

Proyecto # 3: Atari-Pong con RL

Aplicación de algoritmos de Machine Learning para el entrenamiento de un video juego clásico

Daniel Estrada Acevedo

Instituto de Física—Universidad de Antioquia

Resumen

El trabajo que se presenta a continuación corresponde al proyecto de la tercera unidad del curso de Computación científica avanzada I, y consiste en una primera aproximación al paradigma de *Machine Learning, Reinforcement Learnign*. Acá se plasma, de forma sintetizada y sin adentrarse mucho en los modelos matemáticos internos, algunos de los conceptos básicos más importantes de este tipo de algoritmos de inteligencia artificial, además, como ejemplo ilustrativo y con base en el entrenamiento de un problema clásico de control (cart-pole), se implementa el juego de Atari Tele-Pong como un entorno para entrenar un agente por aprendizaje reforzado usando los algoritmos A2C y DQN. Al final podrá ver una serie de posibles causante del bajo desempeño que se consiguió en el aprendizaje del agente.

Palabras clave— Atari, Pong, Machine Learning, Reinforcement learning, Stable-Baselines, Python.

1. Introducción

El aprendizaje por refuerzo - *Reinforcement Learning* (RL), es uno de los tres paradigmas básicos de *Machine Learning* (ML) y consiste en el estudio de como un agente (*Agent*) puede interactuar con su entorno (*Environment*), modelado típicamente como un proceso de decisión Markoviano (MDP), para aprender una regla (*Policy*) que permita maximizar las recompensas (*Reward*) recibidas por una tarea (*Action*) ejecutada. RL ha experimentado un crecimiento importante en interés y atención debido a resultados prometedores en el control de sistemas en robótica, Juegos de Go, atari y otros video juegos competitivos, entre otras aplicaciones posibles de este tipo de algoritmos de inteligencia artificial. [4, 6]

Atacar un problema mediante la aplicación de algoritmos de RL requería forzosamente tener un entendimiento profundo en los conceptos relacionados con el entrenamiento no supervisado del agente, de cómo modelar la función de decisión por medio, usualmente, de redes neuronales, de cómo optimizar el proceso por medio del diseño óptimo del entorno y la función de recompensa con los cuales el agente se entrenará, entre otros aspectos. Sin embargo, hoy día se han desarrollado increíbles y muy completos proyectos y librerías que hacen de estos procesos una tarea más accesible para el público no especializado como lo son TensorFlow, PyTorch y OpenAI.

En este trabajo, como primera aproximación a este paradigma de la programación, se desarrolla el entrenamiento por RL del clásico juego de Atari conocido como Pong (o Tele-Pong). Para ello, se hará uso de varias librerías de Python enfocadas al desarrollo de programas de RL, en especial de las librerías **Stable-Baselines** y **gym** del proyecto de código abierto OpenAI.

Al final, podrá notar que la implementación de los agentes y los entornos, aunque es una tarea sencilla con las herramientas mencionadas, requiere varias nociones teóricas que permitan definir de la mejor forma las propiedades del sistema a estudiar y los hiperparámetros de los modelos, esto se refleja en los resultados obtenidos para el entrenamiento del agente que juega Pong.

2. Marco Teórico

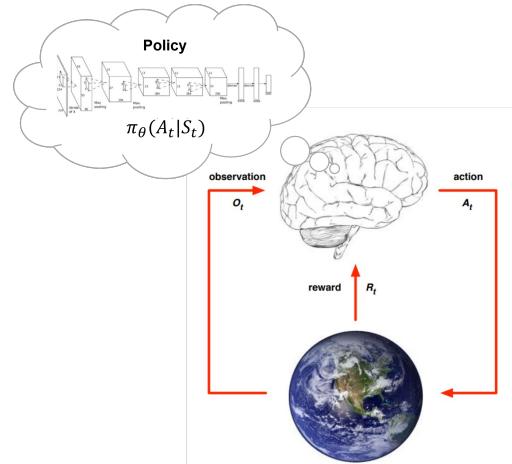


Figura 1: Representación gráfica de la idea general del entrenamiento de un agente con RL.[8]

Considere el problema de la conducción autónoma, aquí usted necesita entrenar una inteligencia artificial de tal forma que en el transcurso de la ejecución de su tarea (conducir) el algoritmo tome las mejores decisiones según el espacio en el que se mueve. Para ello se puede llevar

acabo un proceso de entrenamiento basado en recompensas, de forma que se produzca un estimulo positivo si el algoritmo gira el vehículo en la dirección correcta, y se produce un estimulo negativo o no se da, si el algoritmo se equivoca [5]. Esto es similar a los procesos que se utilizan en la psicología animal, ya que los cerebros biológicos están programados para interpretar las señales de dolor y el hambre como refuerzos negativos, e interpretar el placer y la ingesta de alimentos como refuerzos positivos y en algunas circunstancias, los animales pueden aprender a adoptar comportamientos que optimicen estas recompensas. [3]

De forma Básica, en los algoritmos de RL existe un agente (AI) que aplica un MDP en la interacción con un entorno específico y recibe una recompensa según la acción que ejecute sobre el entorno, el proceso de entrenamiento ejecuta recursivamente está interacción y se enfocado en maximizar la recompensa acumulada. Formalmente el MDP se define como una 4-tupla $\langle \mathcal{S}, \mathcal{A}, \mathcal{O}, p, r \rangle$, donde, \mathcal{S} es el espacio de estados, \mathcal{A} es el espacio de acciones, \mathcal{O} espacio de observaciones p es la probabilidad que define la transición entre estados y r es una función de recompensa ($r : S \times A \rightarrow \mathbb{R}$). [6, 5]

Resumiendo el proceso como lo plantea [8], si se parte de un estado $S_t \in \mathcal{S}$, entonces (ver Figura 2):

El agente:

- Recibe un observable por parte del entorno $O_t \in \mathcal{O}$.
- Por medio de una regla $\pi_\theta(A_t|S_t)$ realiza una acción $A_t \in \mathcal{A}$ sobre el entorno.
- Recibe una recompensa $R_{t+1} \in \mathbb{R}$ y un nuevo observable O_{t+1} (asociado a S_{t+1}).
- Repite el proceso con el nuevo observable.

El entorno:

- Entrega un observable O_t .
- Recibe una acción A_t .
- Entrega una recompensa R_{t+1} definida por $r(S_t, A_t, S_{t+1})$ y actualiza su estado acorde a la probabilidad $p(S_{t+1}|A_t, S_t)$ ¹.
- Dado el nuevo estado S_{t+1} , retorna el observable O_{t+1} para repetir el proceso.

En cada paso de del proceso de entrenamiento, se utiliza la recompensa acumulada para ajustar los pesos θ del modelo de decisión del agente (red neuronal) por medio de diversos algoritmos como: *Trust Region Policy Optimization* (TRPO), *Deep Deterministic Policy Gradients* (DDPG) , *Proximal Policy Optimization* (PPO), *Actor Critic* (AC), *Noisy Cross-Entropy* (CEM), *Deep Q-Network* (DQN), entre otros. Lo cual, ya dependen del problema en particular. De esta forma, se construye la regla π_θ que mapea el estado del sistema S_t en una distribución de probabilidad para A_t de la forma más óptima. [4]

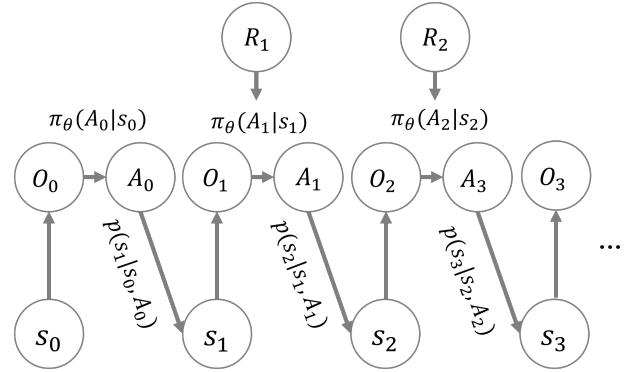


Figura 2: Representación diagramática del proceso base de RL.

3. Metodología

Las implementaciones de algoritmos de RL son altamente sensibles al cambio en los hiperparámetros e incluso en las propiedades del entorno, además de que un solo problema se puede atacar de diversas formas implementando diferentes arquitecturas [4]. Por esto, con el fin de establecer un proceso claro para el entrenamiento del agente que jugará Pong, se decidió tomar como base el entrenamiento de un entorno del *toolkit* de OpenAI gym, *CartPole-v1*, el cual corresponde al problema cart-pole descrito por Barto, Sutton, and Anderson en [2].

Este entorno consiste en una barra unida por una articulación a un carro que se mueve a lo largo de una pista sin fricción y que se controla aplicando una fuerza de 1 o -1 al carro, es decir, moviéndolo a la izquierda o la derecha. El péndulo comienza en posición vertical y el objetivo es evitar que se caiga. Se proporciona una recompensa de 1 por cada paso de tiempo que el poste permanece en posición vertical. El episodio termina cuando el poste está a más de 15° de la vertical, o el carro se mueve más de 2.4 unidades desde el centro.[7]

El agente que actúa en este entorno se implementa con la librería **Stable-Baselines** y se consiguió, a través de su entrenamiento, definir los siguientes pasos a seguir en el proceso:

- **Se crea el entorno para el cual se va a entrenar el agente.** Para esto, gym provee una amplia variedad de entornos ya funcionales que incluyen problemas de MuJoCo, robótica, Atari, entre otros. Y también ofrece una interfaz base para la creación de entornos personalizados. [7, 9, 1]
- **Se crea el agente.** **Stable-Baselines** contiene varios algoritmos de RL que se pueden implementar y basta con especificar el modelo que se usará y el entorno en el que se va a entrenar.
- **Se desencadena el proceso de entrenamiento.** El modelo de agente definido se entrena indicándole el número de pasos que va a realizar y algunos otros argumentos como el tipo de información que

¹Note que los problemas tipo Atari como el que acá se estudia son deterministas, es decir, para una acción dada $p(S_{t+1}|A_t, S_t) = 1$. [6]

va a arrojar en el reporte de entrenamiento, lo que se conoce como *callbacks*.

- **Se extrae la información del entrenamiento.** En el proceso de aprendizaje se pueden generar archivos con la información extraída a partir de los *callbacks*, y gracias al visualizador de TensorFlow, *tensorboard*, es posible obtener gráficamente la información.
- **Se guarda el modelo** entrenado si se consigue un buen desempeño.

La principal razón para escoger este problema de control como modelo para implementar el agente que juega Pong es la similitud entre los espacios de acción de los dos entornos (\mathcal{A}). Para cart-pole existen dos posibles acciones que el agente puede realizar, mover el carro a la izquierda o moverlo a la derecha, y para el juego de Pong, se puede mover la “raqueta” hacia arriba o hacia abajo, o no moverla. Ahora bien, es usual que los problemas tipo Atari se manejen de una forma dista, la mayoría de las implementaciones se basan en el uso de las capturas de pantalla de las partidas como los observables que recibirá el agente (matriz con los píxeles de la pantalla de juego), y a partir de modelos que incluyen capas de redes neuronales convolucionales, se extraen las características que le permiten al agente aprender las acciones correctas, de hecho, en gym hay una gran lista de juegos ya implementados [7, 6]. Sin embargo, con el fin de mantener el problema lo más simple posible y que sea ilustrativo, se decide implementar de forma manual el entorno (juego) Pong (a partir de la librería Pygame de Python y las herramientas disponibles en gym para garantizar la compatibilidad con los algoritmos de Stable-Baselines) y usar como observables variables fáciles de extraer de las partidas, como lo son, la posición de la paleta, la posición de la bola o su velocidad.

Para la creación de entornos personalizados compatibles con los desarrollos de OpenAI y que siga la estructura de los entornos de gym, se debe encapsular las funcionalidades del sistema personalizado en una clase que herede las propiedades de gym.Env, hecho esto, basta con sobrecargar 3 funciones principales que son vitales para el desarrollo de los procesos de entrenamiento [7, 9]:

- **reset()** El cual se llama al inicio de cada episodio para restablecer el estado inicial del entorno.
- **step(action)** Que se llama cuando el agente ejecuta una acción sobre el entorno, este método de clase define la transición de estados y debe retornar el observable de la próxima iteración, la recompensa por la acción realizada, un indicador de final del episodio y un diccionario con alguna información que quisiera incluirse.
- **render(mode='human')** El cual permite la visualización del agente en acción, en el caso particular de Pong, permitirá ver las partidas que juega el agente. Es posible definir varios modos de visualización para optimizar el entrenamiento.

Además, es necesario establecer dos atributos importantes del entorno (ambos correspondientes a un gym.space) [7, 9]:

- **observation_space** Que define el espacio de observaciones, en específico, los posibles valores y la forma o tamaño de la información que se entrega al agente como observable asociado al estado del entorno.
- **action_space** Es un objeto que define el espacio de acciones, es decir, las posibles acciones que puede ejecutar el agente.

Con esto, solo queda seguir el proceso que se describió antes, sobre el entrenamiento del agente pero ahora usando este entorno personalizado.

4. Resultados

Cart-Pole:

Para esta implementación se creó un agente con regla o modelo de decisión dado por una red neuronal multi-capas ('MlpPolicy' - *Multi layer perceptrón policy*) que se entrena bajo el algoritmo A2C (*Advantage Actor Critic*) de stable-baselines3, para actuar en el entorno de gym ‘CartPole-v1’. La programación de este problema resulta bastante sencilla con estas herramientas basadas en OpenAI.

Evaluando en 20 pasos el comportamiento del agente se pudo observar que: Antes del entrenamiento la aleatoriedad de las acciones lleva a un puntajes (recompensa acumulada durante un episodio) de $\approx 18 \pm 2$, y luego del entrenamiento de 20K pasos, el agente alcanza un marcador promedio de aproximadamente 485 ± 46 . Los resultados del este proceso se almacenaron, con lo cual, se pudo obtener las gráficas de la Figura 3 que muestran la evolución del agente a lo largo del entrenamiento.

De la primera gráfica se puede observar, a pesar de la inestabilidad intrínseca del proceso de entrenamiento, como el puntaje medio del agente tiene una tendencia ascendente, lo cual se evidenció en la evaluación de 20 pasos que se mencionaba. En la segunda gráfica se plasma la perdida de entropía, que en estos problemas representa la disminución en la aleatoriedad de las acciones del agente. La última de estas gráficas representa la magnitud del cambio de las reglas de decisión a lo largo del entrenamiento, según la tendencia de esta curva, se podría decir que se realizó un proceso de entrenamiento suficientemente grande, ya que, si se realizaran más, no debería haber un cambio drástico con respecto al comportamiento ya alcanzado por el agente. [1]

Atari-Pong:

Con un entendimiento más amplio del problema gracias a la implementación anterior, se prosiguió con el objetivo principal. Implementar un algoritmo de ML con RL que jugara el clásico de Atari Tele-Pong.

Lo primero que se hizo fue la creación del entorno, que, aunque era posible usar una versión bien diseñada y lista para entrenar, por ejemplo, ‘PongNoFrameskip-v4’ como se comentó antes, recordando que el propósito de este trabajo es educativo, se decidió en su lugar utilizar gym



Figura 3: Gráficas generadas con Tensorbaord de algunas métricas de entrenamiento para el problema cart-pole.

para hacer un entorno personalizado, es decir, para implementar desde cero el juego, para lo cual se usó Pygame y el código de malreddysid² como base.

El entorno se construyó orientado a usar, al igual que en cart-pole, parámetros físicos del sistema como observables. En este caso se definió como observable $O_t = (p_{r1}, p_{r2}, p_{bx}, p_{by}, d_{bx}, d_{by})(t)$, cuyas entradas corresponde, respectivamente, a las posiciones de la raqueta del agente (1) y la de robot o enemigo (2), la posición en x y en y de la pelota, y la dirección de movimiento en x y y de la pelota también. El otro aspecto importante para mencionar es que se escogió como función o regla de recompensa el siguiente sistema: Retornar un valor de $+1$ si se produce un cambio positivo en el puntaje del juego, lo cual se logra golpeando la pelota. Retornar -1 cuando se pierde la partida, es decir cuando llega al extremo izquierdo de la ventana. Y sino ocurre ninguno de los eventos mencionados, simplemente mantener una recompensa nula. Note qué de esta forma, la recompensa acumulada

²Disponible en el repositorio de GitHub [link](#).

³Excepto el número de pasos de entrenamiento, para este problema se usaron 80K pasos.

corresponde con el puntaje total del juego, por lo tanto, el objetivo del agente será maximizar este para cada partida.

Hecho esto, se prosiguió con el proceso de entrenamiento del agente, para lo cual, se decidió usar los mismos parámetros del problema anterior ³, es decir, usar el algoritmo A2C con el modelo ‘MlpPolicy’. Este proceso no tuvo un buen desempeño, como se puede ver en la gráfica superior de la Figura 4, la recompensa promedio, aunque tiene una tendencia creciente en el transcurso del entrenamiento, el incremento total es casi nada, en términos del juego, paso de golpear la pelota 1 vez en 20 episodios, a golpearla unas 2 o 3 veces. Al final del entrenamiento, se pudo notar que el agente efectivamente estableció una regla de decisión para el entorno, sin embargo, está no era buena, básicamente aprendió a moverse en una sola dirección hasta llegar a un extremo.

Ante este fallo en el proceso, se decide modificar el entorno para implementar un nuevo sistema de recompensa, uno que tal vez ayude al agente a alcanzar el objetivo. Para ello se estableció como nuevo sistema premiar al agente si p_{p1} estaba en el rango de no fallo para golpear la pelota, es decir si no se alejaba mucho de p_{by} . De esta forma, se retornaba $+1$ si cumplía la condición y 0 en caso contrario, además, se producía -1 si se perdía la partida al igual que antes. Con esto, se distinguen entonces los entornos 1 y 2 cada una diferenciado por su función de recompensa.

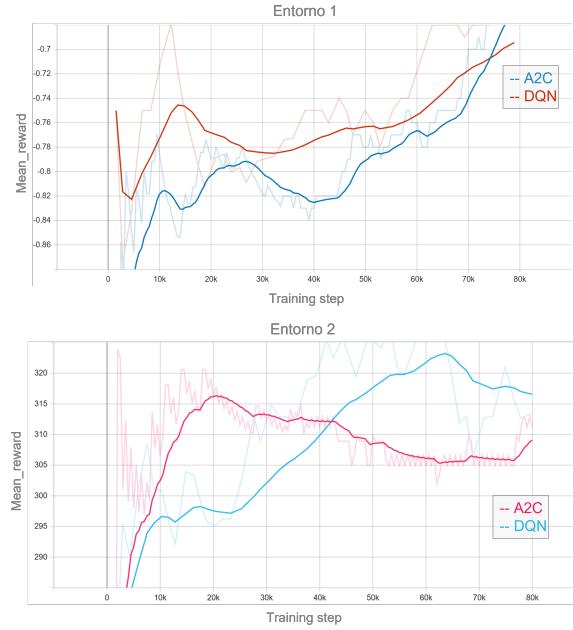


Figura 4: Gráficas generadas con Tensorboard de la recompensa promedio en el proceso de entrenamiento. Superior: versión 1 del entorno Pong, Inferior: versión 2 del entorno Pong.

Hecho esto, se desarrolló el proceso de entrenamiento, pero el resultado no fue mejor como se puede ver en la gráfica inferior de la Figura 4. Por esta razón, se intentó hacer un cambio de algoritmo y en lugar de usar A2C utilizar DQN, y, aunque en términos generales parece ser una

mejor opción porque se evidencia en el juego que permite al agente realizar rangos de movimiento de la paleta más amplios, el resultado final no es muy distinto, en ambos casos no se lograron partidas con puntajes superiores a 3 a lo sumo.

Algoritmo	Estado del Agente	Entorno 1	Entorno 2
A2C	No entrenado	-0.9 ± 0.1	236 ± 16
	Entrenado	-0.6 ± 0.1	271 ± 22
DQN	No entrenado	-0.4 ± 0.2	387 ± 58
	Entrenado	-0.6 ± 0.1	331 ± 36

Cuadro 1: Recompensa promedio del agente en 20 episodios antes y después de ser entrenado, para dos algoritmos y dos entornos diferentes.

En la tabla 1, se captura el valor de la recompensa acumulada promedio en 20 episodio para cada una de las implementaciones que se describieron. Allí se puede ver que, que el agente no aprende de la forma esperada, de Hecho, visualmente se pudo testear que el comportamiento de agente luego del entrenamiento tenía a ir en una sola dirección como se menciono antes, y solo de forma azarosa golpear la pelota en algunas ocasiones.

5. Conclusiones

Aunque el resultado no fue el esperado, se puedo mostrar un procedimiento metodológico que permite implementar proyectos de RL personalizados con el uso de los paquetes basados en OpenAI, lo cual facilita mucho la programación de estos algoritmos de ML.

Por cuestión de tiempo, no fue posible experimentar mucho con la implementación del juego de Pong, de for-

ma que se lograra un entrenamiento exitoso, por lo cual, a continuación, se presentan algunas sugerencias para una futura implementación:

- Aunque el sistema de recompensas se logró modificar, el resultado final no cambio significativamente, por lo que dejando de un lado esto, se podría tratar de cambiar el tipo de observable que se genera, dejar de usar las variables físicas que se mencionaron y pasar a usar la imagen de las partidas como se suele hacer. La razón de esto es porque tal vez las variables que se utilizaron como descriptores del sistema no capturan con suficiente precisión las características del problema, por tanto, sería una buena opción extraer éstas con ayuda de redes convolucionales que actúen sobre la imagen de los juegos.
- Para este proyecto no se pudo profundizar demasiado en los algoritmos que se usan en el entrenamiento, por tanto, podría tratar de identificarse si los modelos usados eran adecuados o no para el problema y cuáles otros podrían usarse [10]. Acá se hizo una selección basada en el tipo de espacio de acciones del problema, el cual era un espacio discreto, por lo que, según la documentación de gym [7], había la opción de usar A2C, DQN, HER o PPO.
- Por último, aunque **Stable-Baselines** ofrece un paquete muy completo para la implementación de algoritmos de RL, no es el único, por tanto, podría tratar de usarse otros paquetes como **Keras-rl**, en donde se debe implementar la arquitectura de las redes capa por capa, lo cual da un control sobre los hiperparámetros más claro y por tanto una posibilidad de experimentación en el proceso de entrenamiento hasta alcanzar un resultado óptimo.

Referencias

- [1] AurelianTactics. *Understanding PPO plots in TensorBoard*. dic de 2018. URL: <https://medium.com/aureliantactics/understanding-ppo-plots-in-tensorboard-cbc3199b9ba2>.
- [2] Andrew G. Barto, Richard S. Sutton y Charles W. Anderson. “Neuronlike adaptive elements that can solve difficult learning control problems”. En: *IEEE Transactions on Systems, Man, and Cybernetics SMC-13.5* (1983), págs. 834-846. doi: [10.1109/TSMC.1983.6313077](https://doi.org/10.1109/TSMC.1983.6313077).
- [3] Wikipedia contributors. *Reinforcement learning*. Nov. de 2021. URL: https://en.wikipedia.org/w/index.php?title=Reinforcement_learning&oldid=1056129752.
- [4] Peter Henderson y col. “Deep Reinforcement Learning that Matters”. 2017. URL: [http://arxiv.org/abs/1709.06560](https://arxiv.org/abs/1709.06560).
- [5] Sergey Levine. *Deep Reinforcement Learning, Lecture4: Introduction to Reinforcement Learning*. 2021. URL: <http://rail.eecs.berkeley.edu/deeprlcourse/>.
- [6] Marlos C. Machado y col. “Revisiting the Arcade Learning Environment: Evaluation protocols and open problems for general agents”. 2017. URL: [http://arxiv.org/abs/1709.06009](https://arxiv.org/abs/1709.06009).
- [7] OpenAI. *Gym: A toolkit for developing and comparing reinforcement learning algorithms*. URL: <https://gym.openai.com/docs/>.
- [8] David Silver. *UCL Course on RL , Lecture 1: Introduction to Reinforcement Learning*. 2015. URL: <https://www.davidsilver.uk/teaching/>.
- [9] *Stable-Baselines3 docs - reliable reinforcement learning implementations — stable Baselines3 1.3.1a6 documentation*. URL: <https://stable-baselines3.readthedocs.io/en/master/index.html>.

- [10] Ujwal Tewari. *Which Reinforcement learning-RL algorithm to use where, when and in what scenario?* abr de 2020. URL: <https://medium.datadriveninvestor.com/which-reinforcement-learning-rl-algorithm-to-use-where-when-and-in-what-scenario-e3e7617fb0b1>.