DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Machine Learning

Trabajo práctico 2

Qlearning

Resumen

Integrante	LU	Correo electrónico
Podavini Rey, Martín Gastón	483/12	marto.rey2006@gmail.com

Palabras claves:

TP, 4 en línea, qlearning

Índice

1. Introducción

En este trabajo practico buscamos modelar el juego de 4 en línea y jugadores que utilicen técnicas de q-learning. Además exploraremos distintas políticas de exploración para utilizar en un jugador y veremos qué impacto tiene cada una de ellas en los resultados obtenidos.

Para finalizar, realizaremos distintos experimentos con el objetivo de observar de manera empírica qué sucede al variar parámetros tales como el coeficiente de aprendizaje, la inicialización de la matriz Q o entrenándolo contra otros tipos de jugadores.

Algo interesante para notar aquí es que, en este juego se contará con la participación de dos agentes que compiten entre ellos, por lo que deberá prestarse particular atención en cómo se modelará la etapa de recompensa del algoritmo.

Correcciones a tener en cuenta: con respecto al algoritmo implementado, lo que hacemos es establecer el tablero y hacer jugar al jugador 1, aprender, jugar al 2, aprender y asi sucesivamente. Las recompensas se actualizan si y solo si el jugador gano. No estamos teniendo el problema del tateti (que el jugador juegue cuando no es su turno) ya que siempre respetamos los turnos. Para ejemplificar, el jugador1 esta en el estado s, pone una ficha (tomando una accion a) y obtiene un r (recompensa) y un s' (estado) nuevo. Entonces actualiza el Q(s,a) analizando el maximo Q(s', a') donde a' es la accion que me da el mayor valor de la funcion Q() para ese nuevo estado. Pero no esta colocando una ficha al actualizar el Q del estado anterior, sino que solo observa en donde deberia poner la ficha para obtener el maximo Q() en el nuevo estado.

Hablando con Agustín Gravano sobre esta implementación y recordando el problema del tateti vista en clase podriamos modificar el algoritmo de la siguiente manera:

```
1: While True
2:     jugador1.jugar()
3:     jugador2.jugar()
4:     jugador1.aprender()
5:     jugador2.aprender()
```

Algorithm 1: Otro tipo de modelado

En este caso, los estados modelados ahora son dos jugadas. Es decir, la semántica de un estado es la siguiente: juega el jugador 1, juega el jugador 2, y ese tablero resultante es un estado. Debido a la falta de tiempo no pude implementarlo. Para justificar el algoritmo implementado se puede decir que bajo la definicion de aprendizaje del libro de Mitchell, es decir que dada cierta experiencia y una metrica de performance, con el tiempo el algoritmo mejora su performance, podemos ver que efectivamente el algoritmo aprende y a medida que gana experiencia (jugando mas juegos) resulta ganador una mayor cantidad de veces. Notar que si el jugador2 es un jugador random, cuando se ejecute la funcion aprender, simplemente no hará nada.

Con respecto a los gráficos de experimentacion, la semántica de los ejes es la siguiente: el eje y representa la cantidad de juegos ganados por cada jugador, y el eje x son valores cada 250 juegos. Es decir, cuando en el eje x tenemos el valor 1 se jugaron 250 juegos y se graficó cuantos gano cada uno. Cuando vale 2, son 500 juegos y asi sucesivamente.

2. Modelado

2.1. Modelado del juego

La estructura de datos que utilizaremos para modelar este juego consistirá en una lista de listas: cada una de ellas representará una columna del tablero. En cada turno, se le entregará el

estado actual del tablero al jugador y este elegirá una columna numerada del 0 al n (siendo n el numero de columnas totales) y elegirá dónde colocar su ficha.

Luego de cada jugada, el juego chequeará si el jugador ganó o si ya no hay más movimientos posibles, terminando el juego e informando qué jugador ganó o si fue empate.

El pseudocódigo del juego será entonces el siguiente:

```
1: While True
2:    columna = player.move(tablero)
3:    jugar_ficha(columna, tablero)
4:    if jugador_gano(tablero)
5:       return player
6:    if tablero_lleno(tablero)
7:       return empate
8:    player = otro_jugador(player)
```

Algorithm 2: jugar()

Como ya adelantamos en la introducción, es necesario definir en qué etapa del algoritmo se le asignará la recompensa al algoritmo de q-learning. Una opción bastante sencilla e intuitiva es la de asignar recompensas en el momento en que algún jugador gana la partida. Por ejemplo al ganar una partida, se le asigna una recompensa de 1 al ganador y una recompensa de -1 a su adversario. Siguiendo con el lineamiento anterior, también sería posible asignarles recompensas en el momento en que se empata, por ejemplo, asignándoles a ambos jugadores una recompensa de 0,5.

En caso de que no se haya llegado a un estado final, una posible propuesta es asignar una recompensa por ejemplo de 0.

El pseudocódigo entonces, pasaría a verse de la siguiente manera:

```
While True
    columna = player.move(tablero)
    jugar_ficha(columna, tablero)
    if jugador_gano(tablero)
        player.recompensa(1, tablero)
        otro_jugador(player).recompensa(-1, tablero)
        return player
    if tablero_lleno(tablero)
        player.recompensa(0.5, tablero)
        otro_jugador(player).recompensa(0.5, tablero)
        return empate
    otro_jugador(player).recompensa(0, tablero)
    player = otro_jugador(player)
```

Algorithm 3: jugar()

Utilizando este modelo, en el siguiente apartado pensaremos cómo modelar un jugador que utilice esta estructura para implementar un algoritmo de q-learning.

2.2. Modelado de jugadores

Un jugador estará compuesto por dos métodos básicos, uno que llamaremos *move*() que, dado un estado del tablero devuelve una acción valida (tirar una ficha en la primera columna, en la segunda, etc) y una función *reward*() que nos dirá en qué estado quedó el juego luego de movernos y de que el oponente moviera, y una recompensa que utilizaremos para actualizar la matriz Q en caso de que el jugador lo requiera.

En particular, para la clase Jugador que implemente q-learning, el algoritmo para decidir qué acción realizar vendrá dado de la siguiente manera:

```
acciones_validas = elegir_acciones_validas(tablero)
Para cada acción en acciones_validas
    obtener q para accion
    guardarlo en lista de qs_validos
accion_elegida = elegir_acción(acciones_validas,qs_validos)
retornar accion_elegida
```

Algorithm 4: move(tablero)

Y la función learn, que básicamente consiste en adaptar la función de aprendizaje vista en la teórica:

```
prev_q = getQ(estado_previo, accion_elegida)
maxqnew = tomar maximo q de tomar una acción en el tablero resultante
q[(estado_previo, accion_elegida)] = prev + α * ((reward + γ*maxqnew) - prev)
```

Algorithm 5: learn(tablero,recomensa)

Aquí quedan por definir varios parámetros, entre ellos α, γ y los valores iniciales para la matriz Q. En primera instancia decidimos tomar valores que consideramos apropiados de acuerdo

a resultados empíricos que realizamos a priori y que consideramos razonables. Tomaremos $\alpha = 0{,}4$, $\gamma = 0{,}9$ y todas los valores de Q iniciarán con 1 con el objetivo de alentar la exploración.

Además de la metodología descripta anteriormente se nos ocurrieron otras heurísticas interesantes, que por falta de tiempo o por ser simplemente malas, no indagamos en profundidad. Una de ellas consistía en utilizar el estado siguiente (aquel perteneciente al jugador rival para determinar el mejor movimiento). La idea básica era, para cada acción, tomo su q, realizo una transición por ese estado, invierto el tablero (las fichas rojas pasan a ser azules y las azules pasan a ser rojas) y le resto el mejor q de este otro estado. Pensamos que de esta manera estaríamos agregando información útil sobre potenciales buenas jugadas del contrincante a nuestra fase de decisión, pero en la práctica daba malos resultados.

Otra heurística interesante surgió de la siguiente idea: cuando pierdo, no solo sé que pierdo, sino que sé que el otro ganó y no solo eso, también sé *cómo* ganó. En base a eso una idea muy interesante para explorar era, al momento de recibir una recompensa realizar $learn(invertir(tablero), -recompensa)$, donde *invertir* funcionaría invirtiendo las fichas rojas por azules y viceversa. Si bien implementamos esto, dado que los resultados obtenidos no variaron significativamente y que nos encontrábamos al final de la etapa de experimentación, los siguientes apartados no reflejan ni tienen en cuenta esta posible mejora.

En la siguiente sección terminaremos de definir la función $elegir\_accion()$ basándonos en distintas heurísticas para resolver el problema del Multi-armed bandit.

2.3. Estrategias de movimiento

Aquí plantearemos distintas estrategias que utilizará nuestro jugador al momento de elegir: si decide explorar nuevos caminos o decide utilizar el mejor camino conocido hasta el momento.

La primera heurística que elegimos implementar es una estrategia greedy en donde el jugador tomará un camino random con probabilidad $\epsilon\,\%$ y en caso contrario utilice el mejor brazo conocido. Es claro que dependiendo de este valor $\epsilon\,\%$ nuestro algoritmo será mas propenso a explorar o a elegir el mejor camino posible.

La segunda estrategia es conocida como ϵ-first: en este caso el jugador toma un camino random en las primeras ϵ iteraciones y luego toma el mejor camino conocido. Esperamos que en estas $\epsilon\,\%$ iteraciones el jugador explore una gran cantidad de estados posibles, tras los cuales podrá decidir con más confianza cuál es el mejor camino a elegir.

Como última heurística utilizaremos una estrategia Softmax: basada en una función probabilística, la estrategia Softmax se basa en darle probabilidades distintas a cada acción posible dependiendo de la recompensa esperada de cada una de ellas.

En particular la probabilidad de cada una de las acciones vendrá dada por:

$$P_t(a) = \frac{\exp(q_t(a)/\tau)}{\sum_{i=1}^{n} \exp(q_t(i)/\tau)}$$

Donde $q_t(a)$ es el valor del valor esperado siguiendo la acción a y la variable τ se denomina "parámetro de temperatura". Para temperaturas cercanas a infinito, todas las acciones tienen aproximadamente la misma probabilidad de ser elegidas, mientras que para temperaturas bajas (cercanas a cero) la probabilidad de la acción de mayor recompensa tenderá a 1.

En particular nuestra estrategia consistirá en comenzar con una temperatura alta para favorecer la exploración e ir gradualmente disminuyéndola para favorecer más a aquellas con mayor recompensa. Faltará definir de alguna manera cuál será el valor inicial de τ y de qué manera reduciremos su valor (algunas posibilidades son de manera lineal, logarítmica, etc). De esta manera esperamos que en situaciones donde todos los qs tengan valores similares, el jugador sea más propenso a explorar nuevos caminos mientras que en situaciones críticas (estando cerca de un estado

ganador) elija el camino con mayor recompensa.

2.4. Análisis: espacio de estados

Según el modelo planteado para la resolución del problema, la matriz Q tendrá como columnas los diferentes estados en los que se puede encontrar el tablero y como filas, las acciones que se pueden realizar sobre ese determinado tablero. Dado que la cantidad de estados diferentes en los que se puede encontrar el tablero tiene complejidad PSPACE y que la cantidad la acciones es acotada (a lo sumo m), podemos determinar que la complejidad espacial de nuestro algoritmo está en PSPACE... *not cool*.

Si bien esto hace que la complejidad de explorar cada una de las posibilidades sea extremadamente costosa, consideramos que para la mayoría de los casos no es necesario conocer todo el árbol sino los estados más generales.

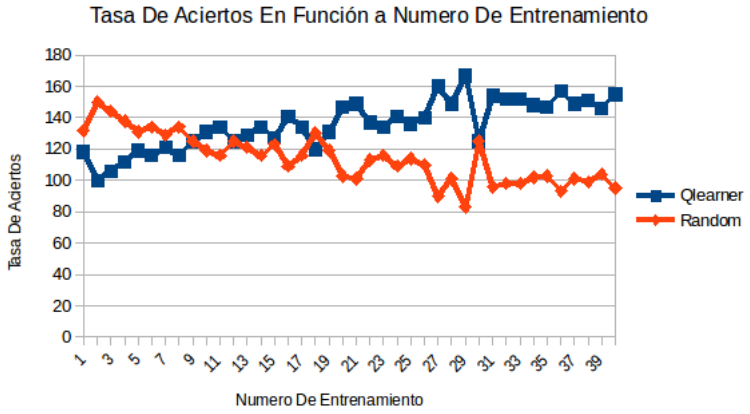
3. Experimentación

3.1. Estrategias vs jugador random

Como primera instancia en la etapa de experimentación, haremos competir a un jugador q-learner contra un jugador que elige entre los posibles movimientos con probabilidad uniforme (random). Utilizaremos las tres estrategias descriptas previamente para ver cómo se comporta cada una.

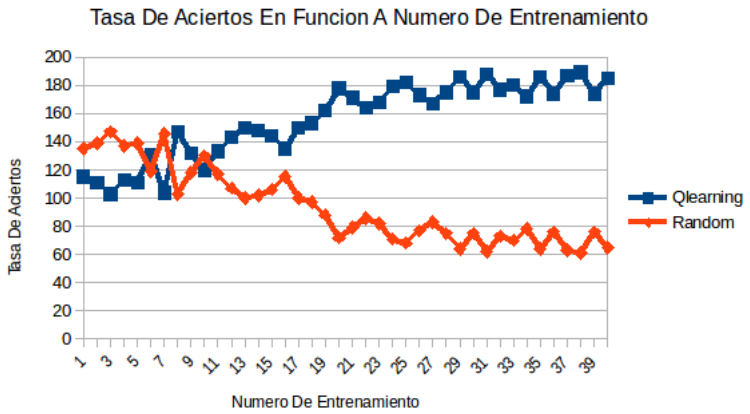
La metodología utilizada para observar cómo evoluciona el algoritmo de q-learning será la de hacer jugar a ambos jugadores 10000 veces, randomizando cual es el que comienza primero en cada iteración. Luego tomaremos estos resultados y cada 250 juegos, graficaremos cuántas veces ganó el algoritmo q-learning.

Para la estrategia greedy, tomando $\epsilon = 0{,}1$ obtuvimos los siguientes resultados:



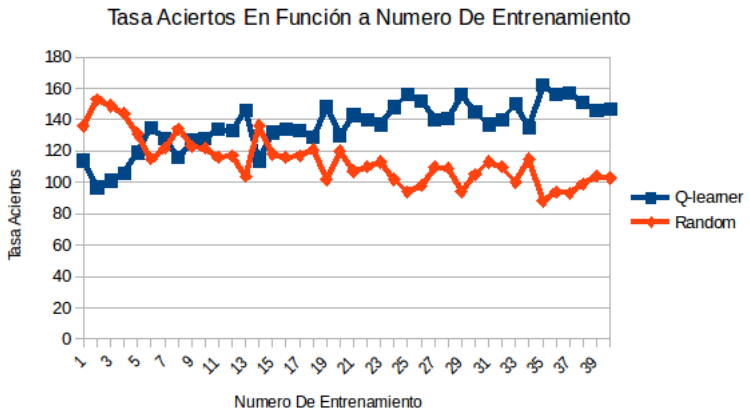
Puede observarse, a medida que pasa el tiempo, cómo el jugador q-learner empieza a converger a una estrategia ganadora, obteniendo al final del experimento un 16 % más de aciertos que el jugador random.

Para la estrategia ϵ-first tomando $\epsilon = 10\,\%$, resultados fueron los siguientes:



Puede verse cómo en el primer 10 % de las iteraciones el algoritmo gana la mitad de las veces y pierde la otra mitad, lo que era de esperarse para una política completamente random. Pasadas esta cantidad de iteraciones, puede verse un repentino crecimiento en la cantidad de partidas ganadas, llegando al final a casi un 32 % más de aciertos que su adversario.

Con la tercera política Soft-Max, decidimos utilizar una temperatura inicial igual a 1 y una función de calor que desciende de manera logarítmica con cada iteración:

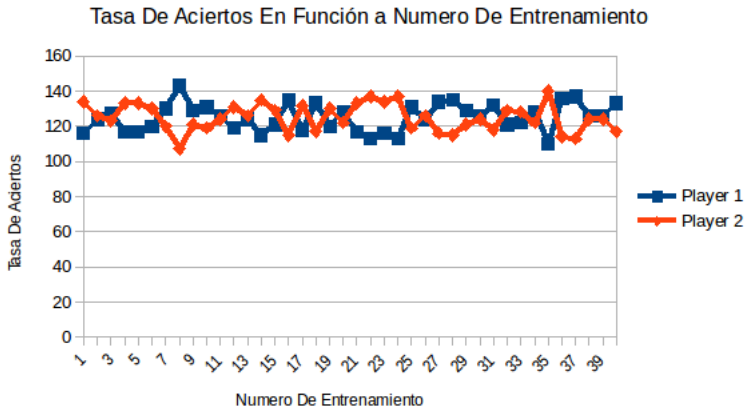


Aquí al igual que en los algoritmos vemos un crecimiento en la tasa de aciertos a medida que se cumplen una mayor cantidad de iteraciones. Aun así esperábamos una mayor diferencia comparado con los otros dos algoritmos, siendo este el que utiliza más información de la matriz Q. Consideramos que posiblemente esto se debe a que los parámetros particulares de este método estuvieran mal ajustados, pero tras intentar con diferentes valores, no logramos obtener una mejora notable.

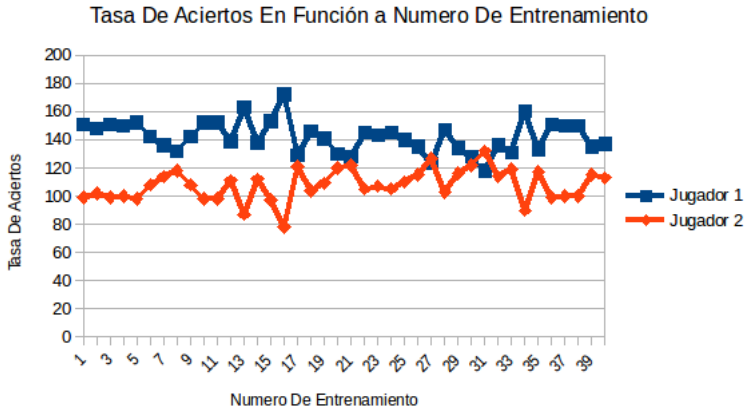
A modo de conclusión, podemos decir que la heurística que se divide en dos etapas, una puramente exploratoria y otra donde se utiliza esta información, resultó ser la mas efectiva. Seguramente esto suceda ya que en la fase exploratoria se llegan a explorar muchos caminos diferentes, los cuales en la etapa de "explotación" utilizaremos el mejor. En cambio en estrategias como greedy y Softmax, al tener siempre una probabilidad considerable de elegir el mejor camino, estemos sesgando al algoritmo para intentar llegar siempre a la misma solución o a soluciones similares que tal vez no sean las óptimas para ese juego en particular.

3.2. Performance de dos q-learners compitiendo

En esta sección analizaremos el comportamiento de dos jugadores q-learners (ambos utilizando una estrategia greedy) compitiendo entre sí y aprendiendo al mismo tiempo. Lo que esperamos observar es que tras un período donde ambos jugadores obtienen una tasa de aciertos cercana al 50 %, esta tasa empiece a converger a un numero más bajo a medida que ambos converjan a una misma estrategia ganadora y empaten en más oportunidades. Los resultados obtenidos fueron los siguientes:



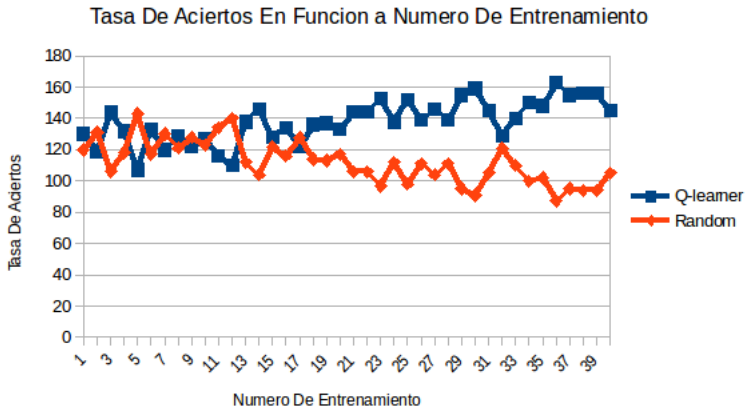
Se realizó la misma experimentación con las otras dos estrategias, obteniéndose resultados similares en ambos casos. Esto no aproxima a los valores esperados, una hipótesis posible es que el jugador que juega primero tenga más ventaja que el que juega segundo y por lo tanto, al estar randomizado el orden en que empiezan, la mitad de las veces gane uno y la otra mitad gane el otro. Para testear esto modificamos el programa para que el jugador 1 siempre comience primero y el jugador 2 siempre comience segundo:



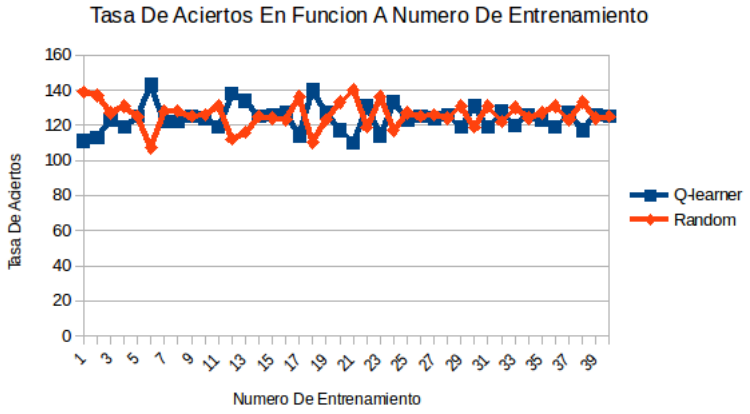
La hipótesis parece confirmarse dado que el primer jugador obtiene un 20 % más de partidas ganadas que el segundo.

3.3. Performance al cambiar la inicialización de Q

En el siguiente experimento buscamos ver qué sucede al variar entre distintos valores iniciales la matriz de Q. Para ello utilizaremos un jugador random y un jugador q-learning con estrategia greedy y variaremos Q. Como primer experimento cambiaremos el valor inicial de 1 a 0. De esta manera, esperamos desalentar la fase de exploración del algoritmo y que se guíe en mayor grado por el camino óptimo encontrado hasta el momento. Los resultados fueron:



Los resultados no parecen ser muy afectados por este cambio. Otra alternativa intermedia que ideamos podría ser inicializar los valores de Q con valores aleatorios en un rango definido, randomizando entonces un poco más cuáles caminos son explorados y cuáles no. Esperamos que de esa manera se exploren caminos que tal vez en otras circunstancias no fueran explorados y que esto mejore la performance a largo plazo. En el siguiente experimento la matriz Q está inicializada con distribución uniforme en el rango $[-1, 1]$:



Como puede verse este tipo de exploración arrojó resultados muy malos, inducidos posiblemente por la dificultad de elegir un buen camino habiendo una tasa tan alta de ruido en el Q inicial.

4. Conclusiones

Como conclusión de este TP, podemos decir que el aprendizaje por refuerzos presenta posibilidades muy interesantes a la hora de entrenar a un autómata, quitando la necesidad de implementar una lógica explicita para el algoritmo o reglas fijas: esta manera de buscar una solución resulta mucho mas maleable y adaptativa. Como desventaja de esta técnica, pudimos ver de manera experimental que resulta muy sensible a los parámetros de aprendizaje y a la manera en la que aprende, debiendo elegirse con cuidado estos parámetros.

Como trabajo futuro sería interesante implementar y experimentar en profundidad con las heurísticas propuestas en el apartado de **Modelado de jugadores**, que presentaba ciertas ideas interesantes y que posiblemente ayudarán a mejorar la performance.