# Playing Atari with Deep Reinforcement Learning

**Abstract**

Primer modelo de DL que exitosamente aprendió políticas de control tomando inputs sensoriales de alta dimensión y utilizando aprendizaje por refuerzo.

El modelo es una CNN, entrenada con una variante de Q-learning cuyo input son pixeles en bruto y cuyo output es una función de valor que estima recompensas futuras.

**Introducción**

Aprender a controlar agentes directamente con inputs sensoriales de alta dimensión, como la visión o el lenguaje, es uno de los desafíos más grandes de RL. La mayoría de las aplicaciones exitosas que operan en estos dominios requieren features diseñadas a mano combinadas con funciones de valor lineal o representaciones de políticas. Por ende, el desempeño de esos algoritmos depende de que tan buenas sean esas features.

El RL presenta varios desafíos desde una perspectiva de Deep Learning. En primer lugar, las aplicaciones de aprendizaje profundo más exitosas hasta la fecha han requerido grandes cantidades de datos de entrenamiento etiquetados manualmente. Los algoritmos RL, por otro lado, deben poder aprender de una señal de recompensa escalar que frecuentemente es escasa, ruidosa y retrasada. El retraso entre las acciones y las recompensas resultantes, que pueden durar miles de pasos, parece particularmente desalentador en comparación con la asociación directa entre los insumos y los objetivos que se encuentra en el aprendizaje supervisado. Otro problema es que la mayoría de los algoritmos de aprendizaje profundo asumen que las muestras de datos son independientes, mientras que en el aprendizaje por refuerzo normalmente se encuentran secuencias de estados altamente correlacionados. Además, en RL la distribución de datos cambia a medida que el algoritmo aprende nuevos comportamientos, lo que puede resultar problemático para los métodos de aprendizaje profundo que asumen una distribución subyacente fija.

Este artículo demuestra que una CNN puede superar estos desafíos, aprendiendo políticas de control exitosas a partir de datos de video sin procesar en entornos RL complejos. La red se entrena con una variante del algoritmo Q-learning, con descenso de gradiente estocástico para actualizar los pesos.

Para aliviar los problemas de los datos correlacionados y distribuciones no estacionarias, utilizamos un mecanismo de repetición de experiencias que muestra aleatoriamente transiciones anteriores y, por lo tanto, suaviza la distribución del entrenamiento sobre muchos comportamientos pasados.

El objetivo es la creación de una única red neuronal que sea capaz de aprender exitosamente a jugar la mayor cantidad de juegos posibles. La red no recibió ninguna información específica del juego ni features visuales diseñadas a mano, ni tampoco tenía conocimiento del estado interno del emulador, en otras palabras, no aprendía nada más que la entrada de vídeo, las señales terminales y de recompensa, y el conjunto de acciones posibles, tal como lo haría un jugador humano.

Además, la arquitectura de la red y todos los hiperparámetros utilizados para el entrenamiento se mantuvieron constantes durante todos los juegos.

**Background**

Consideramos tareas en las que el agente interactúa con un ambiente ξ (emulador Atari), en una secuencia de acciones, observaciones y recompensas. En cada time-step el agente selecciona una acción *at*, del conjunto de acciones posibles A= {1, …, K}. La acción se le pasa al emulador y modifica su estado interior y puntaje. En general, el ambiente ξ suele ser estocástico. El estado interno del emulador no es observado por el agente, pero sí observa una imagen xt ∈ Rd, que es un vector de pixeles brutos que representa el estado actual de la pantalla. En consecuencia, recibe una recompensa rt que representa el cambio en el puntaje del juego. Notar que en general el puntaje del juego puede depender de todo el vector de acciones y observaciones, por lo que el feedback de la acción realizada podría venir recién luego de miles de time-steps.

Dado que el agente solamente observa la imagen de la pantalla, la tarea es parcialmente observada dado que una imagen sola no permite obtener el contexto del estado del juego. Es por este motivo que se consideran secuencias de acciones y observaciones, st = x1, a1, x2, …, x t-1, at-1, xt sobre las que el agente aprenderá estrategias.

Este formalismo da lugar a un amplio, pero finito MDP en donde cada secuencia es un estado diferente. Como resultado, pueden aplicarse métodos estándar de RL para MDPs, simplemente utilizando la secuencia completa st como representación del estado t.

El objetivo del agente es interactuar con el emulador seleccionando acciones de forma de maximizar recompensas futuras. Hacemos el supuesto estándar de que las recompensas futuras vienen descontadas por un factor Ύ por time-step, definiendo el retorno futuro descontado al momento t, como:

donde T es el time-step final del episodio

También definimos el valor óptimo de la función action-value Q\*(s,a) como el retorno máximo esperado alcanzable por cualquier estrategia, luego de ver alguna secuencia s y tomado la acción a

donde π es una política que mapea secuencias a acciones

La función optima action-value obedece a la identidad de Bellman, la cual dice que si el valor optimo Q\*(s’,a’) de la secuencia s’ en el siguiente time-step fuera conocida para todos los valores de a’, entonces la estrategia optima seria seleccionar la acción a’ que maximice el valor esperado de: r + ΎQ\*(s’, a’)

(1)

En la práctica, lo que se hace es utilizar un aproximador a dicha función para estimar la función de action-value:

Q(s,a: θ) ≈ Q\*(s,a)

En la comunidad de RL, el aproximador suele ser una función lineal, pero cuando se utilizan redes neuronales se suele utilizar una no lineal. Es entonces que en redes neuronales nos referimos a redes Q-network con weights θ. Esta red puede ser entrenada minimizando una secuencia de funciones de perdida Li(θi) que cambia en cada iteración i:

(2)

Donde es el objetivo de la iteración i, y p(s,a) es una distribución de probabilidades sobre secuencias s y acciones a, a la que nos referimos como p: behaviour distribution.

Los parámetros de la iteración previa θi-1 se mantienen fijas mientas se optimiza la función de perdida Li(θi). Los targets entonces dependen de los pesos de la red (en contraste con los targets utilizados en aprendizaje supervisado, que son fijos antes de que la red comience a entrenar).

El gradiente lo obtenemos al diferenciar la función de perdida respecto a los weights.



Para ser eficiente computacionalmente se suele optimizar la función de perdida por descenso de gradiente estocástico.

El algoritmo resultante será entonces:

(i) model free

(ii) off-policy

En la práctica, la behaviour distribution se selecciona con estrategia ɛ-greedy (1- ɛ elije acción greedy, ɛ elije acción aleatoria).

**Related work**

El trabajo anterior mas cercano a este paper es el que habla sobre Neural Fitted Q-Learning (NFQ), el cual optimiza la secuencia de funciones de perdida, utilizando el algoritmo RPROP para actualizar los parámetros de la Q-Network. A su vez, utiliza actualización por batch (conjunto de datos, en lugar de actualizar dato a dato) de tamaño proporcional al tamaño del dataset.

En este paper, al utilizar RL de principio a fin, directamente de las imágenes de input, permite que algoritmo pueda aprender features que son relevantes para discriminar action-values.

**Deep Reinforcement Learning**

El objetivo es conectar el algoritmo de RL a una red de DL que opere sobre imágenes RGB y procese eficientemente datos de entrenamiento utilizando actualizaciones de gradiente estocástico.

El punto de inicio es el TD-Gammon de Tesauro.

Se utiliza una técnica conocida como experience replay, donde se almacena la experiencia del agente en cada time-step, et = (st, at, rt, st+1), en un dataset D = e1, …, eN, a lo largo de muchos episodios, generando una replay memory.

Durante el loop del algoritmo, se aplican actualizaciones de Q-Learning, o minibatch, sobre muestras aleatorias de esa experiencia.

Ejemplo: Imaginar que un agente está aprendiendo a jugar un videojuego:

En lugar de ajustar su estrategia inmediatamente después de cada movimiento (como lo haría en un enfoque en línea), el estado del juego en ese momento, la acción que tomó, la recompensa que recibió y el estado siguiente. Después de jugar varias partidas, tiene una gran colección de estas experiencias.

Al entrenar, selecciona aleatoriamente experiencias de esta colección para ajustar sus estrategias, aprovechando mejor cada experiencia individual.

Después de realizar experience replay, el agente selecciona y ejecuta una acción de acuerdo a una política e-greedy. En este algoritmo, la Q-function trabaja directamente sobre representaciones de largo fijo de historias producidas por la función φ. El algoritmo completo recibe el nombre de Deep Q-Learning.

Este enfoque tiene varias ventajas:

1. Cada paso de experiencia puede llegar a utilizarse en varias actualizaciones de pesos, lo que aumenta la eficiencia de los datos
2. Aprender sobre muestras consecutivas es ineficiente, debido a la fuerte correlación entre muestras, por lo que tomarlas de manera aleatoria rompe esa correlación y por ende reduce la varianza de las actualizaciones.
3. En esto contexto, utilizar on-policy para aprender implica que los parámetros actuales determinan el próximo set de datos sobre los que se entrenara el modelo.

Ejemplo: Imaginar un robot que está aprendiendo a encontrar un tesoro en un campo. El robot puede moverse hacia la izquierda o hacia la derecha.

Primera Fase:

El robot comienza con una política que le dice que moverse a la izquierda es la mejor opción porque encontró algunos premios allí. La mayoría de sus movimientos serán hacia la izquierda y la mayoría de sus datos de entrenamiento provendrán de esa área.

Cambio de Parámetros:

Supongamos que el robot ahora encuentra un premio mayor en el lado derecho.

Cambia su política para moverse hacia la derecha, y ahora todos sus movimientos y datos de entrenamiento provienen del lado derecho.

Bucles de Retroalimentación:

Si el robot siempre cambia su política basada en la última mejor recompensa sin explorar adecuadamente, puede quedarse atascado:

Mínimo Local: Podría quedarse buscando en un área pequeña del campo donde solo encuentra premios pequeños, sin darse cuenta de que hay premios mayores en otra parte.

Divergencia: Si sigue cambiando su política drásticamente, puede terminar moviéndose de manera errática, perdiendo el enfoque y no encontrando ningún premio.

Solución Potencial:

Experience Replay: Almacenar y reutilizar experiencias pasadas para asegurarse de que el agente aprende de una variedad de datos, no solo de los datos más recientes.

Exploración Adecuada: Asegurarse de que el agente explore diferentes áreas y no solo explote las acciones que parecen mejores en el corto plazo.

Al utilizar experience replay, la behaviour distribution es un promedio sobre muchos de sus estados previos, suavizando el aprendizaje y evitando oscilaciones o divergencias en los parámetros.

Es necesario notar que al aprender por experience replay, es necesario aprender off-policy (porque los parámetros actuales son diferentes de aquellos utilizados para generar la muestra).

En la práctica, el algoritmo solo guarda las ultimas N tuplas de experiencia en replay memory, y muestrea uniformemente sobre D para realizar actualizaciones. Este acercamiento implica restricciones dado que el buffer de memoria no es infinito y por ende termina sobrescribiendo transiciones antiguas con transiciones recientes, pudiendo perder información valiosa en el proceso. A su vez, el muestreo uniforme le da la misma importancia a cada experiencia en replay memory (podría utilizarse prioritized sweeping).

**Preprocesamiento y Arquitectura del modelo**

Trabajar directamente con frames de Atari, de 210x160 pxs, y paleta de 128 colores, puede ser demandante computacionalmente, por lo que se procesan los inputs a fin de reducir su dimensionalidad:

1. Pasar los frames de RGB a escala de grises
2. Reducir el tamaño a frames 110x84pxs
3. Recortarlos a un tamaño de 84x84pxs que contemple toda el área de juego (esto es porque se utiliza implementación GPU de convoluciones 2D, la cual espera inputs cuadrados

En este paper, la función φ aplica este preprocesamiento a los últimos 4 frames de la historia y los apila para producir el input de la Q-function.

En resumen, la arquitectura de este algoritmo es siempre la misma:

* Input: imágenes 84x84x4 producidas por la función φ.
* Primera capa oculta: 16 convoluciones de 8x8 y stride=4 sobre la que aplica reLU.
* Segunda capa oculta: 32 convoluciones de 4x4 y stride=2, sobre la que se aplica reLU.
* Ultima capa oculta: Fully-connected con 256 neuronas.
* Capa de salida: Fully-connected con tantas neuronas como acciones posibles (entre 4 y 18 dependiendo el juego).

**Experimentos**

Para los 7 juegos sobre los que se probó el modelo, la arquitectura y el algoritmo se mantenían en incambiados, aunque fue necesario hacerle un ajuste ya que la escala de la puntuación entre ellos variaba bastante. De esta manera

* Recompensas positivas = +1
* Recompensas negativas = -1
* Sin recompensas = 0

En estos experimentos, se utilizó el algoritmo RMSProp con minibatches de tamaño 32. La política de comportamiento durante el entrenamiento fue e-greedy, con e disminuyendo linealmente de 1 a 0.1 durante el primer millón de frames, y fijándose en 0.1 a partir de entonces. Se entrenó durante un total de 10 millones de frames y se utilizó un replay memory de un millón de frames.

También se utilizó una técnica simple de frame-skipping, donde el agente observa y selecciona acciones en cada k-ésimo frame en lugar de en cada uno, repitiendo su última acción sobre los frames omitidos. Dado que avanzar el emulador un paso requiere mucho menos cálculo que hacer que el agente seleccione una acción, esta técnica permite que el agente juegue aproximadamente k veces más juegos sin aumentar significativamente el tiempo de ejecución. Usamos k=4 para todos los juegos excepto Space Invaders, donde notamos que usar k=4 hace que los láseres sean invisibles debido al período en el que parpadean. Usamos 𝑘=3 para hacer los láseres visibles y este cambio fue la única diferencia en los valores de hiperparámetros entre cualquiera de los juegos.

**Entrenamiento y Estabilidad**

Evaluar acertadamente el progreso de un agente puede ser desafiante en RL. Dado que nuestra métrica de evaluación, es la recompensa total que el agente recoge en un episodio o juego promediada sobre varios juegos, la calculamos periódicamente durante el entrenamiento. La métrica de recompensa total promedio tiende a ser muy ruidosa porque pequeños cambios en los pesos de una política pueden llevar a grandes cambios en la distribución de estados que la política visita. Otra métrica, más estable, es la función de valor de acción estimada por la política, Q, proporciona una estimación de cuánta recompensa descontada puede obtener el agente al seguir su política desde cualquier estado dado. Recogemos un conjunto fijo de estados ejecutando una política aleatoria antes de que comience el entrenamiento y rastreamos el promedio del máximo Q predicho para estos estados. Además de observar una mejora relativamente suave en el Q predicho durante el entrenamiento, no experimentamos problemas de divergencia en ninguno de nuestros experimentos. Esto sugiere que, a pesar de no tener garantías teóricas de convergencia, nuestro método es capaz de entrenar grandes redes neuronales utilizando una señal de aprendizaje por refuerzo y descenso de gradiente estocástico de manera estable