Tema 5: Aprendizaje por Refuerzo

AlphaGo, AlphaGo Zero, AlphaZero, MuZero

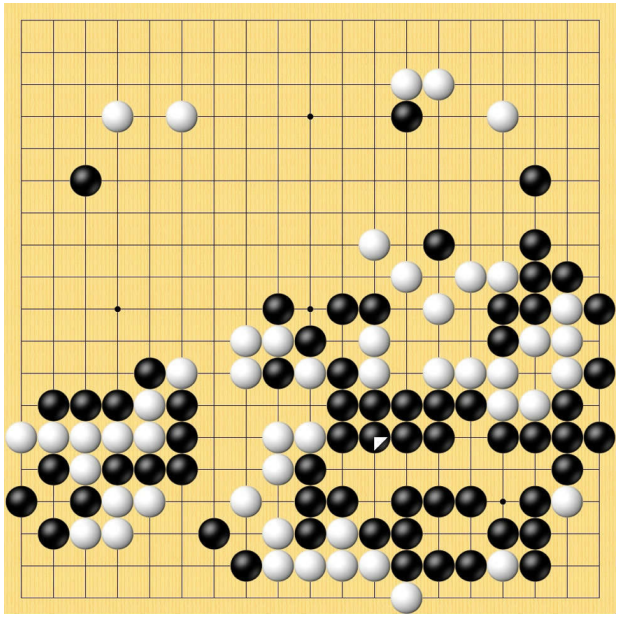
Prof. Wladimir Rodriguez

wladimir@ula.ve

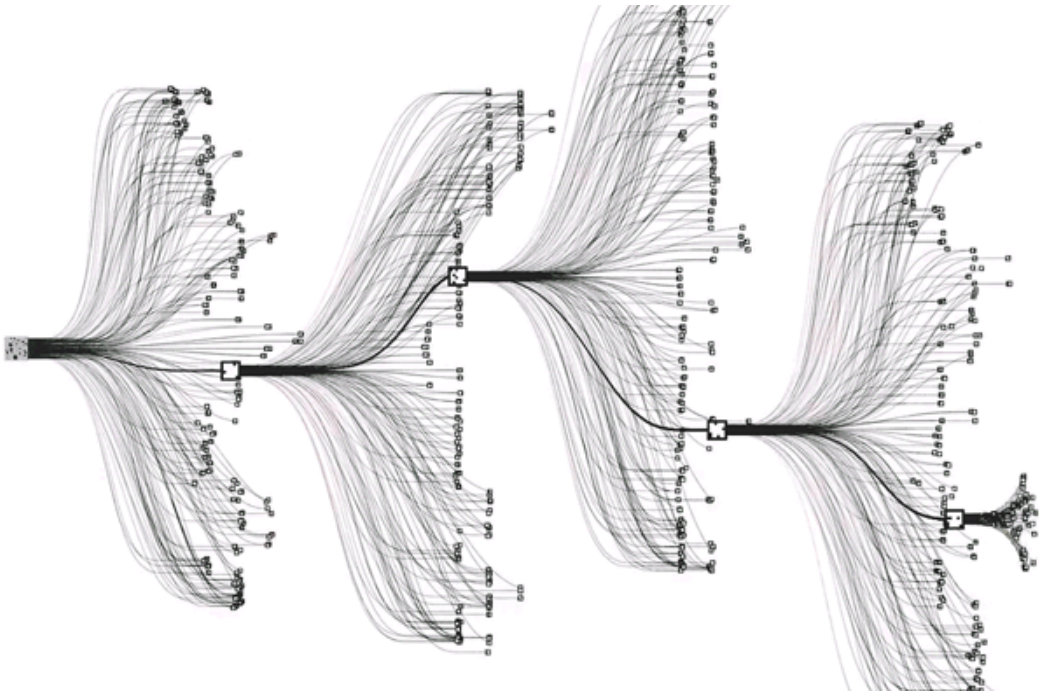
Departamento de Computación

El juego de Go

Go se originó en China hace más de 3.000 años. Ganar este juego de mesa requiere múltiples capas de pensamiento estratégico. Dos jugadores, con piedras blancas o negras, se turnan para colocar sus piedras en un tablero. El objetivo es rodear y capturar las piedras de su oponente o crear estratégicamente espacios de territorio. Una vez que se han jugado todos los movimientos posibles, se cuentan tanto las piedras en el tablero como los puntos vacíos. El número más alto gana.



Por simples que parezcan las reglas, Go es profundamente complejo. Hay un asombroso 10 a la potencia de 170 posibles configuraciones del tablero, más que la cantidad de átomos en el universo conocido. Esto hace que el juego de Go sea un googol veces más complejo que el ajedrez.



AlphaGo

AlphaGo es un programa de computadora que juega el juego de mesa Go. Fue desarrollado por DeepMind Technologies, una subsidiaria de Google (ahora Alphabet Inc.).

AlphaGo es el primer programa de computadora en derrotar a un jugador humano profesional de Go, el primero en derrotar a un campeón mundial de Go, y posiblemente sea el jugador de Go más fuerte de la historia.

Las versiones posteriores de AlphaGo se volvieron cada vez más potentes, incluida una versión que compitió con el nombre AlphaGo Zero, que fue completamente autodidacta sin aprender de los juegos humanos. AlphaGo Zero luego se generalizó en un programa conocido como AlphaZero, que jugaba juegos adicionales, incluidos el ajedrez y el shogi (ajedrez japones). AlphaZero, a su vez, ha sido reemplazado por un programa conocido como MuZero que aprende sin que se le enseñen las reglas.

AlphaGo y sus sucesores utilizan un algoritmo de búsqueda llamado *Monte Carlo Tree Search* (MCTS) para encontrar sus movimientos en función del conocimiento adquirido previamente mediante el aprendizaje automático, específicamente mediante una red neuronal artificial, tanto del juego humano como de la computadora. Se entrena una red neuronal para identificar los mejores movimientos y los porcentajes ganadores de estos movimientos. Esta red neuronal mejora la fuerza de la búsqueda del árbol, lo que da como resultado una selección de movimiento más fuerte en la siguiente iteración.

AlphaGo es un sistema que combina:

- Aprendizaje por imitación (clonación conductual)
- Aprendizaje por refuerzo (juego por cuenta propia)
- Planificación (búsqueda anticipada con MCTS)
- Poder computacional masivo (centro de datos de Google)

Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) es una técnica de búsqueda en el campo de la Inteligencia Artificial (IA). Es un algoritmo de búsqueda heurístico y probabilístico que combina las implementaciones clásicas de búsqueda de árboles junto con los principios de aprendizaje automático del aprendizaje por refuerzo.

En la búsqueda de árbol, siempre existe la posibilidad de que la mejor acción actual no sea en realidad la acción más óptima. En tales casos, el algoritmo MCTS se vuelve útil ya que continúa evaluando otras alternativas periódicamente durante la fase de aprendizaje ejecutándolas, en lugar de la estrategia óptima percibida actual. Esto se conoce como el "trade-off de exploración-explotación". Explota las acciones y estrategias que se encuentran como las mejores hasta ahora, pero también debe continuar explorando el espacio local de decisiones alternativas y averiguar si podrían reemplazar a las mejores actuales.

La exploración ayuda a explorar y descubrir las partes inexploradas del árbol, lo que podría resultar en encontrar un camino más óptimo. En otras palabras, podemos decir que la exploración expande el ancho del árbol más que su profundidad. La exploración puede ser útil para garantizar que MCTS no pase por alto ningún camino potencialmente mejor. Pero rápidamente se vuelve ineficiente en situaciones con gran cantidad de pasos o repeticiones. Para evitarlo, se compensa con la explotación. La explotación se apega a un solo camino que tiene el mayor valor estimado. Este es un enfoque codicioso y esto extenderá la profundidad del árbol más que su ancho. En palabras simples, la fórmula UCB aplicada a los árboles ayuda a equilibrar la compensación de exploración-explotación al explorar periódicamente nodos relativamente inexplorados del árbol y descubrir caminos potencialmente más óptimos que el que está explotando actualmente.

Por esta característica, MCTS se vuelve particularmente útil para tomar decisiones óptimas en problemas de Inteligencia Artificial (IA).

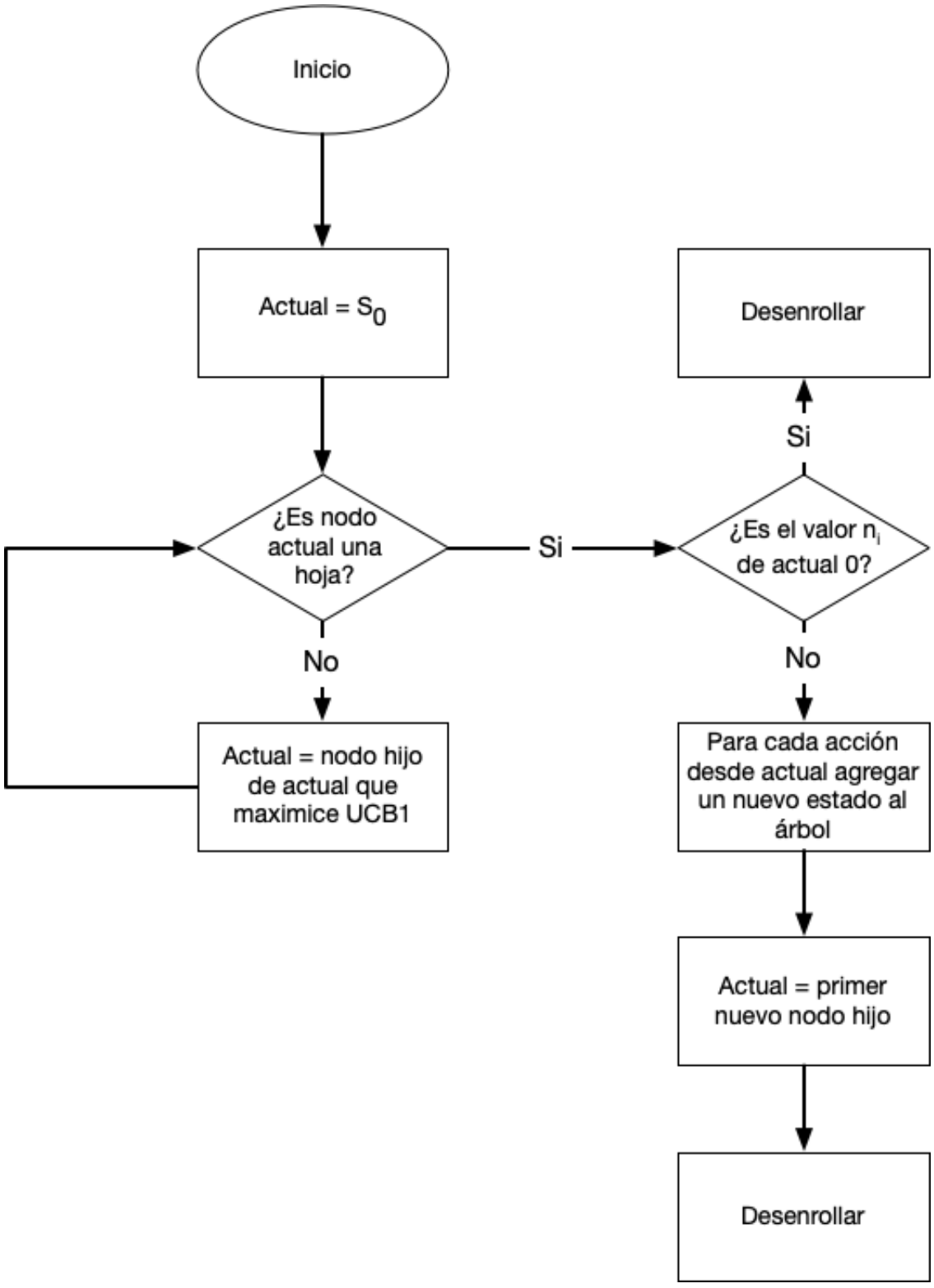
Algoritmo Monte Carlo Tree Search (MCTS):

En MCTS, los nodos son los componentes básicos del árbol de búsqueda. Estos nodos se forman en función del resultado de una serie de simulaciones. El proceso de Monte Carlo Tree Search se puede dividir en cuatro pasos distintos, a saber, selección, expansión, simulación y propagación hacia atrás. Cada uno de estos pasos se explica en detalle a continuación:

- *Selección*: en este proceso, el algoritmo MCTS atraviesa el árbol actual desde el nodo raíz utilizando una estrategia específica. La estrategia utiliza una función de evaluación para seleccionar de manera óptima los nodos con el valor estimado más alto. MCTS utiliza la fórmula *Upper Confidence Bound (UCB)* aplicada a los árboles como estrategia en el proceso de selección para atravesar el árbol. Equilibra el compromiso entre exploración y explotación. Durante el recorrido del árbol, se selecciona un nodo en función de algunos parámetros que devuelven el valor máximo. Los parámetros se caracterizan por la fórmula que se suele utilizar para este fin, que se indica a continuación.

$$UCB = V_i + 2\sqrt{\frac{ln(N)}{n_i}}$$

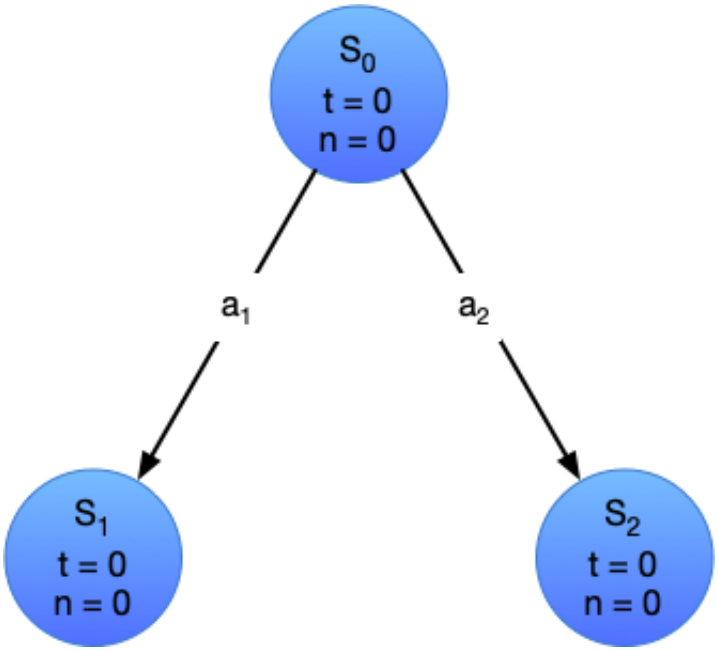
- dónde,
  - V_i es la recompensa/valor promedio de todos los nodos debajo de este nodo
  - N es el número de veces que se ha visitado el nodo principal y
  - n_i es el número de veces que se ha visitado el nodo secundario i
  - Al atravesar un árbol durante el proceso de selección, el nodo secundario que devuelve el mayor valor de la ecuación anterior será el que se seleccionará. Durante el recorrido, una vez que se encuentra un nodo secundario que también es un nodo hoja, el MCTS salta al paso de expansión.
- *Expansión*: en este proceso, se agrega un nuevo nodo secundario al árbol a ese nodo que se alcanzó de manera óptima durante el proceso de selección.
- *Simulación*: En este proceso, se realiza una simulación eligiendo movimientos o estrategias hasta lograr un resultado o estado predefinido.
- *Propagación hacia atrás*: después de determinar el valor del nodo recién agregado, el árbol restante debe actualizarse. Entonces, se realiza el proceso de propagación hacia atrás, donde se propaga desde el nuevo nodo al nodo raíz. Durante el proceso, se incrementa el número de simulaciones almacenadas en cada nodo. Además, si la simulación del nuevo nodo da como resultado una victoria, entonces el número de victorias también se incrementa.



Hagamos un recorrido completo del algoritmo.

- Iteración 1:
  - Partimos de un estado inicial S_0. Aquí tenemos las acciones a_1 y a_2 que

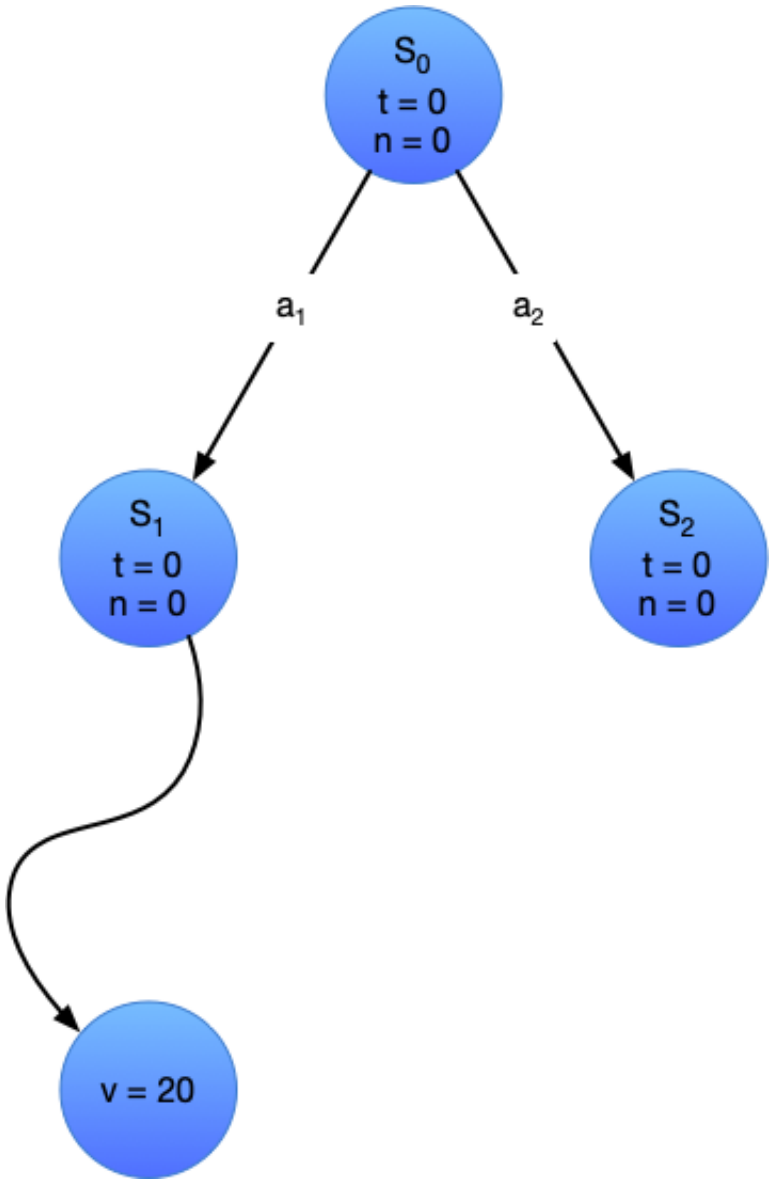
conducen a los estados S_1 y S_2 con puntuación total t y número de visitas n. Pero, ¿cómo elegimos entre los 2 nodos secundarios?



Estado inicial

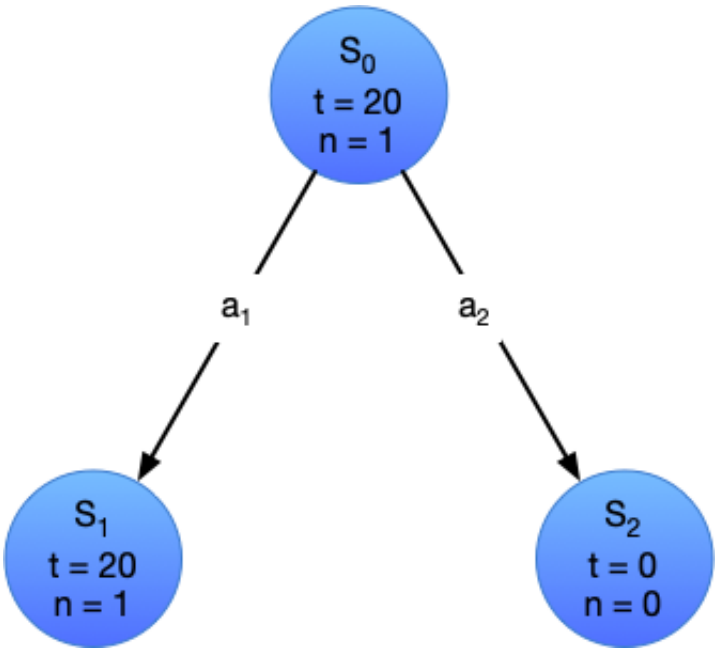
Aquí es donde calculamos los valores UCB para ambos nodos secundarios y tomamos el nodo que maximiza ese valor. Como ninguno de los nodos ha sido visitado todavía, el segundo término es infinito para ambos. Por lo tanto, solo vamos a tomar el primer nodo.

Ahora estamos en un nodo de hoja donde debemos verificar si lo hemos visitado. Resulta que no lo hemos hecho. En este caso, sobre la base del algoritmo, hacemos un despliegue hasta el estado terminal. Digamos que el valor de este desenrollado es 20

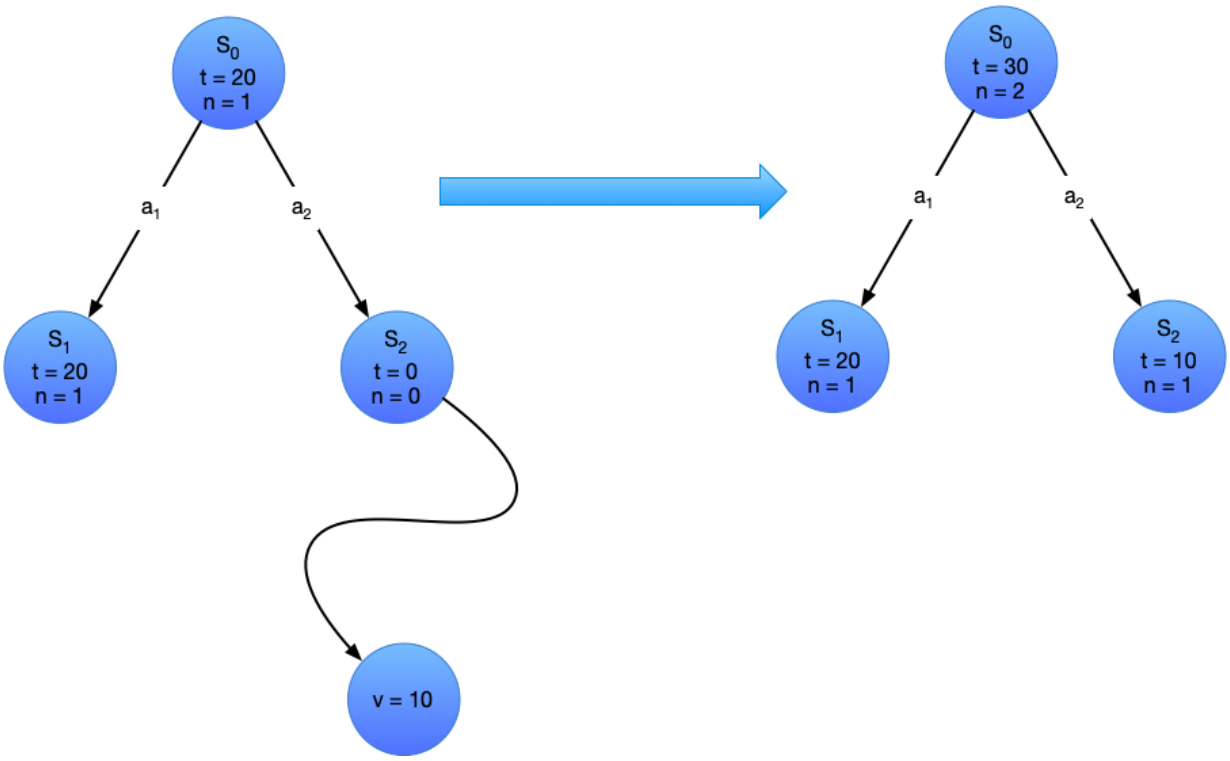


Desenrollado desde S_1

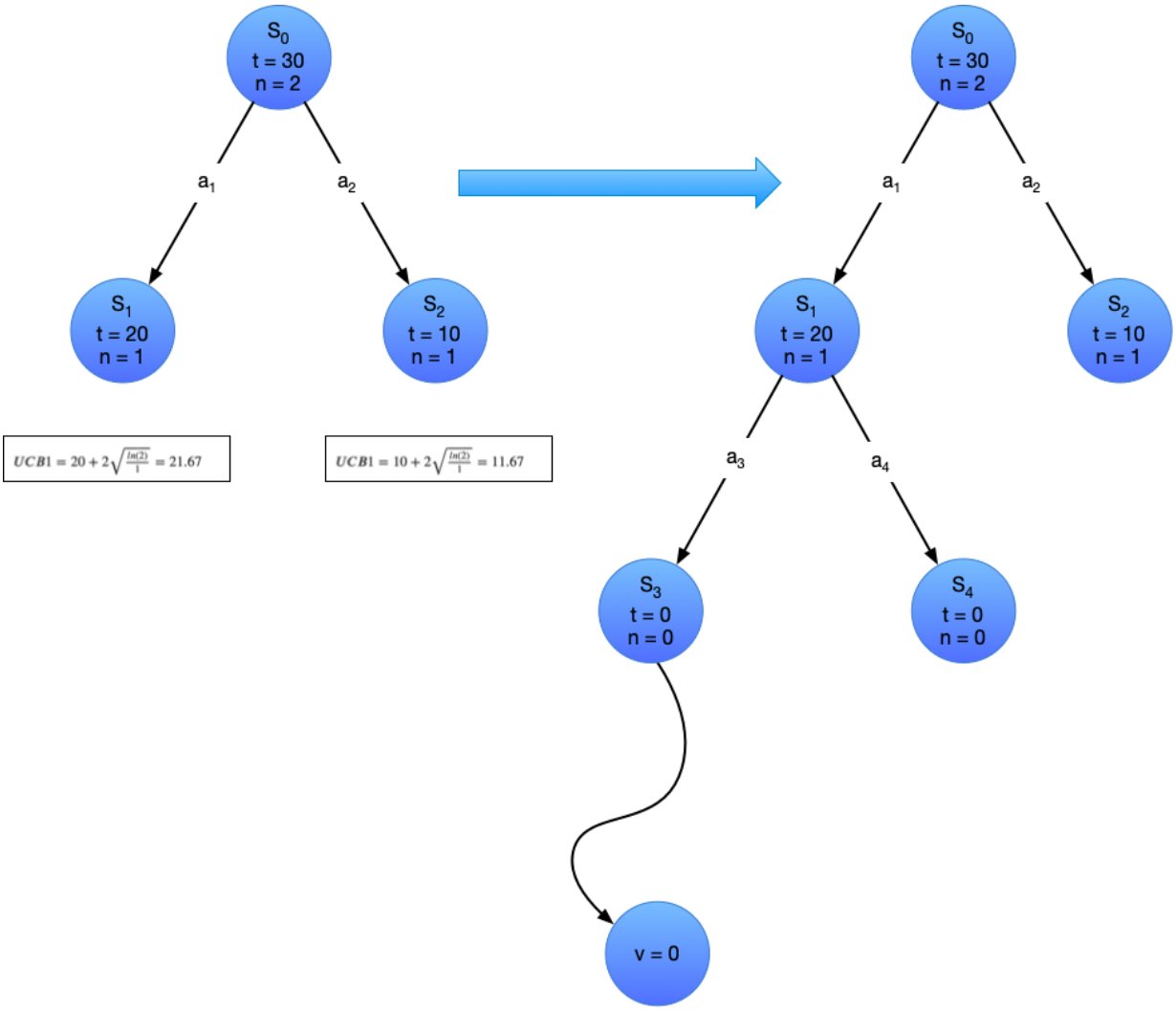
Ahora viene la 4ª fase, o la fase de propagación hacia atrás. El valor del nodo hoja (20) se propaga hacia atrás hasta el nodo raíz. Ahora, $t = 20$ y $n = 1$ para los nodos S_1 y S_0



- Iteración 2:
  - Volvemos al estado inicial y preguntamos qué nodo secundario visitar a continuación. Una vez más, calculamos los valores UCB, que serán $20 + 2 * \sqrt{ln(1)/1} = 20$ para S_1 e infinito para S_2. Como S_2 tiene el valor más alto, elegiremos ese nodo
  - El despliegue se realizará en S_2 para llegar al valor 10, que se propagará hacia atrás al nodo raíz. El valor en el nodo raíz ahora es 30

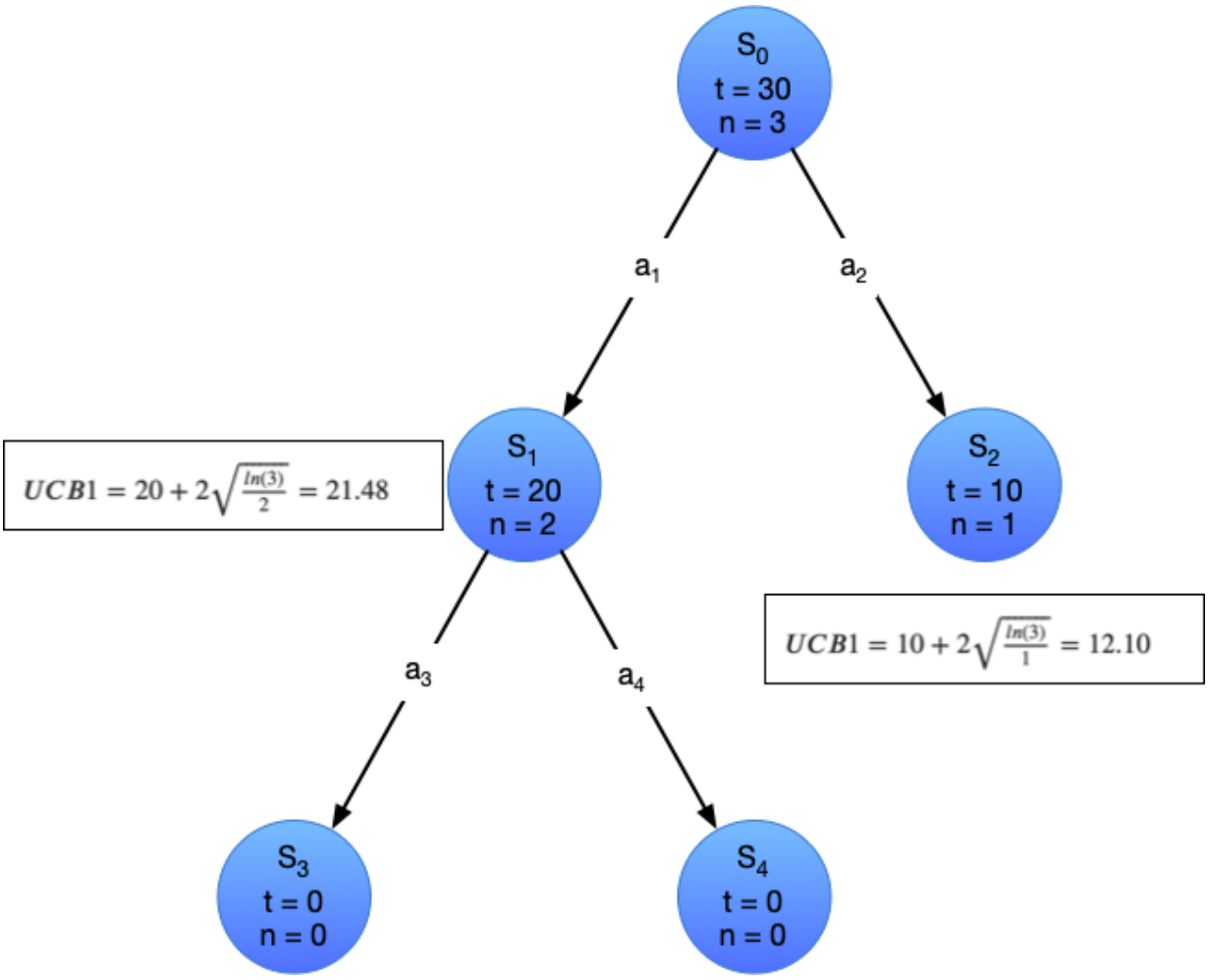


- Iteración 3:
  - En el siguiente diagrama, S_1 tiene un valor UCB1 más alto y, por lo tanto, la expansión debe realizarse aquí:



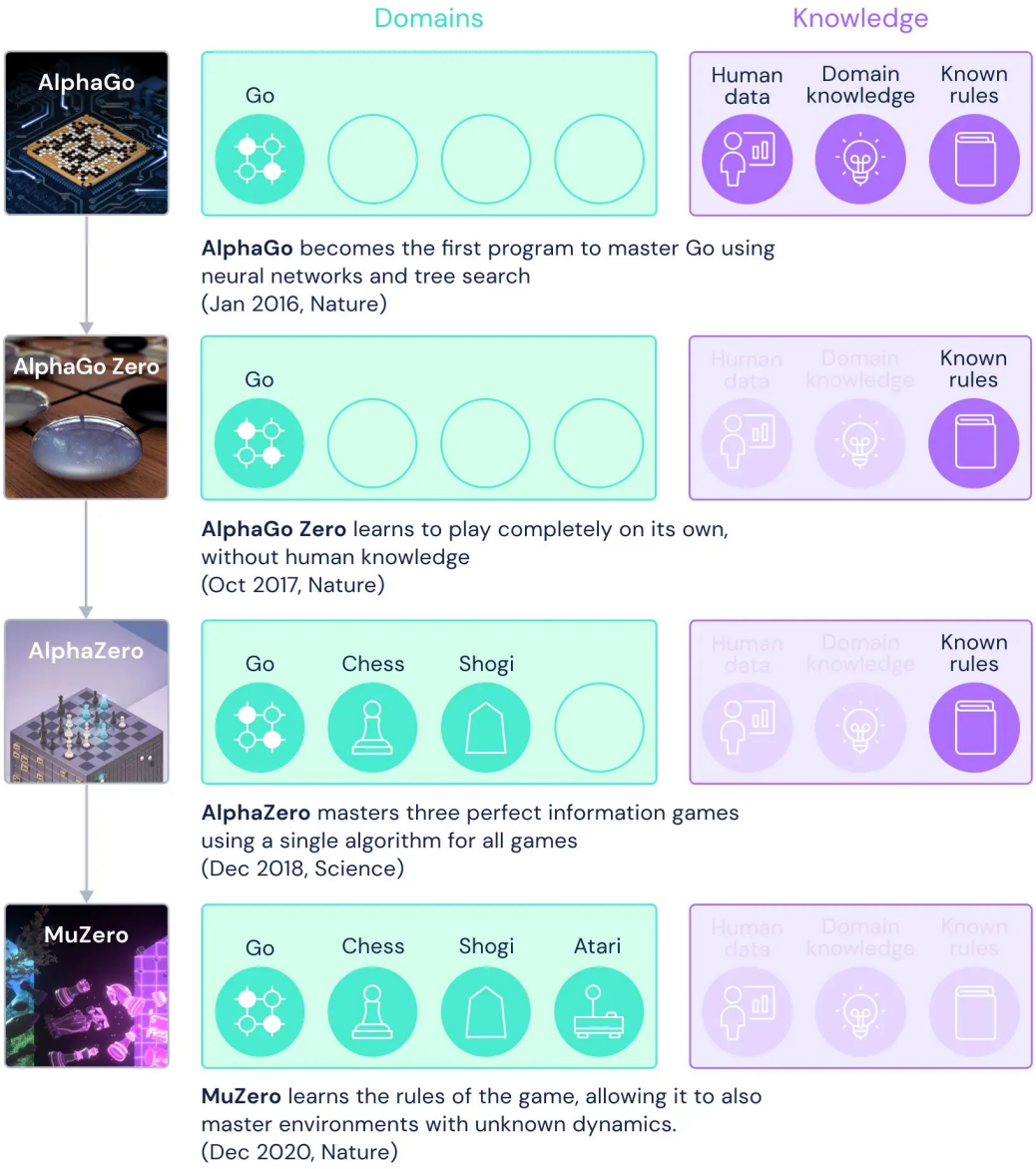

```
- Ahora en $S_1$, estamos exactamente en la misma posición que el estado inicial con los valores de UCB1 para ambos nodos como infinitos. Hacemos un despliegue desde $S_3$ y terminamos obteniendo un valor de 0 en el nodo hoja
```

- Iteración 4:
  - Nuevamente tenemos que elegir entre S_1 y S_2. El valor UCB para S_1 resulta ser 21,48 y 12,10 para S_2:

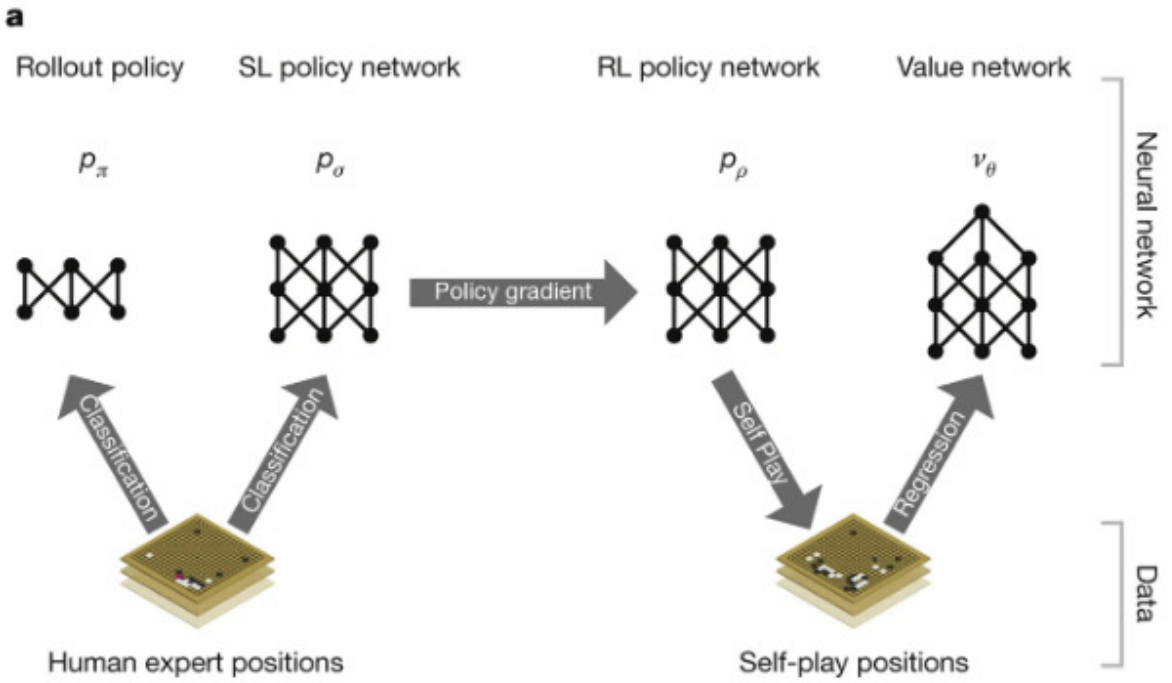


Evolución de AlphaGo a MuZero

- *AlphaGo*: Monte Carlo Tree Search utilizando 3 redes convolucionales de políticas, 2 de las cuales están entrenadas para copiar movimientos expertos, otra con los gradientes de políticas y una red de valor separada.
- *AlphaGo Zero*: sin aprendizaje supervisado de movimientos de expertos, redes de políticas y de valor combinadas en una única red neuronal residual, La Red de Política se actualiza para que coincida con las acciones de MCTS
- *AlphaZero*: reestructura las representaciones de entrada y salida para jugar ajedrez y shogi también, reestructura el algoritmo de juego automático
- *MuZero*: elimina la suposición de un modelo dinámico dado, introduce un estado oculto para hacer MCTS con un modelo dinámico aprendido, a partir de una inicialización del estado raíz aprendido



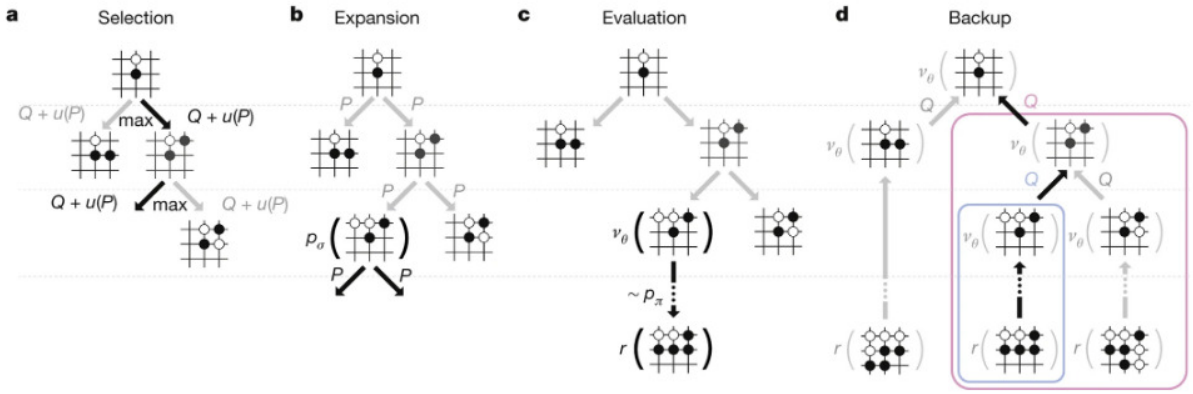
Algoritmo de AlphaGo



AlphaGo utiliza varias redes para aprender a predecir el próximo movimiento a realizar. Primero, una pequeña política de implementación p_π y una política más grande p_σ se entrenan a través del aprendizaje supervisado para predecir el ganador del juego en función de las representaciones estatales. Aunque p_π solo logra ~24% de precisión y p_σ ~57%, siguen siendo útiles en pasos posteriores. Una vez entrenadas ambas políticas, p_σ se utiliza para inicializar p_ρ , que se entrena a través del autojuego. Marcos de p_ρ luego se utilizan para entrenar v_θ, que aprende una función de valor para los estados.

$$v_\theta(s) = \mathbb{E}[z_t | s_t = s, a_{t..T} \sim p_\rho]$$

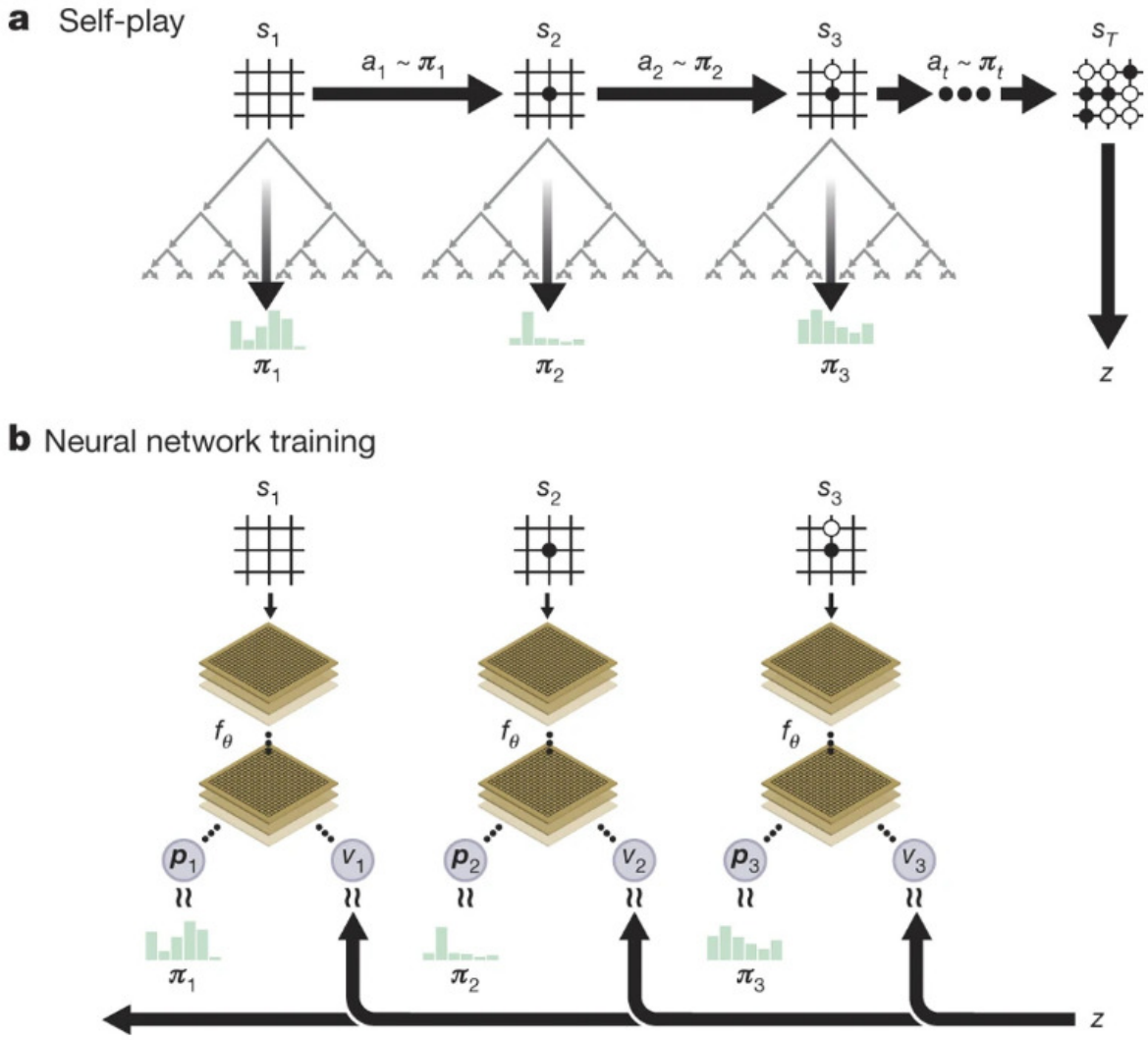
siendo z el resultado del juego, ± 1 dependiendo del jugador. v_θ fue entrenado con una colección de 30 millones de estados de juego, cada uno de un juego diferente jugado por versiones de p_ρ para proporcionar datos etiquetados no correlacionados, reduciendo así el sobreajuste.



Una vez que finaliza el entrenamiento, se utiliza MCTS para jugar.

Algoritmo de AlphaGo Zero

En un artículo posterior, Deepmind presentó AlphaGo Zero, que logra un mayor rendimiento que AlphaGo utilizando menos conocimiento previo. Solo necesita las reglas de juego.



Hay que tener en cuenta un par de diferencias:

- No se utiliza aprendizaje supervisado para inicializar políticas
- Se utiliza una sola política
- Se utiliza una sola red tanto para la política como para la función de valor.
- La red se alimenta de configuraciones del tablero sin procesar en lugar de características hechas a mano.

Esta nueva versión del algoritmo toma la forma

$$p, v = f_\theta(s)$$

, lo que significa que la red genera tanto las probabilidades de acción p como la estimación del valor v para los estados s. Durante el entrenamiento, en cada movimiento, la red realiza simulaciones MCTS

AlfaZero

Todavía en una búsqueda para eliminar conocimiento previo, Deepmind luego lanzó AlphaZero. AlphaZero mantiene el mismo modelo y proceso de entrenamiento general que AlphaGo Zero, pero elimina algunos componentes que no se traducen bien a otros juegos. Los cambios notables incluyen:

- Eliminación del aumento de datos (rotar o invertir tableros) que realizó AlphaGo Zero porque crearía configuraciones imposibles para Ajedrez y Shogi
- Los opositores al autojuego ahora son simplemente la misma red neuronal exacta en lugar de una lista de versiones anteriores del algoritmo.
- AlphaZero reutiliza los mismos hiperparámetros excepto uno (la exploración depende del número de movimientos válidos) para todos los juegos

MuZero

Ajedrez, Shogi y Go son juegos que fácilmente pueden ofrecer un simulador perfecto ya que sus reglas son simples y bien definidas. Sin embargo, este no es el caso para la mayoría de los escenarios del mundo real, como la robótica o los videojuegos. El aprendizaje por refuerzo basado en modelos intenta abordar este problema aprendiendo un simulador y luego permitiendo la planificación utilizando este simulador aprendido.

MuZero mejora a AlphaZero aprendiendo un simulador y extendiendo el algoritmo de búsqueda de árbol a la configuración de aprendizaje de refuerzo general de un solo agente que aprende de recompensas no escasas con descuento, lo que permite que el algoritmo domine la suite Atari Learning Environment mientras mantiene un rendimiento sobrehumano en juegos anteriores.

MuZero utiliza un enfoque diferente para superar las limitaciones de los enfoques anteriores. En lugar de intentar modelar todo el entorno, MuZero solo modela aspectos que son importantes para el proceso de toma de decisiones del agente. Después de todo, es más útil saber que un paraguas te mantendrá seco que modelar el patrón de las gotas de lluvia en el aire.

Específicamente, MuZero modela tres elementos del entorno que son críticos para la planificación:

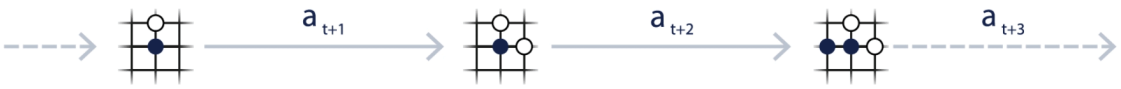
- El valor: ¿qué tan buena es la posición actual?
- La política: ¿qué acción es la mejor a tomar?
- La recompensa: ¿qué tan buena fue la última acción?

Todo esto se aprende utilizando una red neuronal profunda y es todo lo que se necesita para que MuZero comprenda lo que sucede cuando realiza una determinada acción y planifique en consecuencia.

Ilustración de cómo se puede utilizar Monte Carlo Tree Search para planificar con las redes neuronales MuZero. Comenzando en la posición actual en el juego (tablero Go esquemático en la parte superior de la animación), MuZero usa la función de representación (h) para mapear desde la observación hasta una incrustación utilizada por la red neuronal (s_0). Usando la función dinámica (g) y la función de predicción (f), MuZero puede considerar posibles secuencias futuras de acciones (a) y elegir la mejor acción.

MuZero utiliza la experiencia que recopila al interactuar con el entorno para entrenar su red neuronal. Esta experiencia incluye tanto las observaciones y recompensas del entorno, como los resultados de las búsquedas realizadas a la hora de decidir la mejor acción.

Durante el entrenamiento, el modelo se despliega junto con la experiencia recopilada, prediciendo en cada paso la información previamente guardada: la función de valor v predice la suma de las recompensas observadas (u), la estimación de la política (p) predice el resultado de la búsqueda anterior (π), la estimación de recompensa r predice la última recompensa observada (u).



Este enfoque viene con otro beneficio importante: MuZero puede usar repetidamente su modelo aprendido para mejorar su planificación, en lugar de recopilar nuevos datos del entorno. Por ejemplo, en las pruebas en la suite Atari, esta variante, conocida como MuZero Reanalyze, usó el modelo aprendido el 90% del tiempo para volver a planificar lo que debería haberse hecho en episodios anteriores.

In [ ]: