-Difference Learning

Los métodos TD y Monte Carlo utilizan la experiencia para resolver el problema de predicción (estimar la función de valor de una policy). Dada una cierta experiencia siguiendo una política π, ambos métodos actualizan su estimación v de v_π para los estados no terminales S_t que ocurren en ese episodio. A grandes rasgos, los métodos de Monte Carlo esperan hasta que el return siguiendo a la visita de un estado es conocido, y a continuación, utilizan ese return como un target de $V(S_t)$. Como por ejemplo:

$$V(S_t) \leftarrow V(S_t) + \alpha\left[G_t - V(S_t)\right],$$

Donde G_t es el retorno real después del tiempo t, y α es un parámetro constante. Este método se llama constant-α MC. Mientras que los métodos de Monte Carlo deben esperar hasta el final del episodio para determinar el incremento a $V(S_t)$ (sólo entonces se conoce G_t), los métodos TD solo necesitan esperar hasta el siguiente paso. En el instante t+1 forman inmediatamente un target y hacen una actualización utilizando el retorno observada R_{t+1} y la estimación de $V(S_{t+1})$. El método TD más simple, conocido como TD (0), es

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big].$$

En efecto, el objetivo de la actualización de Monte Carlo es G_t, Mientras que el target de la actualización de TD es $R_{t+1} + \gamma V(S_{t+1})$.

Dado que el método TD basa su actualización en parte en una estimación existente, se dice que es un método de bootstrap. Se sabe que:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \;\middle|\; S_t = s\right] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s]. \end{aligned}$$

A grandes rasgos, los métodos de Monte Carlo utilizan una estimación de la primera ecuación como objetivo, mientras que los métodos DP utilizan una estimación de la última ecuación como objetivo.

El target de Monte Carlo es una estimación porque el valor esperado no se conoce; se utiliza una muestra del return en lugar del return esperado real.

El target de TD es una estimación por dos razones: se muestrean los valores esperados y utiliza la estimación actual V en lugar de la verdadera v_π.

Input: the policy π to be evaluated
Initialize $V(s)$ arbitrarily (e.g., $V(s) = 0, \forall s \in \mathcal{S}^+$)
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    $A \leftarrow$ action given by π for S
    Take action A; observe reward, R, and next state, S'
    $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
  until S is terminal

7. Algoritmo para TD(0)

2.3.1. Optimalidad de TD(0)

Éste método es el que se utilitzará cuando se presenten algoritmos que utilizan TD para resolver los juegos.

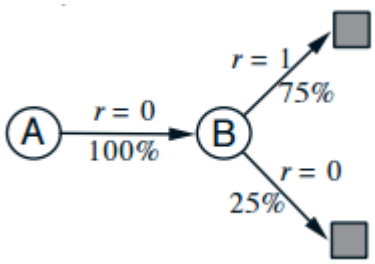
A continuación un ejemplo muy ilustrativo del libro de Sutton y Barto para tener una idea de la diferencia entre los métodos MC y TD

Ejemplo:

Se supone que se observan los siguientes ocho episodios:

A, 0, B, 0	B, 1
B, 1	B, 1
B, 1	B, 1
B, 1	B, 0

Esto significa que el primer episodio ha empezao en el estado A, transicionado a B con un reward de 0, y luego terminado en B con un reward de 0. Los otros siete episodios han sido aún más cortos, empezando por B y terminando inmediatamente. Dado este batch de datos, ¿cuales se dirían que son las predicciones óptimas, los mejores valores para las estimaciones V(A) y V(B)? Todo el mundo probablemente estaría de acuerdo en que el valor óptimo para V(B) es 3/4, ya que seis de las ocho veces en el estado B el proceso ha terminado inmediatamente con un return de 1, y las otros dos el proceso ha terminado inmediatamente con un retorno de 0.
Pero ¿cuál es el valor óptimo para la estimación V(A) dada esta información? Aquí hay dos respuestas razonables. Uno es observar que el 100% de las veces el proceso estaba en el estado A y que ha pasado inmediatamente a B (con una recompensa de 0);y ya que ya se ha decidido que B tiene valor 3/4, A debe tenir valor 3/4 también. Una forma de ver esta respuesta es que ésta se basa primero en modelar el proceso de Markov, en este caso como



Y luego calcular las estimaciones correctas dado el modelo, que de hecho en este caso da V(A) = 3/4. Esta es también la respuesta que el batch TD (0) da.
La otra respuesta razonable es simplemente observar que se ha visto A una vez y el return que ha seguido ha sido 0; por lo tanto se estima V(A) como 0. Esta es la respuesta que los métodos de batch Monte Carlo dan.

2.3.2. On-policy TD Control (SARSA)

El primer paso es aprender una función de valor de acción en lugar de una función de valor de estado. En particular, para un método on-policy se debe estimar $q_\pi(s,a)$ para la política de comportamiento actual π y para todos los estados s y acciones a.

Se consideran las transiciones de un par estado-acción a otro par, y se aprende el valor de éstos.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big].$$

Esta actualización se realiza después de cada transición de un estado no terminal S_t . Si S_{t+1} es terminal, entonces $Q(S_{t+1}, A_{t+1})$ se define como cero. Esta regla utiliza cada elemento del quintuple de los acontecimientos, (S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}), que constituyen una transición de un par estado-acción al siguiente. Este quintuple da lugar al nombre Sarsa para el algoritmo.

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state,\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

8. Algoritmo SARSA

2.3.3. Off-policy TD Control (Q-Learning)

Uno de los avances más importantes en reinforcement learning fue el desarrollo de un algoritmo de control TD off-policy conocido como Q-learning. En su forma más simple se define como:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big].$$

En este caso, la función de valor de acción aprendida, Q, se aproxima directamente a q_*, la función de valor de acción óptima, independientemente de la policy seguida. La policy todavía tiene un efecto en el sentido que determina qué pares de acción-estado son visitados y actualizados. Sin embargo, todo lo que se requiere para la convergencia correcta es que todos los pares sigan siendo actualizados.

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
    $S \leftarrow S'$;
  until S is terminal

9. Algoritmo Q-Learning

2.4. Métodos de aproximación de funciones

Afortunadamente, la generalización a partir de ejemplos ya ha sido ampliamente estudiada, y no es necesario inventar métodos totalmente nuevos para su uso en el aprendizaje de refuerzo. En gran medida solo necesitamos combinar métodos de aprendizaje de refuerzo con los métodos de generalización existentes. El tipo de generalización que requerimos a menudo se llama aproximación de función porque toma ejemplos de una función deseada (por ejemplo, una función de valor) y trata de generalizar a partir de ellos para construir una aproximación de toda la función. La aproximación de funciones es una instancia de aprendizaje supervisado, el tema principal estudiado en el aprendizaje automático, las redes neuronales artificiales, el reconocimiento de patrones y la adaptación de curvas estadísticas.

Como método de aproximación para encontrar los parámetros de la función se utilizará Gradient-Descent. No es objeto de este proyecto definir como funciona este método de aprendizaje ya que se da por supuesto que el lector ya ha tenido contacto con el machine learning previamente.

2.4.1. Métodos Lineales

Se explica este partado para familiarizar al lector con uno de los métodos utilizados para uno de los experimentos (MountainCar). Uno de los casos típicos de aproximación de funciones es aquel en que la relación entre los parámetros y las características (estados, acciones, etc) es lineal.

En este apartado se introduce de manera breve uno de los métodos para construir características sobre los juegos para poder aplicar métodos lineales de aproximación de funciones.
Los cuatro métodos más utilizados son:
- Coarse coding
- Tile coding
- Radial basis functions
- Kanerva coding

Solo se explicarán las Radial Basis Functions ya que han sido utilizadas para construir caractarísticas para el juego de MountainCar.

Radial basis functions:

La característica extraida del estado puede tenir cualquier valor en el intervalo [0, 1], reflejando varios grados en los que la característica está presente. Por ejemplo, una característica RBF típica puede tener una respuesta gaussiana $x_i(s)$ dependiente solamente de la distancia entre el estado, s, y el estado prototípico de la característica o estado central, c_i, y relativo al ancho de la característica, σ_i:

$$x_i(s) = \exp\left(-\frac{\|s - c_i\|^2}{2\sigma_i^2}\right).$$

De esta manera un punto en el espacio puede ser transformado a un conjunto de características n, en función del número de RBF que se utilicen, y cada una de estas RBF indica quan presente está la característica. Por ejemplo cuando la distancia entre el estado y el estado central vale 0, la presencia de ésta será máxima con valor 1.

2.4.2. Métodos no Lineales

En este proyecto se han usado deep neural networks para poder atacar los juegos más complejos. Queda fuera del abaste de este proyecto explicar el funcionamiento del deep learning o de las shallow networks.

2.4.3. Aproximación de la policy

La policy se representa directamente, con sus propios pesos independientes de cualquier función de valor.

Los métodos de policy gradient son un tipo de técnicas de reinforcement learning que se basan en la optimización de las policies parametrizadas con respecto al rendimiento esperado (recompensa acumulativa a largo plazo) utilizando gradient descent. No sufren muchos de los problemas que han estado teniendo los enfoques tradicionales de reinforcement learning tales como la falta de garantías de una función de valor, el problema de la intratabilidad que resulta de la incierta información de los estados y la complejidad que surge de los espacios de estados y acciones continuos.

function REINFORCE
Initialise θ arbitrarily
for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
for $t = 1$ to $T - 1$ **do**
$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
end for
end for
return θ
end function

10. Algoritmo de policy gradient

3. Juegos utilizados:

3.1. BlackJack

Se trata de un juego de cartas, propio de los casinos con una o más barajas inglesas de 52 cartas sin los comodines, que consiste en sumar un valor lo más próximo a 21 pero sin pasarse. En un casino cada jugador de la mesa juega únicamente contra el crupier, intentando conseguir una mejor jugada que este. El crupier está sujeto a reglas fijas que le impiden tomar decisiones sobre el juego. Por ejemplo, está obligado a pedir carta siempre que su puntuación sume 16 o menos, y obligado a plantarse si suma 17 o más. Las cartas numéricas suman su valor, las figuras suman 10 y el As vale 11 o 1, a elección del jugador. En el caso del crupier, los Ases valen 11 mientras no se pase de 21, y 1 en caso contrario. La mejor jugada es conseguir 21 con solo dos cartas, esto es con un As más carta de valor 10. Esta jugada se conoce como Blackjack o 21 natural. Un Blackjack gana sobre un 21 conseguido con más de dos cartas.

11. Juego BlackJack en un casino

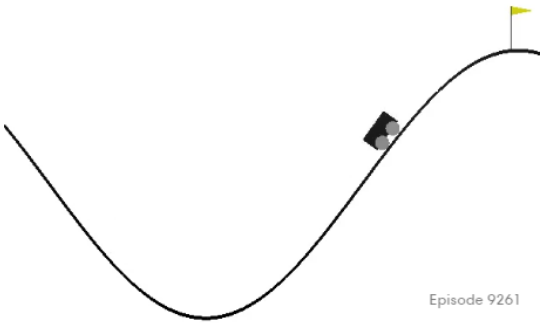
3.2. FrozenLake

El invierno está aquí. Tú y tus amigos estabáis tirando un frisbee en el parque cuando de golpe has hecho un lanzamiento muy fuerte que ha hecho que el frisbee aterrice en medio del lago. La mayor parte de la superficie está congelada, pero hay algunos agujeros donde el hielo se ha derretido. Si entras en uno de esos agujeros, caerás en el agua helada. En estos momentos, hay una escasez de frisbees en el mundo, por lo que es absolutamente necesario andar a través del lago para recuperar el disco. Sin embargo, el hielo es resbaladizo, por lo que no siempre vas a moverte en la dirección que pretendes.

12. Juego de FrozenLake

3.3. MountainCar

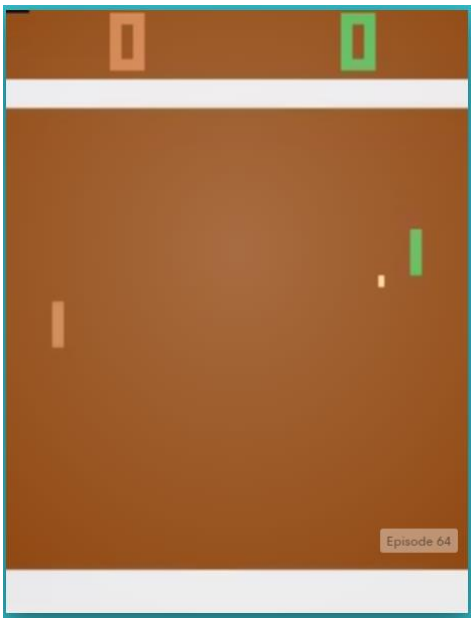
El objetivo es subir la montaña a la derecha; Sin embargo, el motor del coche no es lo suficientemente potente para subit la montaña en un solo paso. Por lo tanto, la única manera de tener éxito es conducir hacia adelante y hacia atrás para acumular impulso.



13. Juego de MountainCar

3.4. Pong

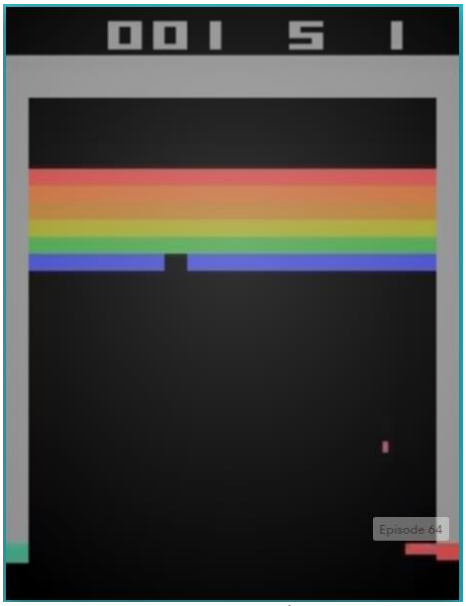
Se trata de un juego en dos dimensiones que simula un tenis de mesa. El jugador controla en el juego una paleta moviéndola verticalmente en la parte izquierda de la pantalla, y puede competir tanto contra un oponente controlado por computadora, como con otro jugador humano que controla una segunda paleta en la parte opuesta. Los jugadores pueden usar las paletas para pegarle a la pelota hacia un lado u otro. El objetivo consiste en que uno de los jugadores consiga más puntos que el oponente al finalizar el juego llegando a 21 puntos. Estos puntos se obtienen cuando el jugador adversario falla al devolver la pelota.



14. Juego Pong

3.5. Breakout

En la parte inferior de la pantalla una linea simula una raqueta que el jugador puede desplazar de izquierda a derecha. En la parte superior se situa una banda conformada por rectángulos que simulan ser ladrillos. Una pelota desciende de la nada y el agente debe golpearla con la raqueta, entonces la pelota asciende hasta pegar en el muro y los ladrillos tocados por la pelota desaparecen. La pelota vuelve a descender y así sucesivamente hasta gastar las 5 vidas. El objetivo del juego es terminar con la pared de ladrillos.



15. Juego Breakout

4. Experimentos

4.1. On-policy Monte Carlo Control para BlackJack

En este primer experimento se propone calcular la función de valor del par estado-acción utilizando el método de control de Monte Carlo con epsilon-greedy.

El modulo de gym proporciona este entorno con nombre ‘Blackjack-v0’.

El entorno retorna cuatro valores:

- Suma del jugador
- La carta visible del dealer
- Booleano indicando si el As puede ser usado como 11.
- Reward: este vale 0 para todos los movimientos dentro de la partida y pasa a valer 1, 0 o -1 en función de si gana el jugador, empata o pierde.

A continuación se muestran 5 ejemplos jugando de manera aleatoria para entender como funciona este entorno. A continuación se comentan dos de ellos:

```python
#score, dealer score, usable ace
#stick -> 0

for i in range(5):
    first_step= env.reset()
    print(first_step)
    for i in range(100):
        action = env.action_space.sample()
        print('Action taken: {}'.format(action))
        observation, reward, done, _  = env.step(action)
        print(observation)
        if done:
            print('Episode end with reward: {}'.format(reward))
            print('--------------------------------')
            break
```

```
(6, 5, False)
Action taken: 0
(6, 5, False)
Episode end with reward: -1.0
--------------------------------
(7, 8, False)
Action taken: 0
(7, 8, False)
Episode end with reward: -1.0
--------------------------------
(18, 2, True)
Action taken: 1
(18, 2, False)
Action taken: 1
(20, 2, False)
Action taken: 0
(20, 2, False)
Episode end with reward: 1.0
--------------------------------
(18, 10, False)
Action taken: 0
(18, 10, False)
Episode end with reward: 1.0
--------------------------------
(11, 8, False)
Action taken: 0
(11, 8, False)
Episode end with reward: -1.0
--------------------------------
```

16. Imágen se una simulación muestreando acciones de manera aleatoria

Primera simulación:

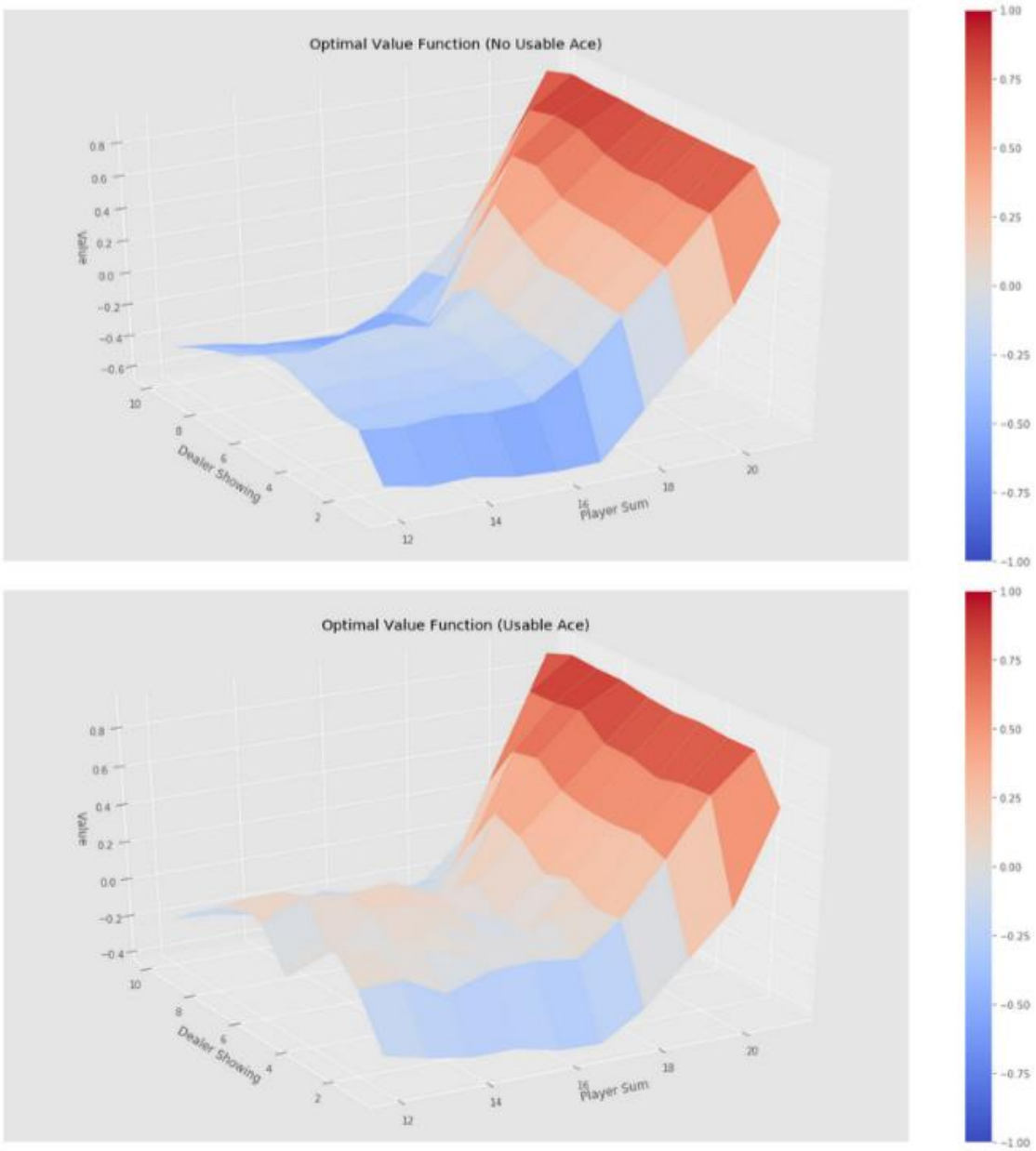
El jugador recibe dos cartas que suman 6 puntos de las cuales ninguna es un As y el dealer tiene una de las dos cartas con valor 5.

El jugador decide pasar y pasa a ser el turno del dealer que mediante la otra carta que tiene (la cual no observamos) o esta otra carta más seguir cogiendo cartas hasta llegar a un mínimo de 17 puntos gana la partida.

Tercera simulación:

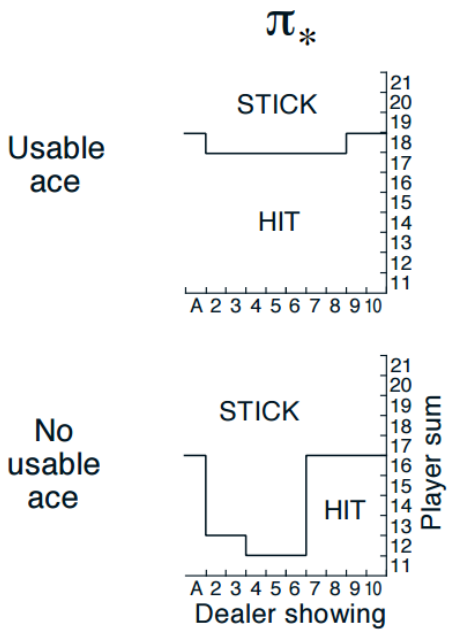
El jugador recibe dos cartas que suman 18 puntos de las cuales una es un As y por lo tanto la otra es un 7 y el dealer tiene una de las dos cartas con valor igual a 2.
El jugador decide coger otra carta y la suma total sigue valiendo 18 (de aquí sabemos que esta carta vale 10 ya que el As pasa a ser no usable).
El jugador decide coger otra carta que resulta ser un dos y decide parar con un total de 20 puntos. A continuación el dealer sigue jugando y como vemos que el jugador ha acabado pasando, podemos saber que el dealer se ha pasado de 21.

A continuación se muestra el resultado de la función de valor del par estado-acción:



17. Gráficos de la función de valor óptima encontrada sin As usable y con As usable

Se ha separado el caso en que se tiene un As usable y en el que no para poder observer la función de valor con todas sus dimensiones (3). Con esta función de valor se obtiene la siguiente estrategia óptima:

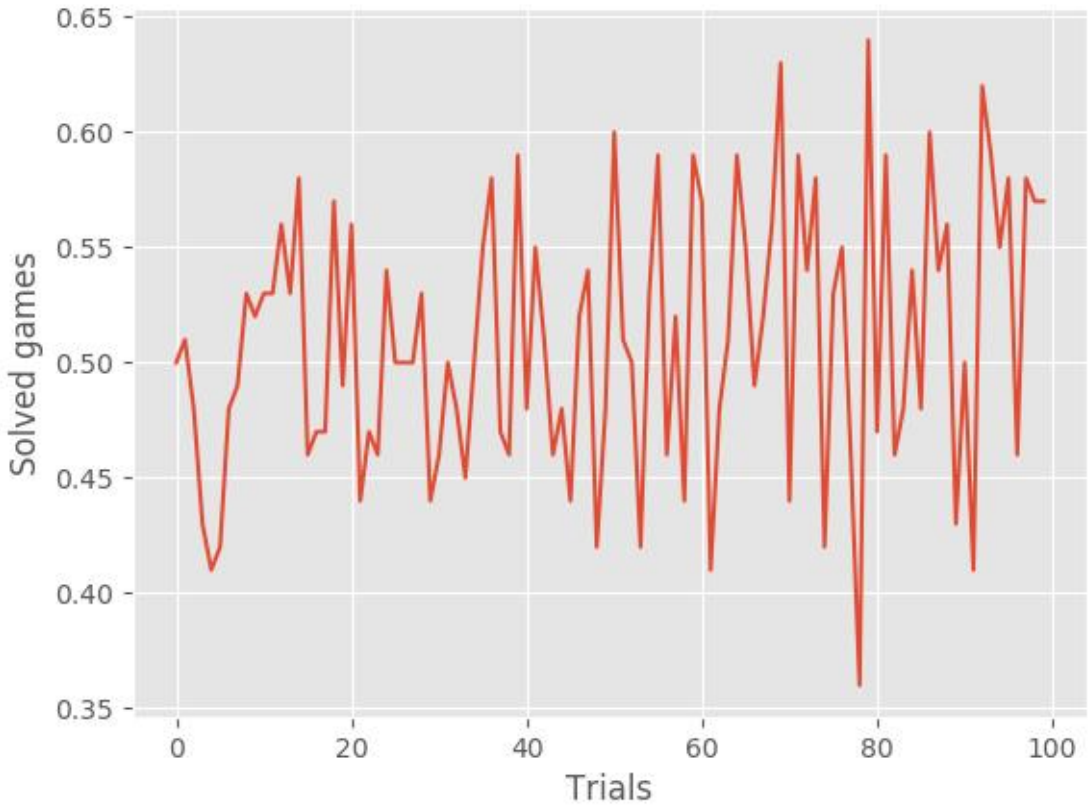


18. Policy óptima sin As usable y con As usable

Se puede observar pues cual debería ser la acción a tomar para cada uno de los 200 posibles estados. Como curiosidad se observa que el echo de tener un As usable da más margen para coger cartas debido a la dualidad de valor del As en esta situación.

4.2. FrozenLake (Monte Carlo on-policy)

Como el número de estados de este juego es finito, se va a utilizar la versión tabular, es decir, se tiene un valor del par estado-acción para cada una de las combinaciones.



19. Partidas ganadas por cada 100 episodios repetido 100 veces

Como se obserba en el gráfico, la media de juegos resueltos por cada 100 juegos jugados es de 0,53.

Los métodos de Monte Carlo requieren llegar al final de la partida para poder actualizar los valores de la tabla de valor del par estado-acción, eso hace que se requiera de mucho más tiempo que los métodos off-policy y por tanto de más experiencia.

Parámetros utilizados:

num_episodes = 2.000.000
epsilon = 0.1

Al ejecutar el algortimo proporcionado se peude observar la tabla de Q obtenida.

Comentarios:

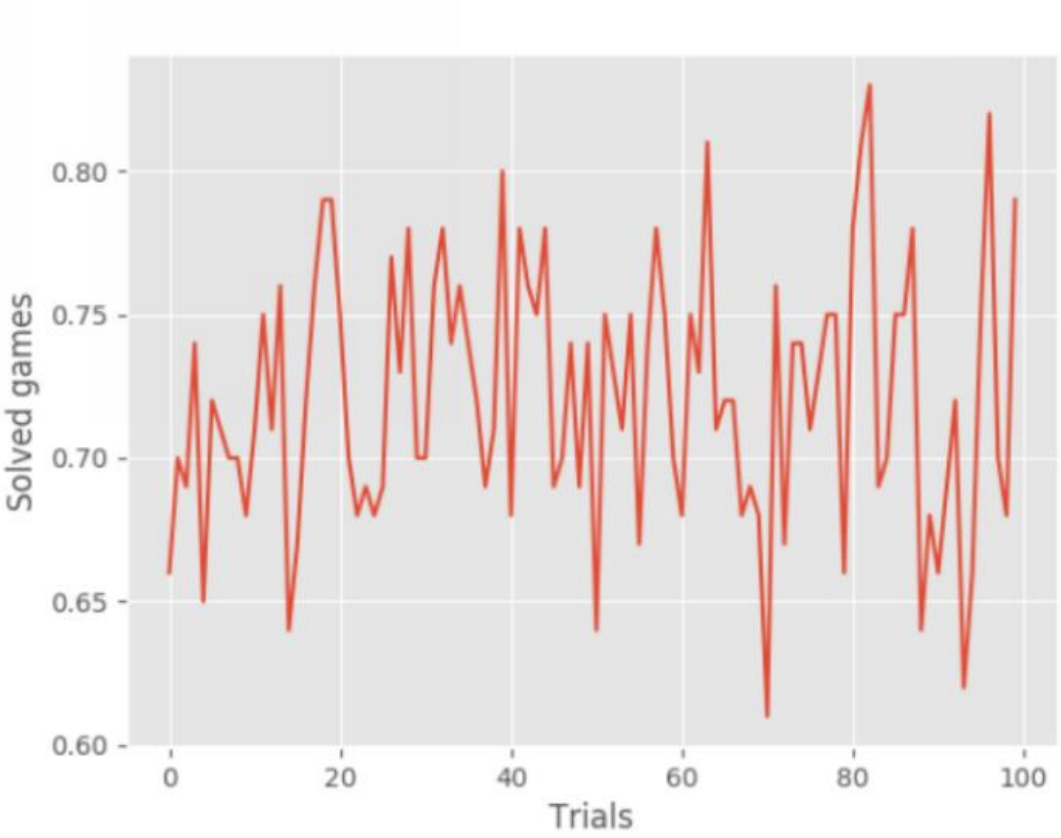
En este experimento se mantiene epsilon constante, es decir, el ratio entre exploración y explotación se mantiene igual durante todo el experimento. Una clara mejora sobre este experimento sería empezar con una epsilon mayor promoviendo así la exploración al principio e ir disminuyendo el parámetro de manera lineal con tal de que a medida

que el agente ha experimentado más episodios pueda explotar esa información.

4.3. FrozenLake: Q-Learning

Se trata otra vez de un método tabular debido al número de estados posibles (fácilmente representable en una tabla).

Parámetros:
learning_rate = 0.85
gamma = 0.99
num_episodes = 2000



20. Partidas ganadas por cada 100 episodios repetido 100 veces

El objetivo de este experimento es presentar una primera forma del algoritmo Q-learning y compararlo con la versión más 'naive' de Monte Carlo on-policy.

Se observa que no solo se ha requerido de un 0.1% de la experiencia utilizada en la versión naive de MC sinó que la media de episodios resueltos es claramente superior.

El hecho de que los algoritmos de temporal-difference no requieren de toda la experiencia de un episodio para poder actualizar la función de valor hace que la experiencia requerida sea mucho menor.

Para este juego sencillo es clara la preferencia por los algoritmos TD, pero no siempre será tan claro debido a qué se está tratando con estimaciones biased del valor de los pares de estado-acción y cuando se utilicen métodos de aproximación de funciones esto puede comportar una difícil convergencia hacia una solución.

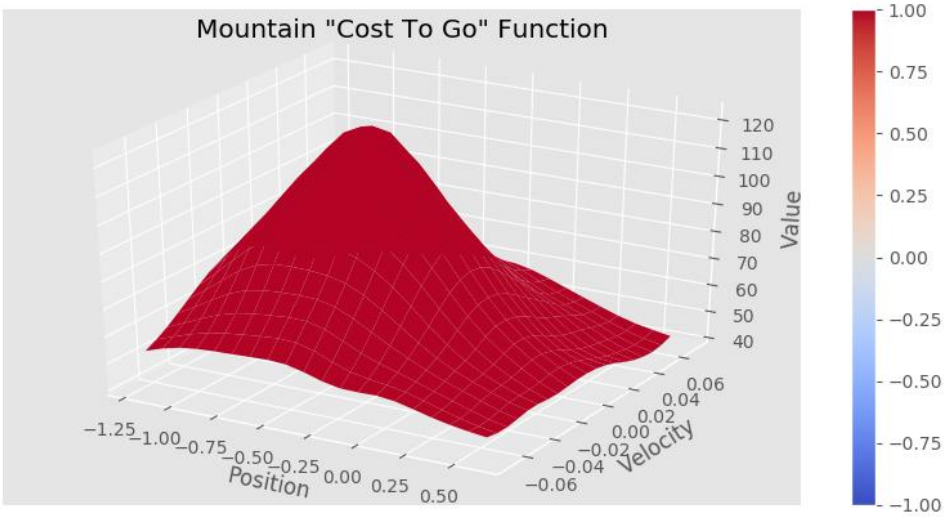
Al ejecutar el algortimo proporcionado se peude observar la tabla de Q obtenida.

4.4. Function approximation: Mountain Car - Sarsa

En este experimento se utilizará el método on-policy SARSA. Las dos variables que definen el estado del juego son la posición y la velocidad. Las posibles acciones que el agente puede tomar son ir hacia la izquierda, ir hacia la derecha o no actuar.

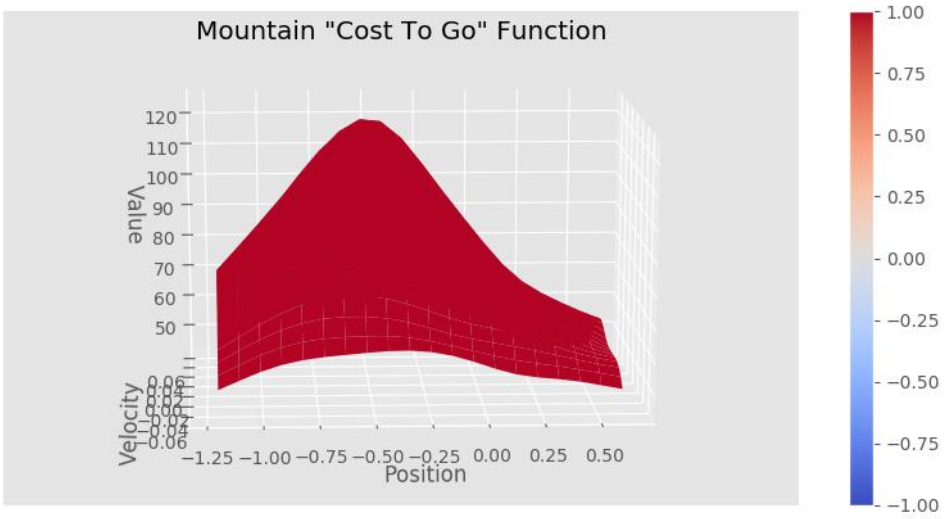
El máximo en la función de valor encontrada se encuentra en -0.5 de posición y 0.0 de velocidad.

A continuación, se muestra el gráfico de la función de valor de estado:

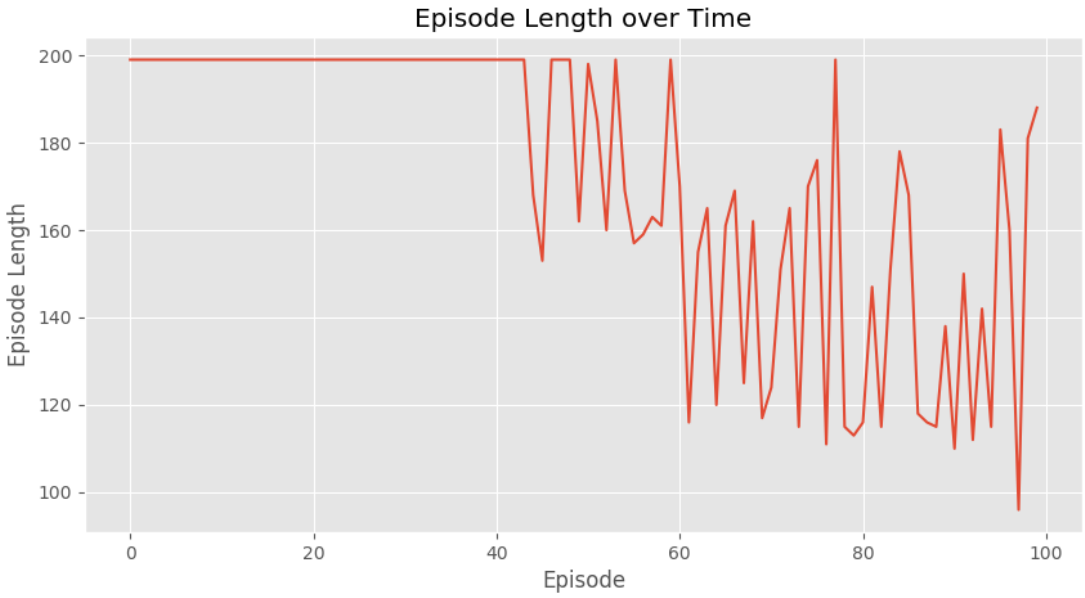


21. Función de valor para MountainCar

En el siguiente gráfico se puede observar que partiendo de la posición 0.0, el agente deberá ir hacia la posición de la izquierda que tiene un valor más elevado. Este hecho es el que hará que el agente pueda coger el impulso sificiente para poder llegar a la meta. Se observa, que efectivamente la posición -0.5 es la que tiene un valor más elevado.



22. Función de valor donde se observa que el máximo en la posición se alcanza en -0,5

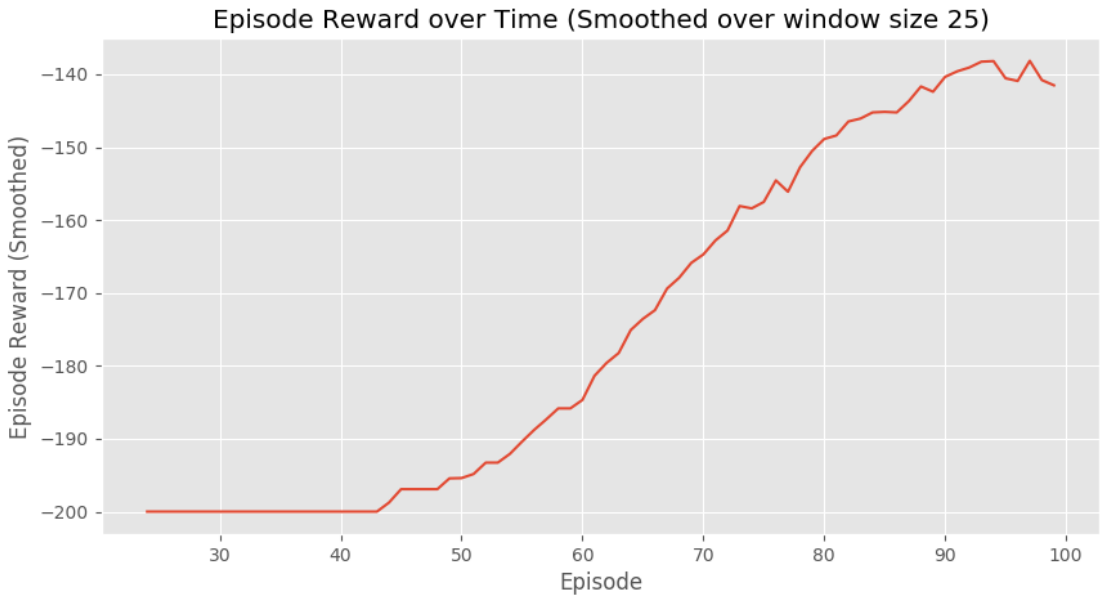


23. Durada de los episodio en 'steps' de tiempo

En este gráfico se puede observar la evolución de la duración de los episodios. Es normal que ésta vaya disminuyendo ya que con el comportamiento óptimo se debería llegar a la meta y no quedarse

oscilando en el valle y por tanto a medida que se mejora la policy, la durada del episodio será inferior.

24. Reward para cada episodio

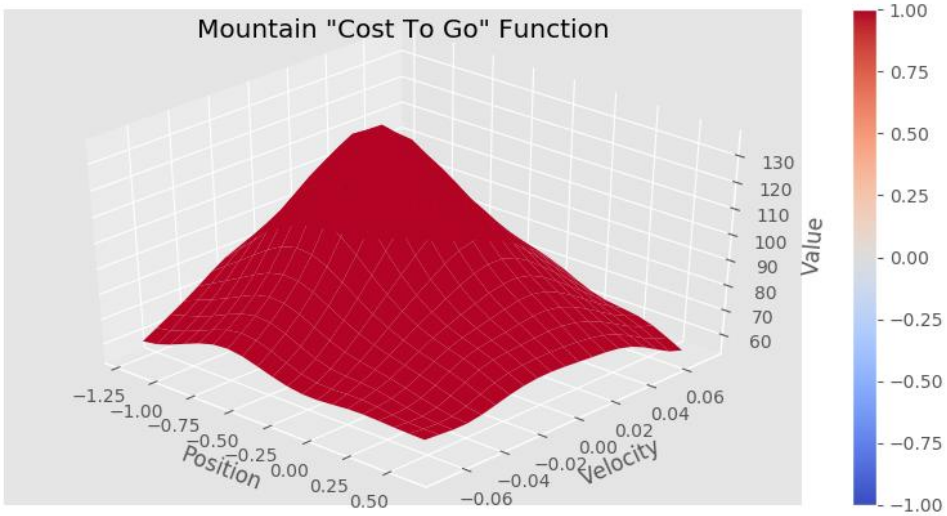


De manera similar, el ‘reward’ recibido aumentará con el tiempo.

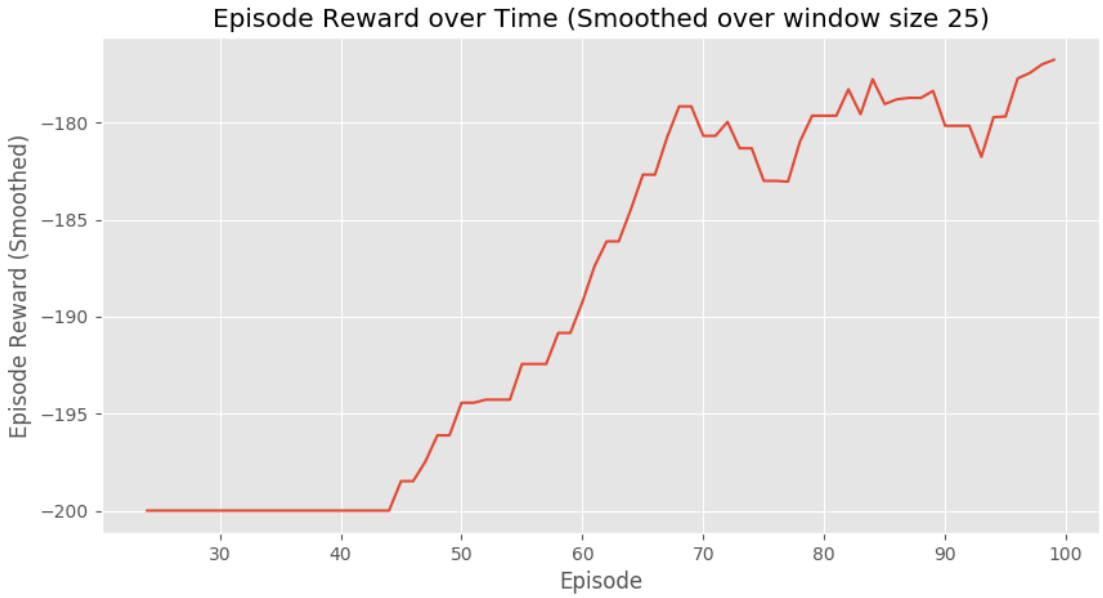
4.5. Q-Learning:

Se resolverá el mismo juego comentado anteriormente, pero utilizando esta vez Q-Learning.

25. Función de valor para MountainCar con Q-Learning



Cabe destacar que las funciones de valor encontradas por ambos métodos son muy similares, como se puede comprobar en el gráfico, y con más tiempo de entrenamiento deberían converger a la misma función de valor óptima.



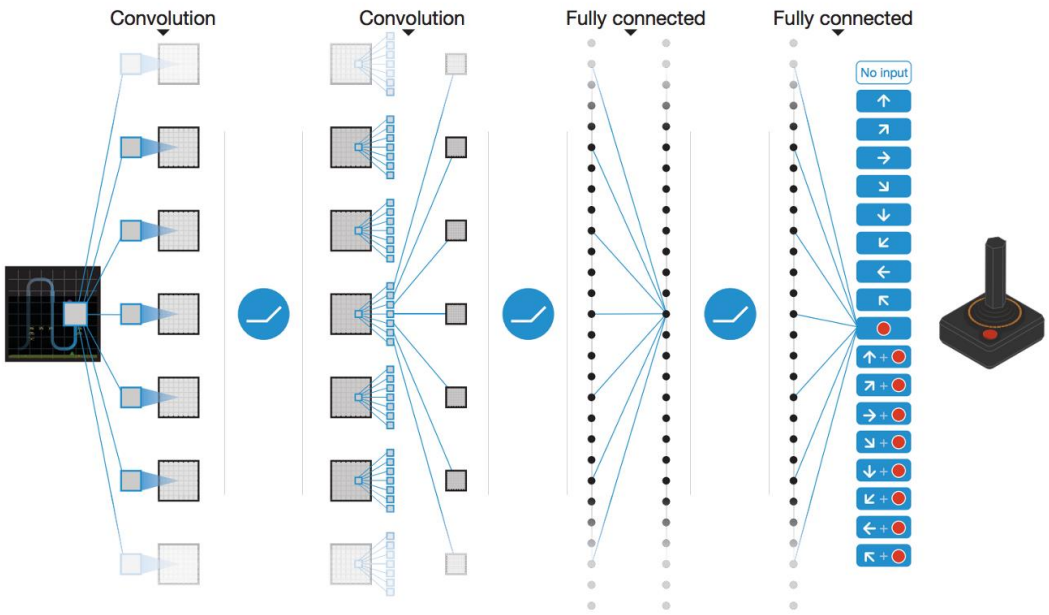
26. Reward a lo largo de los episodios

Esta gráfica es de importancia clave para ilustrar la diferencia entre SARSA y Q-Learning, ya que se puede observar que el reward obtenido con el tiempo, aunque va aumentando, no llega a los niveles que SARSA consigue. Ésto se debe a la diferencia entre los dos algoritmos y en la manera que cada uno tiene de hacer el 'update' de la función de valor.

Como Q-Learning aprende de los valores máximos, se puede decir que encuentra la manera más óptima para resolver el problema. Pero como la policy es epsilon-greedy, esto hace que muchas veces no consiga seguir el patrón de posición y velocidad ideal. Por otro lado, SARSA aprende de los valores de la propia policy y es por ello que se puede decir que coge el patrón "más robusto" y conservador que no se verá tan afectado por la policy epsilon-greedy.

4.6. Deep Q learning

Se pretende aprender a jugar al juego Breakout de Atari mediante el algoritmo presentado en el artículo de DeepMind llamado “Human level control through deep reinforcement learning”.



27. Arquitecura de la Deep Q-Netowrk presentada por DeepMind

La idea es utilizer Deep convolutional neural networks para entrenar un agente de principio a fin a partir de datos con una alta dimensionalidad: imágenes.

Haciendo que el agente pueda aprender directamente de las imágenes y de la puntuación del emulador de Atari, se puede decir que el agente está más cerca de la ‘verdadera IA’ ya que utilizará, tal como hacen los humanos, directamente las imágenes como input.

El objetivo del artículo es encontrar un agente que pueda ser aplicado a todos los juegos de Atari 2600, de tal manera que esta misma arquitectura presentada sería utilizada de igual manera para resolver todos los juegos, así como los mismos hiperparámetros.

Las convolutional neural networks utilizan layers de manera jerárquica de filtros de convolución para similar los efectos de los campos receptivos, explotando así las correlaciones espaciales presentes en las imágenes y construyendo robusteza a transformaciones naturales tales como los puntos de vista o la escala.

La idea es utilizar este tipo de red para aproximar la función de valor del par estado-acción en Breakut. El principio es el mismo comentado en el

algoritmo de Q-learning básico, solo que ahora se utilizará una red para aproximar la función.

En el artículo se presentan varios 'hacks' para supercar ciertos problemas al entrenar esta red en este escenario:

Experience Replay:

Una gran adición para hacer que las DQN funcionen es el Experience Replay. La idea básica es que almacenando las experiencias de un agente, y luego muestreando aleatoriamente batches de ellas para entrenar la red, se puede aprender de manera más robusta a realizar bien la tarea. Al mantener al azar las experiencias, se evita que la red sólo aprenda acerca de lo que está haciendo en ese momento dentro del entorno y le permita aprender de una muestra más variada de experiencias pasadas. Cada una de estas experiencias se almacenan como una tupla de <estado, acción, recompensa, estado siguiente>. El búfer de Experience Replay almacena un número fijo de steps recientes, y tal y como nuevos steps entran, los antiguos se eliminan. Cuando llega el momento de entrenar, simplemente se muestrea un batch de manera uniforme de steps aleatorias del buffer, y se entrena la red con ellos. Para la DQN, se va a construir una clase simple que maneja el almacenamiento y recuperación de steps pasados.

Separar la red de destino:

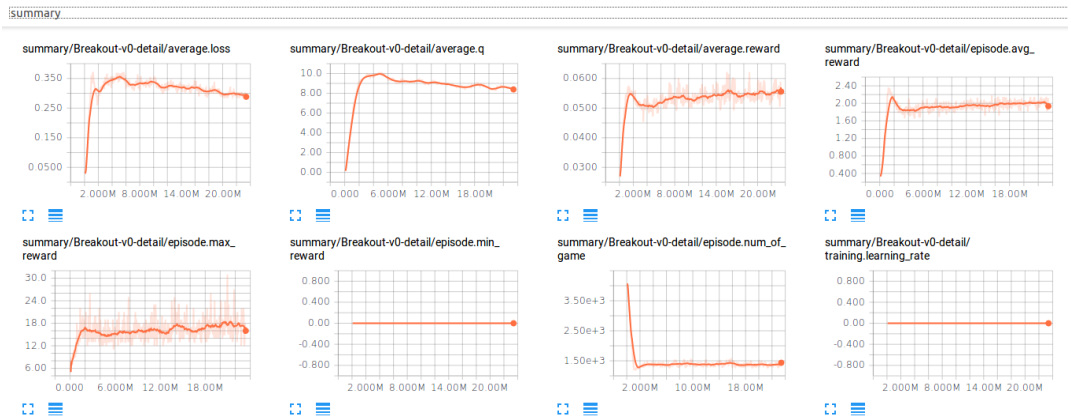
La segunda mayor adición a la DQN que la hace única es la utilización de una segunda red durante el procedimiento de entrenamiento. Esta segunda red se utiliza para generar los valores Q objetivo (target) que se utilizarán para calcular la pérdida (loss) para cada acción durante el entrenamiento. ¿Por qué no usar sólo una red para ambas estimaciones? El problema es que en cada paso del entrenamiento, los valores de la Q-network cambian, y si estamos usando un conjunto de valores constantemente cambiantes para ajustar nuestros valores de red, entonces las estimaciones de valor pueden fácilmente salirse de control. La red puede desestabilizarse al caer en bucles de realimentación entre el objetivo y los valores Q estimados. Para mitigar ese riesgo, los pesos de la red de destino son fijos y solo se actualizan periódicamente o lentamente a los valores de la Q-network primaria. De esta manera el entrenamiento puede proceder de una manera más estable.

En lugar de actualizar periódicamente la red de destino y de una sola vez, se actualizará con frecuencia, pero lentamente. Esta técnica fue introducida en otro documento de DeepMind a principios de este año, donde se descubrió que estabilizó el proceso de entrenamiento.

Cabe destacar la diferencia entre este experimento con Breakout y el siguiente juego con Pong:

En Breakout, cada vez que se pierde una vida, el juego continúa hasta las 5 vidas. Es decir, cada vez que se empieza un episodio dentro de la misma partida, el estado es distinto ya que se han ido acumulando los logros de las partidas anteriores (bloques destruidos ya que la pelota los ha alcanzado). En cambio, en el juego de Pong, aunque una partida no se acaba hasta llegar a los 21 puntos, el estado en que empieza el episodio es similar todas las veces ya que el entorno no acumula ninguna interacción anterior.

A continuación, se muestran los resultados del experimento:



28. Resultados de las distintas métricas evaluadas en el experimento de Deep Q-Learning

En el gráfico se pueden ver los resultados de las métricas definidas para este experimento:

Average loss:
Pérdida media de toda la partida. Se define como la diferencia entre nuestro estimador y el valor del estado teniendo en cuenta el reward que se recibe.

Average q:
Valor medio de la función de valor para todos los estados.

Average reward:
Reward medio para cada uno de los episodios.

Episode average reward:
Reward medio de toda la partida, es decir, de las 5 vidas.

Episode max. Reward:
Reward máximo conseguido durante los 5 episodios.

Episode min. Reward:
Reward mínimo conseguido durante los 5 episodios.

Training learning rate:
Variaición del learning rate a lo largo del entrenamiento.

Se puede encontrar todo el código adjunto en el proyecto.

4.7. Double Deep Q-Network

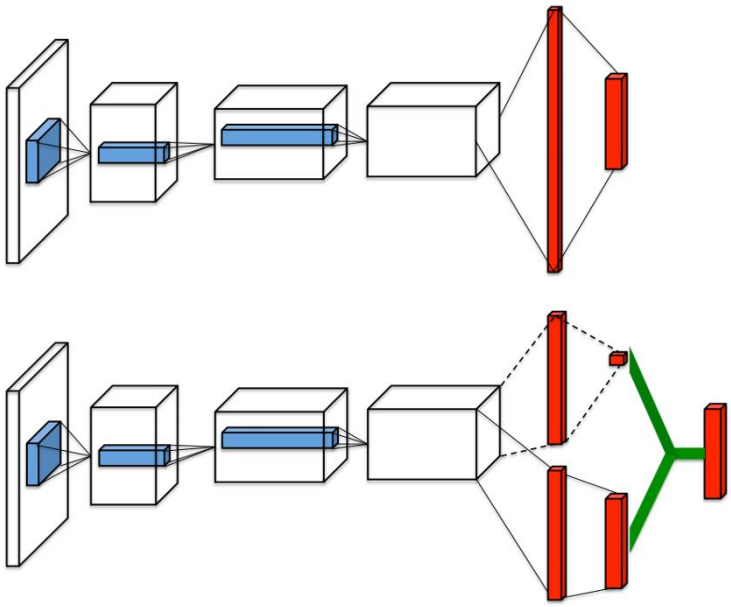
La principal intuición detrás de Double DQN es que la DQN estándar a menudo sobrestima los valores Q de las acciones potenciales a tomar en un estado dado. Mientras que esto estaría bien si todas las acciones fueran siempre sobreestimadas por igual, había razones para creer que no era el caso. Se puede imaginar fácilmente que si ciertas acciones subóptimas recibieran regularmente valores Q más altos que las acciones óptimas, el agente tendría dificultades para aprender la policy ideal. Para corregir esto, los autores de DDQN proponen un truco simple: en lugar de tomar el máximo sobre los valores de Q al calcular el valor de Q objetivo (target) para el paso de entrenamiento, se utiliza la network primaria para elegir una acción y la network Objetivo para generar el valor Q objetivo para esa acción. A continuación se muestra la nueva ecuación DDQN para actualizar el valor objetivo.

$$\text{Q-Target} = r + \gamma Q(s', \text{argmax}(Q(s', a, \Theta), \Theta'))$$

29. Resultados de las distintas métricas evaluadas en el experimento de Double Deep Q-Learning

Debido al coste en tiempo para ejecutar los algoritmos de DQN, a continuación se muestra una gráfica con las métricas comentadas para el anterior experimento pero no con el suficiente tiempo de entrenamiento. De esta manera, no ha sido posible comprobrar la mejora en performance comentada en el párrafo anterior. De todas maneras en el código se encuentra implementado y se puede ejecutar para probar que debería converger hacia un mínimo local de manera más rápida que la versión estándat de DQN.

4.8. Dueling Deep Q-Network



30. Diagrama Dueling Deep Q-Network

Con el fin de explicar el razonamiento detrás de los cambios de arquitectura que Dueling DQN hace, primero se tienen que explicar algunos términos de reinforcement learning adicionales. El par de estado-acción puede descomponerse en dos nociones más fundamentales de valor. La primera es la función de valor V (s), que dice simplemente cuan bueno es estar en cualquier estado. La segunda es la función de ventaja A(a), que indica cuánto mejor sería tomar una determinada acción comparado con las demás. Se puede entonces pensar en Q como la combinación de V y A. Más formalmente:

$$Q(s,a) = V(s) + A(a)$$

El objetivo de Dueling DQN es tener una red que calcula por separado las funciones de ventaja y valor, y las combina de nuevo en una sola función Q sólo en la capa final. Puede parecer algo inútil hacer esto a primera vista. ¿Por qué descomponer una función que se acaba de recomponer? La clave para darse cuenta del beneficio es apreciar que el agente puede no tener que preocuparse por el valor y la ventaja en un momento dado. Se pueden lograr estimaciones más sólidas del valor de estado al disociarlo de la necesidad de estar unido a acciones específicas. Es decir, de esta manera se està reduciendo la varianza de el estimador.

Aunque se encuentra implementado en el código, ha habido problemas de convergencia que no han podido ser resueltos. Se continua trabajando en encotrar la causa de error.

4.9. Policy Gradient (Pong)

Parámetros:

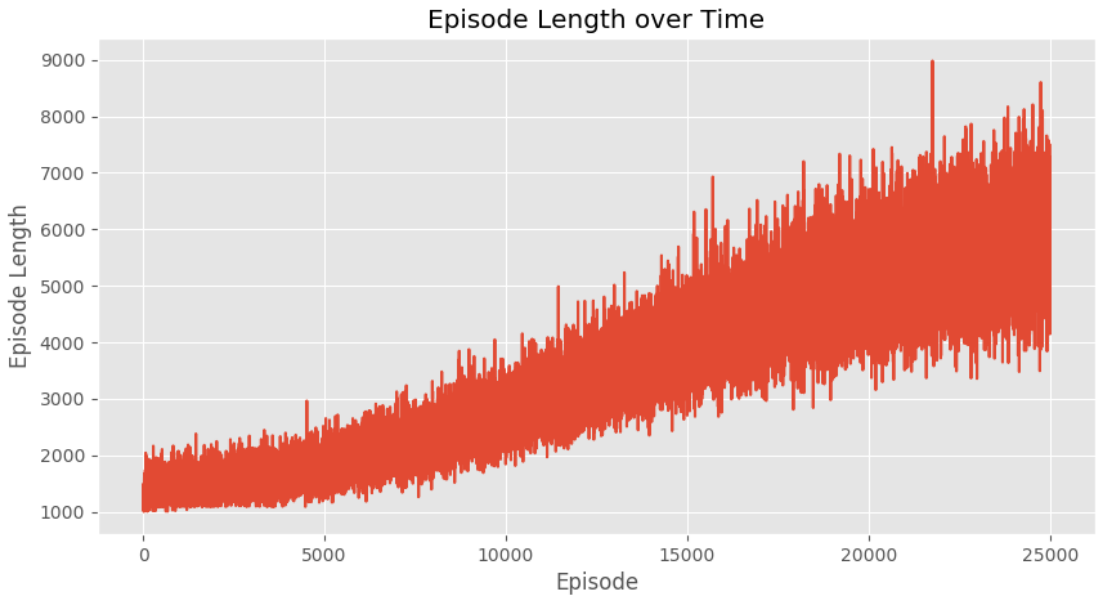
hidden layer neurons = 200
batch_size = 10
learning rate = 1e-4
gamma = 0.99
decay_rate = 0.99
image_size = 80x80

Preprocessing:
La imágen de estrada es transformada en una imágen de 80x80.

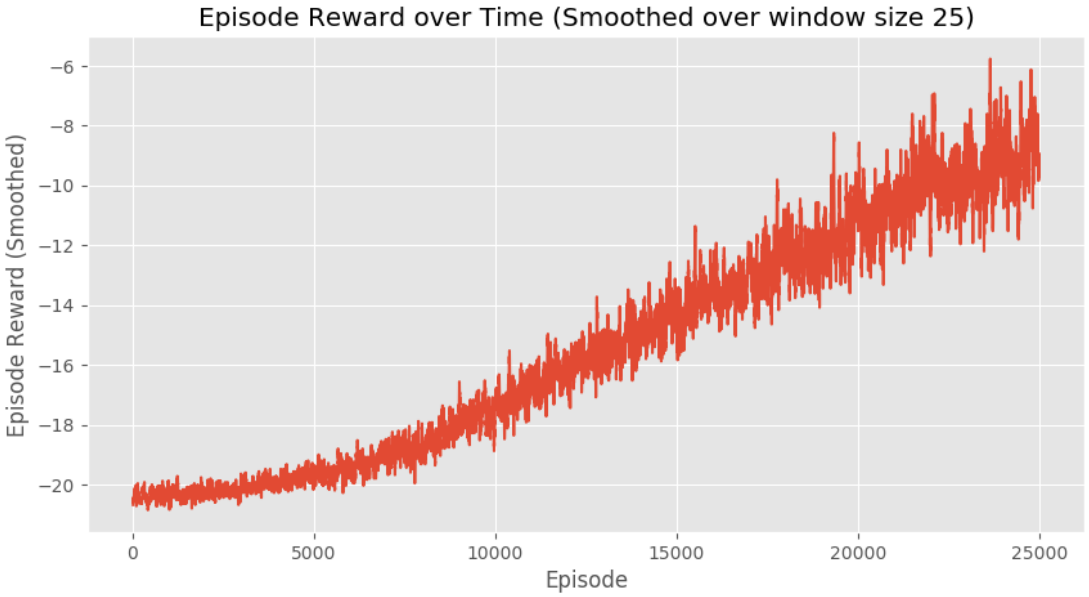
1 - crop
2- downsample
3- erase background
4- binary image

Gracias al preproceso cada uno de nuestros inputs consiste en la diferencia entre el frame actual y el anterior. De esta manera se podrà capturar el movimiento de la pelota como característica.

Rewards:
+1 si la pelota pasa al oponente
-1 si la pelota pasa al agente
0 para todos los demás estados



31. Duración del episodio a lo largo del entrenamiento



32. Reward por episodio a lo largo del tiempo

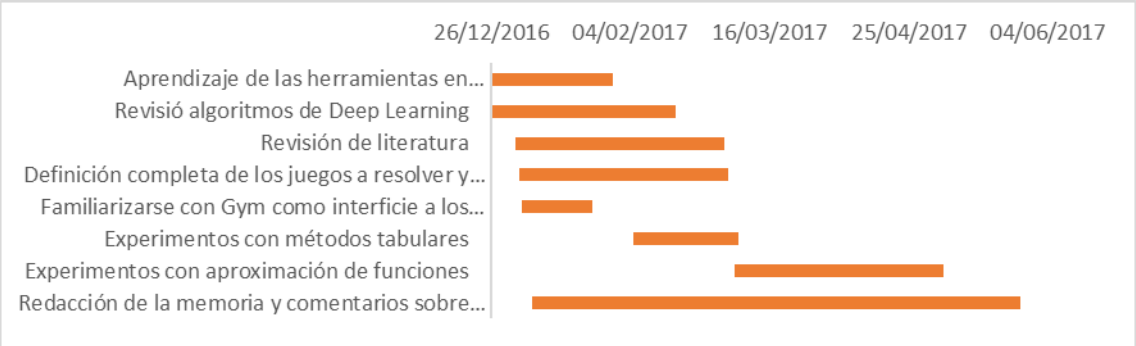
5. Tecnologías utilizadas

- python 3.6
- tensorflow R1.0
- tensorboard
- gym
- numpy
- matplotlib

6. Diagrama de Gantt

Tareas:

Aprendizaje de las herramientas en Reinforcement Learning
Revisión algoritmos de Deep Learning
Revisión de literatura
Definición completa de los juegos a resolver y los métodos que se utilizarán
Familiarizarse con Gym como interficie a los juegos
Experimentos con métodos tabulares
Experimentos con aproximación de funciones
Redacción de la memoria y comentarios sobre los resultados



7. Conclusiones

A continuación se listan las conclusiones más relevantes extraïdas del proyecto:

- Los métodos TD tienen un major rendimiento en la mayoría de aplicacions que los métodos MC
- Si un valor de estado de juego puede ser representado en forma tabular, no hay necesidad de utilizar métodos de aproximación
- Los algoritmos de RL necesitan tricks para funcionar bien, convergir y ser eficientes.
- Los métodos basados en políticas convergen más rápido métodos basados en valor de tan
- Si hay una necesidad de aprender una política estocástica, la política basada debe ser utilizada
- La combinación de valor basado y basado en la política da los mejores resultados (A3C)

Por otra parte, ha sido costoso entrenar algunos de los modelos y se ha necesitado una GPU. Sin haver utilizado una GPU muchos de los experimentos hubiesen tardado más de 5 o 6 veces en ejecutarse. Esto ha sido un impedimento, entre otras coses, para poder abarcar todos los algoritmos que se plateaba comentar al inicio del proyecto.
No se han podido abarcar los métodos A3C que parecen realmente interesantes y son el actual estado del arte en muchas de las tareas. Así, como Trabajo futuro, se plantea implementar el algoritmo A3C y entrenar un modelo para ver como los métodos policy y value based rinden juntos.

8. Bibliografía

- Richard S. Sutton, Andrew G. Barto, Reinforcement Learning: An Introduction, MIT Press, Cambridge, Massachusetts; London, England, 2012.

- Ian Goodfellow, Yoshua Bengio, Aaron Courville, Deep Learning, 2016

- DeepMind, Human Level Control through Deep Reinforcement Learning, Nature, 2015

- Web: https://github.com/dennybritz/reinforcement-learning

- Web: http://karpathy.github.io/2016/05/31/rl/

- Web: https://gym.openai.com/

- Web: https://www.tensorflow.org/