Práctica 2. Técnicas Básicas de Aprendizaje por Refuerzo

Andrés García Meroño, Pablo Guillén Marquina, Alejandro Alonso Puig

Extensiones de Machine Learning, Curso 2024/2025 - Grupo: AAPGAG

Resumen

El documento presenta un estudio comparativo de diferentes técnicas de aprendizaje por refuerzo, con el objetivo de comprender su funcionamiento y desempeño en diversos escenarios. Se implementaron y compararon seis algoritmos: Monte Carlo (primera visita, todas las visitas, off-policy), SARSA y Q-Learning. Los experimentos se realizaron en el entorno Gymnasium, evaluando la proporción de recompensas y el tamaño de los episodios generados por cada algoritmo. Los resultados muestran que todos los algoritmos convergen a valores similares, pero difieren en la velocidad de convergencia y la sensibilidad a la semilla inicial. Q-Learning y Monte Carlo All Visits tienden a converger más rápido. El estudio también verifica la convergencia de la matriz Q para Monte Carlo On Policy All Visits y First Visit.

Los experimentos fueron organizados en notebooks independientes para cada enfoque, facilitando su ejecución y análisis desde un main.ipynb con acceso a Google Colab. El repositorio se encuentra en el siguiente enlace.

1. Introducción

El problema del aprendizaje en entornos complejos se puede abordar con modelos como los bandidos contextuales o los procesos de decisión de Markov con recompensas. Sin embargo, en la práctica, muchas veces no se conoce cómo funciona el entorno ni cuáles son las mejores decisiones a tomar.

Para aprender en estos entornos, los agentes siguen un ciclo donde observan el estado, toman una acción, reciben una recompensa y repiten el proceso. Existen diferentes enfoques:

- **Monte Carlo**, que espera a terminar un episodio para actualizar la estrategia.
- **Diferencias Temporales**, que actualiza la estrategia continuamente.
- **Métodos intermedios**, que consideran solo algunas recompensas del episodio.

Cuando hay demasiados estados y acciones, se usan técnicas de aproximación como redes neuronales (aprendizaje profundo) o métodos de optimización directa de la estrategia, como el modelo **Actor-Crítico**.

El **objetivo** de este estudio es comprender el funcionamiento del entorno Gymnasium y utilizarlo como plataforma para analizar y comparar distintas técnicas de aprendizaje por refuerzo. A través de experimentos y pruebas, se evaluará el desempeño de cada método en diferentes escenarios, identificando sus ventajas, desventajas y posibles aplicaciones.

Para ello se implementan varias familias de métodos:

- **Monte Carlo**: on-policy primera visita y todas las visitas; off-policy para estimar π y Q
- **Diferencias Temporales**: SARSA y Q-Learning

El documento comienza describiendo la estructura general del algoritmo de entrenamiento y de los agentes, seguido de la descripción de cada uno de los agentes implementados. Continúa con la metodología empleada y comentando los resultados obtenidos. Por último un apartado de conclusiones.

Estos experimentos permiten comparar la eficiencia de cada estrategia y el impacto de sus hiperparámetros en la convergencia y estabilidad del aprendizaje.

2. Algoritmos Implementados

2.1. Estructura del algoritmo de entrenamiento

Todos los estudios tienen en común el siguiente algoritmo que nos permite entrenar cada uno de los agentes implementados:

- inicializar el agente: agent.initAgent()
- repetir hasta num_episodes:
  - resetear el entorno: env.reset(seed)
  - indicar al agente que comienza un nuevo episodio: agent.iniEpisode()
  - mientras no terminado o truncado:
    - opcionalmente decaer epsilon: agent.decay_epsilon()
    - obtener una nueva acción: agent.get_action(env, state)
    - ejecutar la acción: env.step(action)
    - actualizar el estado del agente a nivel de step: agent.updateStep(state, action, reward, terminated, truncated, next_state)
    - establecer el nuevo estado
  - actualizar el estado del agente a nivel de episodio: agent.updateEpisode()

2.2. Estructura de todos los agentes

Para facilitar el intercambio de agentes, todos ellos tienen los siguientes métodos:

- __init__(self, env, epsilon, discount_factor): Inicializa la clase con los parámetros comunes: epsilon y fator de descuento
- initAgent(self): inicializa el agente con la configuración inicial parametros del epsilon greedy y su decaimiento

- initEpisode(self): Inicializa el agente con un nuevo episodio
- get_action(self, env, state): Conocido el estado actual, devuelve una acción
- updateStep(self, state, action, reward, terminated, next_state): Actualiza el estado interno del agente a partir de la información obtenida tras ejecutar la acción a nivel de step (Gymnasium)
- updateEpisode(self): Actualiza el agente a nivel de episodio
- decay_epsilon(self): implementa el mecanismo de decaimiento del parámetro epsilon
- pi_star_from_Q(self, Q): obtiene la política óptima a partir de Q

La elección de estos métodos encaja en la estructura de algoritmo de entrenamiento de manera que sin cambiar la estructura del entrenamiento se pueda cambiar de implmentación de agente.

Así se ha implementado 6 clases:

- **FrozenAgentGreedy**: agente que implementa una variante similar a Monte Carlo Todas las visitas según el cuaderno del tutor
- **FrozenAgentMC_On_First**: agente que implementa Monte Carlos On Policy Primera visita
- **FrozenAgentMC_On_All**: agente que implementa Monte Carlos On Policy Todas las visitas
- **FrozenAgentMC_Off_Pi**: agente que implementa Monte Carlo Off Policy para estimar Pi
- **FrozenAgentSARSA**: agente que implementa Diferencias Temporales SARSA
- **FrozenAgentQ_Learning**: agente que implementa Diferencias Temporales Q-Learning

2.3. FrozenAgentGreedy

Este agente estima la matriz Q(S,A) usando una versión incremental de Monte Carlo on policy todas las visitas.

El paso getAction implementa una política epsilon-greedy a partir de un epsilon-soft. Esta política asigna una probabilidad de epsilon/nA a cada acción excepto a la acción que maximiza Q(S,A) para el estado actual que le asigna una probabilidad de 1-epsilon(nA-1)/nA, donde nA es el número de acciones posibles.

El paso updateStep apila el par (estado,acción) en el episodio y acumula progresivamente el valor de la recompensa de hacia adelante en la variable result_sum (nótese que la técnia de Montecarlo hace esto pero al revés)

El paso updateEpisode actualiza el valor de Q(S,A) para cada uno de los pares (estado,acción) del episodio en el sentido en que se ha indo generando. Para cada par se realiza:

- incrementar el número de visitas del par (S,A)
- asignar α=1 / número de visitas del par (S,A)
- Q(S,A) = Q(S,A) + α * (result_sum – Q(S,A))

Por último termina acumulando resultados:

- list_stats = suma de todas las recompensas acumuladas / episodios
- list_episodes = suma len(episodio) / episodios

2.4. Monte Carlo on-policy first visit (FrozenAgentMC_On_First)

Esta clase implementa el algoritmo descrito en Sutton y Barto [1, 2]., estima π y Q.

Inicializa la variable Q(S,A) con números aleatorios entre (0,1) y la policy(S,A) como una epsilon soft greedy como se ha descrito en el apartado anterior.

También inicializa dos variables, una (returns) para almacenar la suma de todas las recompensas por estado-acción y otra (nreturns) para almacenar la cantidad de elementos de todas las recompensas por estado-acción.

En el paso de inicialización del episodio (initEpisode) crea un conjunto vacío para determinar la primera vez que el par (S,A) aparece en el episodio

En el paso getAction devuelve aleatoriamente una acción de entre las posibles conforme a la probabilidad almacenada en la variable policy del estado actual.

En el paso updateStep y para optimizar el cálculo al finalizar el episodio, se determina si el par (estado,acción) actual ya se ha visitado antes (reflejado en la variable existe). También almacena el par (estado,acción) en el conjunto de nodos visitados. Y, por último, apila la terna (estado, accion, recompensa, existe) en el episodio.

En el paso updateEpisode recorre en sentido inverso (del último al primero) la secuencia anterior. Para cada elemento de la secuencia realiza:

- Acumular el valor G = discount_factor * G + recompensa
- si el nodo es la primera vez que aparece en la secuencia (existe=False):
  - Acumula G a returns(S,A)
  - Incrementa en 1 el valor de nreturns(S,A)
  - Actualiza el valor de Q(S,A) = returns(S,A) / nreturns(S,A)
  - Actualiza la política para S usando una epsilon soft greedy
- Acumula resultados: proporción de recompensas, y proporción de episodios

2.5. Monte Carlo on-policy all visit (FrozenAgentMC_On_All)

Esta clase es similar a la anterior salvo que la actualización de Q y de la política se haca para cada uno de los pares (estado, acción) y no sólo con la primera vez que aparece una ocurrencia (estado, acción) durante cada episodio.

Por lo tanto no hace falta registrar las visitas, ni determinar si un par ya se ha visitado. El resto es idéntico a Monte Carlo first visit.

2.6. Monte Carlo off policy pi (FrozenAgentMC_Off_Pi)

Este agente implementa Monte Carlo Off Policy. Utiliza una política para generar episodios (bpolicy) a la vez que estima π y Q.

En la inicialización del agente (initAgent), además de Q, policy, returns y nreturns (descritas en los puntos anteriores), inicializa una variable C para acumular el muestreo por importancia y una política soft (bpolicy)

En cada comienzo de un episodio (initEpisode) se vuelve a inicializar la política soft (bpolicy) usando la técnica epsilon soft greedy descrita enteriormente.

En el paso getAction devuelve aleatoriamente una acción de entre las posibles conforme a la probabilidad almacenada en la variable bpolicy del estado actual.

El paso updateStep se limita a apilar la terna (estado, acción, recompensa) en el episodio

El paso updateEpisode recorre el episodio en sentido inverso. Para cada elemento de la secuencia realiza:

- Acumular el valor G = discount_factor * G + recompensa
- Acumular a C(S,A) el valor W
- Acumular a Q(S,A) el valor W/C(S,A) * (G – Q(S,A))
- Actualiza la política para S según epsilon soft greedy (policy)
- Si la acción que se tomó no coincide la acción usando pi (policy) se sale del episodio
- En otro caso actualiza W = W / bpolicy(S,A)

por último, acumula resultados: proporción de recompensas, y proporción de episodios

2.7. Diferencias Temporales SARSA (FrozenAgentSARSA)

Este agente implementa Diferencias Temporales SARSA

Además de los hiperparámetros epsilon, discount_factor, necesita que se suministre alpha: tamaño de paso.

En la inicialización del agente (initAgent) sólo usa Q, se inicializa todo a 1 excepto los nodos terminales que se ponen a 0: Q(T,A)=0

En la inicialización del episodio (initEpisode) el retorno se pone a 0

En el paso getAction se implementa un epsiol soft greedy

En el paso updateStep se actualiza el retorno según la recompensa actual y el factor de descuento: G = discount_factor * G + recompensa.

Si la acción no es terminal actualiza Q:

Q(S,A) = Q(S,A) + alpha * (recompensa + discount_factor * Q(S_next,A_next) – Q(S,A))

Si la acción es terminal actualiza Q:

Q(S,A) = Q(S,A) + alpha * (recompensa – Q(S,A))

El valor de A_next se obtiene llamando a getAction de S_next

El paso updateEpisode sólo acumula resultados: proporción de recompensas, y proporción de episodios

2.8. Diferencias Temporales Q-Learning (FrozenAgentQ_Learning)

Este agente implementa Diferencias Temporales Q-Learning. Es exactamente igual al método SARSA con la única diferencia en que cuando actualiza Q(S,A) no utiliza Q(S_next,A_next) si no el valor máximo de Q(S_next) independientemete de la acción.

3. Metodología y Experimentos

Comenzamos con comprender cómo funciona gymnasium usando la estructura del algoritmo de entrenamiento y el agente que implementa el código del tutor. Observamos cómo aumenta la proporción de recompensas conforme aumentan los episodios.

Para verificar la correcta implementación de cada agente se entrena individualmente cada uno de ellos y se comprueba que la proporción de recompensas aumentaba al aumentar los episodios.

Después se comparó dos a dos algunos de los agentes implementados, en concreto se comparó:

- Monte Carlo on policy first visit y all visit
- SARSA y Q-Learning

También se verificó que una misa política debe generar la misma Q(S,A) con independecia del agente

4. Resultados y Análisis

4.1. Monte Carlo on policy first visit y all visit

La comparativa Monte Carlo on policy y off policy arrojó que es muy sensible a la semilla que se utilice, y que para poder comparar ambos algoritmos habría que asegurarse que ambos usaban las mismas semillas de partida para los números aleatorios.

Aparentemente puede parece una mejor que la otra, pero repitiendo el experimento con otros valores de semilla se puede observar que la elección de la semilla inlfuye en el resultado. Incluso se detectó una semilla en la que MC All visit no converge. Se decide inicializar la semilla al comienzo de cada entrenamiento.

Se observa que las dos gráficas son idénticas, lo cual nos hace pensar que los dos algoritmos producen la misma proporción de recompensas (en igualdad de condiciones).

Esto es debido a que ambos algoritmos implementan la misma política (epsilon soft greedy) y que ambos actualizan la política al menos una vez por cada par estado-acción. El MC Primera visita sólo actualiza la política una vez y el MC Todas las visitas la actualiza todas las veces. La media acumulada por recorrer todas las visitas está muy próxima a las recompensas acumuladas en la primera visita, así que la actualización de la política tiene los mismos efectos en ambos algoritmos

Se comprueba que la Q obtenida en ambos algoritmos son diferentes, pero la política óptima resultante es idéntica, por lo que eso explica que ambos hubieran generado los mismos episodios y por ello la misma gráfica de proporción de recompensas.

Con el mapa de 8x8 al contar con episodios más largos se observan pequeñas diferencias en las ganancias obtenidas. Esto es debido a que la cantidad de ganancias acumuladas por cada par estado-acción es mayor en MC todas las visitas que en MC priemra visita, por lo que es más probable que las dos medias sean diferentes y provoquen algún cambio en las políticas resultantes.

Se comprueba los valores de Q y de la política óptima y esta vez sí que se observan diferencias en la política resultante, lo que justifica que ambos algoritmos generen episodios diferentes.

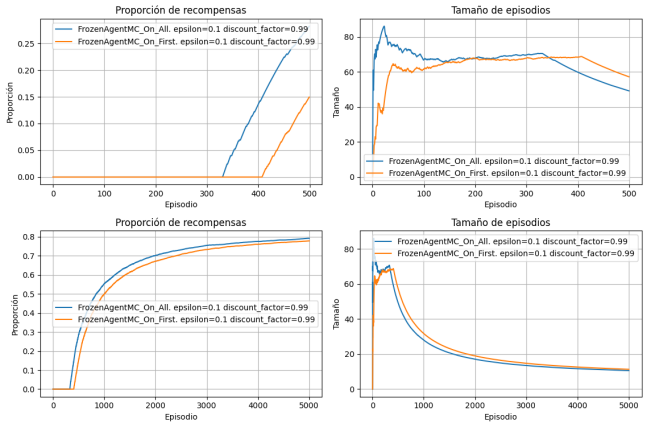

Figura 1: Ejemplo de ejecución de comparativa MC First y MC All visit

4.2. SARSA y Q-Learning

Partiendo de la lección aprendida con Monte Carlo, se comienza estableciendo la misma semilla para cada agente.

Esta comparativa muestra que Q-Learning suele ser más rápido al inicio, y SARSA eventualmente se acerca a su desempeño. Ambos convergen a una proporción de recompensas alta, aunque Q-Learning mantiene una ligera ventaja.

En la siguiente figura se puede comprobar estos resultado:

4.3. Convergencia de Q

Este estudio pretende mostrar que, ante una misma política y diferentes algoritmos, el retorno esperado (esto es Q) converge a un valor óptimo.

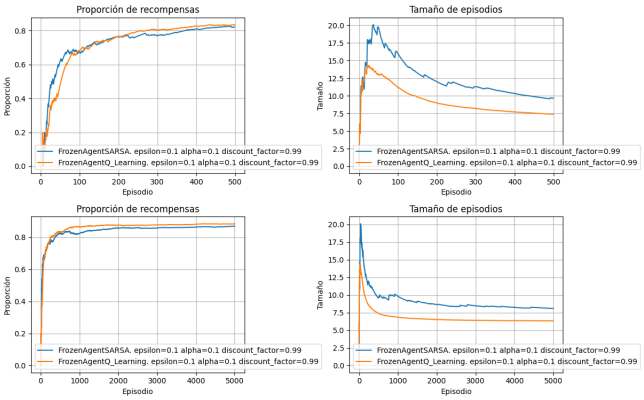

Figura 2: Ejemplo de ejecución de comparativa SARSA y Q-Learning

Para mostrar este resultado se usará la evolución de la suma de la matriz Q conforme se generan episodios. Es de esperar que esta suma converja y sea la misma para dos agentes.

Definimos dos agentes: Monte Carlo On Policy All Visits y Monte Carlo On Policy First visit. Ejecutamos en entrenameiento con el mapa de 4x4 y de 8x8 y obtenemos la evolución de la suma de la matriz Q.

En la siguiente imagen se puede verificar que al principio hay diferencias pero conforme se generan episodios la suma de Q converge al mismo valor para ambos agentes.

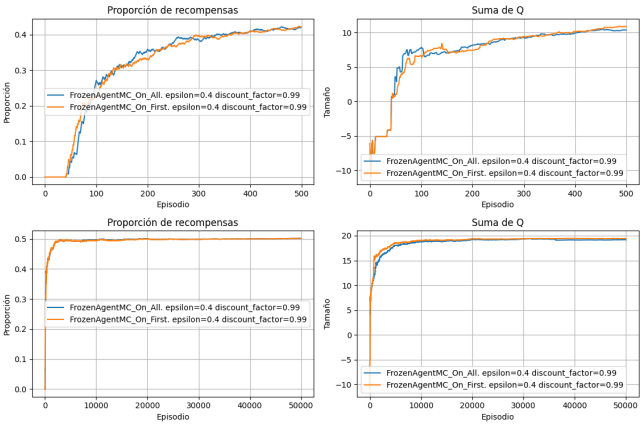

Figura 3: Convergencia de Q

5. Conclusiones

Ginamsium es un entorno de trabajo que permite implementar agentes usando aprendizaje por refuerzo. Se ha diseñado una estructura de agentes y de algoritmo de entrenamiento que facilita el intercambio de agentes lo que simplifica la creación de diferentes escenarios de casos de uso.

Se han implementado varios agentes usando diferentes técnicas tabulares y comparando algunas de ellas.

La elección de la semilla inlfuye en el resultado obtenido. Pudiendo llegar a no terminar un episodio.

Generalmente todos los agentes convergen a valores similares pero unos lo hacen antes que otros. Si bien esto es debido al

azar, Monte Carlo All Visits suele converger un poco antes que First visit y Q-Learning antes que SARSA.

Se ha comporbado la convergencia de la matriz Q en Monte Carlo On Policy All Visits y Monte Carlo On Policy First visit.

6. Referencias

Referencias

[1] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction (2nd ed.)*. MIT Press.

[2] Slides y material de la asignatura Extensiones de Machine Learning, Universidad de Murcia.