Aprendizaje por Refuerzo Profundo

Eduardo Morales

Contenido

Aproximación de Funciones

- Hasta ahora hemos supuesto que se tiene una representación explícita en forma de tabla, lo cual funciona para espacios pequeños, pero es impensable para dominios como ajedrez (10^{120}) o backgammon (10^{50}).
- Una alternativa es usar una representación implícita, i.e., una función.
- Por ejemplo en juegos, una función de utilidad estimada se puede representar como una función lineal pesada sobre un conjunto de atributos (F_i's):

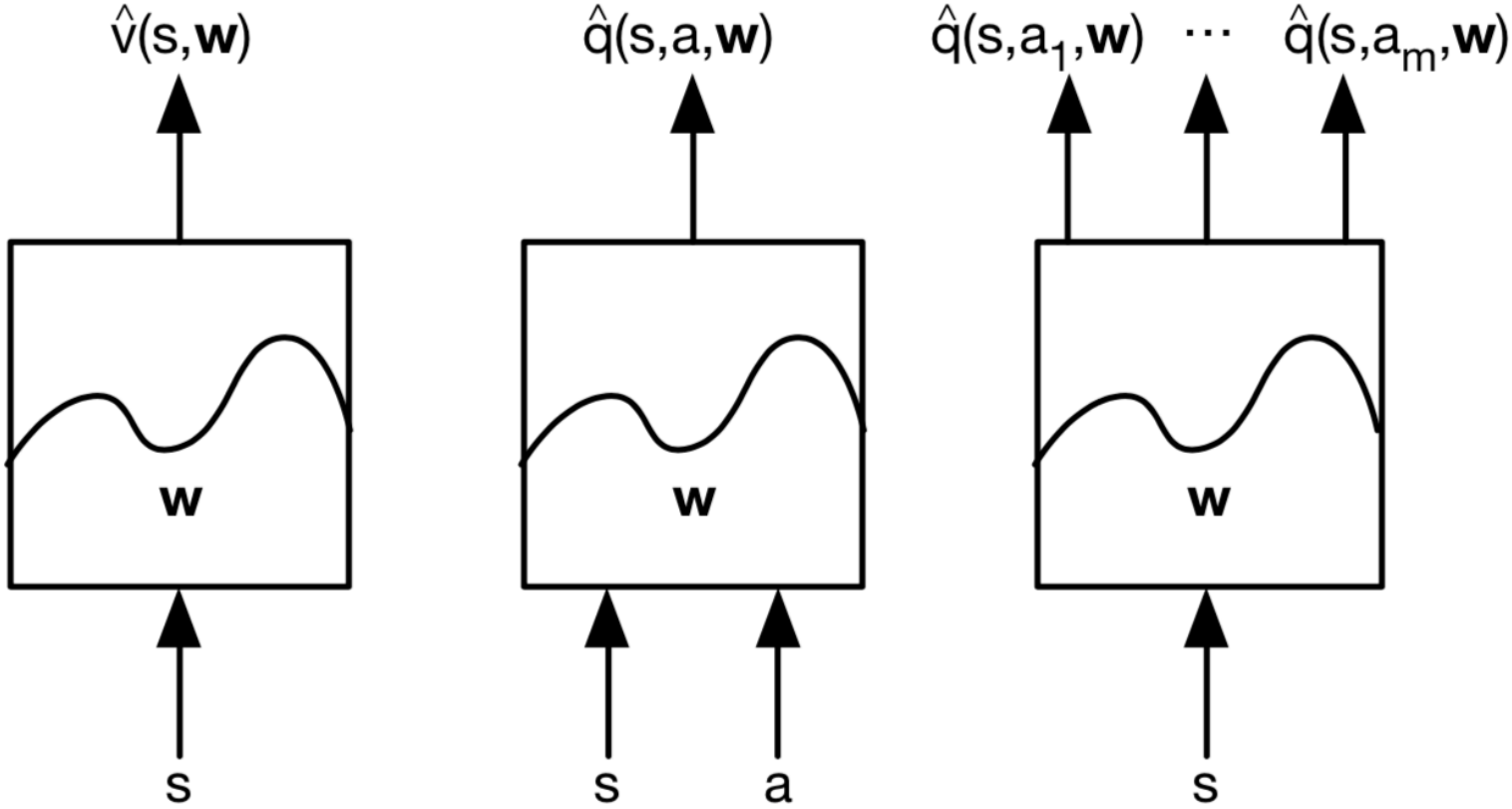
$$V(i) = w_1 f_1(i) + w_2 f_2(i) + \ldots + w_n f_n(i)$$

- En ajedrez se tienen aproximadamente 10 pesos, por lo que es una compresión bastante significativa.

Aprendizaje de Funciones

- La compresión lograda por una representación implícita permite al sistema de aprendizaje, generalizar de estados visitados a estados no visitados
- Existen muchas alternativas que se pueden usar y en RL se han usado NN, SVM, árboles de decisión, procesos gaussianos, etc.
- Como en todos los sistemas de aprendizaje, existe un balance entre el espacio de hipótesis y el tiempo que toma aprender una hipótesis aceptable
- En RL aparecen características poco comunes en sistemas de aprendizaje supervisado: no estacionariedad, recompensas diferidas, *bootstrapping*, aprender en línea,

Algunas variantes



Aprendizaje de Funciones

- Vamos a ver cómo se puede hacer con una combinación lineal de funciones base $\phi_1, \phi_2, \ldots, \phi_n$ de la forma $\sum_{i=1}^{n} w_i \phi_i$ donde los pesos w_i son los que tenemos que aprender
- Muchos sistemas de aprendizaje supervisado tratan de minimizar el error cuadrático medio (MSE)
- Si $\vec{w}_t$ representa el vector de parámetros de la función parametrizada que queremos aprender:

$$MSE(\vec{w}_t) = \sum_{s \in S} [V^{\pi}(s) - V_t(s, \vec{w}_t)]^2$$

Aprendizaje de Funciones

- Para ajustar los parámetros del vector de la función que queremos optimizar, las técnicas de gradiente ajustan los valores en la dirección que produce la máxima reducción en el error:

$$\begin{aligned} \vec{w}_{t+1} &= \vec{w}_t - \tfrac{1}{2}\alpha\nabla[v_\pi(s_t) - \hat{v}(s_t, \vec{w}_t)]^2 \\ &= \vec{w}_t + \alpha[v_\pi(s_t) - \hat{v}(s_t, \vec{w}_t)]\nabla\hat{v}(s_t, \vec{w}_t) \end{aligned}$$

donde α es un parámetro positivo $0 \leq \alpha \leq 1$ y $\nabla f(\vec{w})$ denota un vector de derivadas parciales.

Algoritmo Básico usando Monte Carlo

Dada una política π a evaluar y
una función diferenciable para v
Inicializa los pesos (w) de la función de valor
de manera arbitraria
Repite (para cada episodio):
Genera un episocio $s_0, a_0, r_1, s_1, a_1, \ldots, r_T, s_T$ usando π
Repite para cada paso del epidosio: $t = 0, 1, \ldots, T-1$

$$\vec{w} \leftarrow \vec{w} + \alpha[G_t - \hat{v}(S, \vec{w})]\nabla \hat{v}(S, \vec{w})$$

Algoritmo Básico en línea (*semi-gradient*)

Dada una política π a evaluar y
una función diferenciable para v
Inicializa los pesos (w) de la función de valor
de manera arbitraria
Repite (para cada episodio):
Inicializa S
Repite (Para cada paso del epidosio):
Selecciona $A \sim \pi(\cdot|S)$
Toma la acción A, observa R, S'
$\vec{w} \leftarrow \vec{w} + \alpha[R + \gamma \hat{v}(S', \vec{w}) - \hat{v}(S, \vec{w})] \nabla \hat{v}(S, \vec{w})$
$S \leftarrow S'$

Algoritmo Básico

El algoritmo se puede usar para diferentes esquemas:

- Para MC:

$$\delta \vec{w} = \alpha[G_t - \hat{v}(S, \vec{w})] \nabla \hat{v}(S, \vec{w})$$

- Para TD(0):

$$\delta \vec{w} = \alpha[R + \gamma \hat{v}(S', \vec{w}) - \hat{v}(S, \vec{w})] \nabla \hat{v}(S, \vec{w})$$

- Para TD(λ):

$$\delta \vec{w} = \alpha[G_t^{\lambda} - \hat{v}(S, \vec{w})] \nabla \hat{v}(S, \vec{w})$$

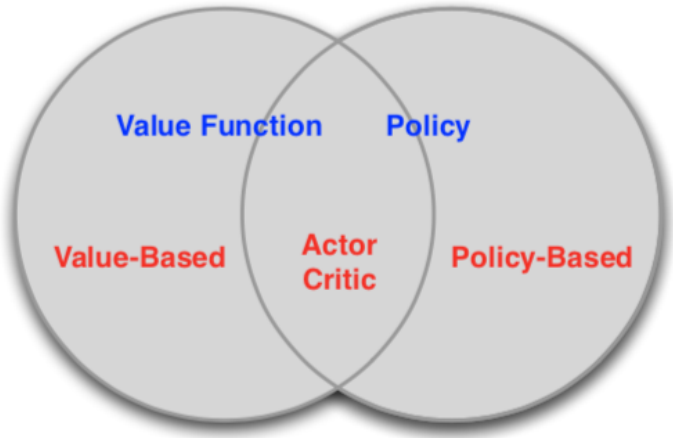
Donde $G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} \ldots = \sum_{k=0}^{\infty} r_{t+k+1}$

Aprendizaje de Funciones

- Para aprender funciones hay que tener cuidado porque no tenemos una función real y el error se calcula con la función que estamos aprendiendo!!
- Para los algoritmos *off-policy* no siempre se logra convergencia
- Para mejorar el estimador del error, a veces se usa información con trazas de elegibilidad

Opciones para Aprender Funciones

- Funciones de valor (V o Q)
- Política
- Actor-Critic: Aprende funciones de valor y política



Opciones para Aprender Funciones

- Actor ($\pi(s, a; \Theta)$): Controla cómo se comporta el agente
- Critic ($q(s, a; \Theta)$): Mide qué tan bien son las acciones
- Actor-Critic: Corren en paralelo, actualizando la política y la función de valor

Aprender la política

- En lugar de tratar de aprender la función de valor (V o Q) podemos tratar de aprender directamente la política (π)
- Normalmente se tiene definida una función parametrizada que se quiere optimizar y que es diferenciable con respecto a sus parámetros
- En lo que se conoce como gradiente de política (*policy gradient*)

Policy Gradient

- Se tiene una medida de desempeño $\eta(\Theta)$ con Θ representando los pesos de la función de la política
- Se aproximan usando gradiente:

$$\Theta_{t+1} = \Theta_t + \alpha \widehat{\nabla \eta(\Theta)}$$

- Donde $\widehat{\nabla \eta(\Theta)}$ es un estimador del gradiente
- Las ventajas de aprender directamente una política son que posiblemente es más fácil que aprender la función de valor y que se pueden aprender políticas estocásticas

The Policy Gradient Theorem

- El desempeño se puede definir como:

$$\eta(\Theta) = v_{\pi_\Theta}(s_0)$$

donde v_{π_Θ} es la función de valor verdadera para la política empezando en un estado inicial

- El teorema del gradiente de política dice que:

$$\nabla\eta(\Theta) \propto \sum_s d_\pi(s) \sum_a q_\pi(s,a) \nabla_\Theta \pi(a|s,\Theta)$$

donde $d_\pi(s)$ se define como el número esperado de pasos de tiempo en donde el estado $S_t = s$ en un episodio generado aleatoriamente empezando en s_0 y siguiendo la política π

REINFORCE

- Después de varias manipulaciones la actualización de los parámetros de la política quedan como:

$$\Theta_{t+1} = \Theta_t + \alpha \gamma^t G_t \frac{\nabla_\Theta \pi(A_t|S_t, \Theta)}{\pi(A_t|S_t, \Theta)}$$

Donde:

- γ^t es el factor de descuento multiplicado el número de veces en que llega al estado
- G_t es el retorno que se obtiene a partir de ese estado
- A_t es la acción seleccionada por la política

REINFORCE

- El teorema de política del gradiente se puede generalizar para incluir una comparación entre el valor de la acción y un valor base (*baseline*) lo que reduce la varianza:

$$\nabla \eta(\Theta) = \sum_s d_\pi(s) \sum_a (q_\pi(s,a) - b(s)) \nabla_\Theta \pi(a|s,\Theta)$$

- Con esto la actualización queda como:

$$\Theta_{t+1} = \Theta_t + \alpha (G_t - b(S_t)) \frac{\nabla_\Theta \pi(A_t|S_t,\Theta)}{\pi(A_t|S_t,\Theta)}$$

- Un candidato natural para $b(S)$ es el estimado de la función de valor $\hat{v}(S_t, w)$, donde w son los parámetros de la función de valor

Algoritmo REINFORCE

Inicializa los pesos de política (Θ) y de función de valor (w)
Repite:
Genera un episodio $S_0, A_0, R_1, S_1, \ldots, S_{T-1}, A_{T-1}, R_T$ siguiendo π
Para cada paso del epidosio $t = 0, 1, \ldots, T-1$
$G_t \leftarrow$ retorno desde el tiempo t
$\delta \leftarrow G_t - \hat{v}(S_t, w)$
$w \leftarrow w + \beta\delta\nabla_w \hat{v}(S_t, w)$
$\Theta = \Theta + \alpha\gamma^t\delta\nabla_\Theta log\pi(A_t|S_t, \Theta)$

con $\nabla logx = \frac{\nabla x}{x}$

Actor-Critic

- Aunque el algoritmo de REINFORCE aprende una política y una función de valor, la función de valor sirve como base y no como crítica
- También, como buen método Monte Carlo el aprendizaje tiende a ser lento
- Se puede hacer un algoritmo de diferencias temporales, cambiando el paso de recompensa completa que usa REINFORCE por el de un solo paso:

$$\Theta_{t+1} = \Theta_t + \alpha \left(G_t - \hat{v}(S_t, w)\right) \nabla_\Theta log\pi(A_t|S_t, \Theta)$$

Algoritmo Actor-Critic de un paso

- G_t para un solo paso lo podemos reemplazar por: $\hat{q}(S, A, w)$ ó por $R_{t+1} + \gamma \hat{v}(S_{t+1}, w)$
- En el primer caso, nos queda la función de ventaja (*advantage function*)

$$A(s, a) = Q(s, a) - V(s)$$

- La función de ventaja nos dice qué tanto se mejora con respecto al promedio en ese estado (nos da la recompensa extra que se obtiene al realizar esa acción)

$$\Theta_{t+1} = \Theta_t + \alpha A_{\pi_\Theta(s,a)} \nabla_\Theta ln\pi(A_t|S_t, \Theta)$$

- En el segundo caso, nos queda: $\delta_t \leftarrow R_t + \gamma \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)$

$$\Theta_{t+1} = \Theta_t + \alpha \delta_t \nabla_\Theta ln\pi(A_t|S_t, \Theta)$$

Algoritmo Actor-Critic de un paso

Inicializa los pesos de política (Θ) y de función de valor (w)
Repite:
Inicializa S (estado inicial del episodio)
$I \leftarrow 1$
Mientras S no es estado terminal:
$A \sim \pi(\cdot|S, \Theta)$
Toma acción A, observa S', R
$\delta \leftarrow R + \gamma \hat{v}(S', w) - \hat{v}(S, w)$
$w \leftarrow w + \alpha^w \delta \nabla_w \hat{v}(S, w)$
$\Theta \leftarrow \Theta + \alpha^\Theta I \delta \nabla_\Theta ln\pi(A|S, \Theta)$
$I \leftarrow \gamma I$
$S \leftarrow S'$

Deep RL

- El aprender directamente de entradas de alta dimensionalidad (e.g., imágenes, videos, etc.) es uno los grandes retos de RL
- Generalmente se tiene que definir uan representación adecuada la cuál es fundamental para que funcione el algoritmo
- Los algoritmos de Deep Learning pueden extraer esa representación automáticamente

Deep RL

Sin embargo, existen varios retos:

- Desde el punto de vista de DL se requieren grandes cantidades de datos etiquetados
- Desde el punto de vista de RL se tienen recompensas esparsas, ruidosas y con retrasos
- DL supone que los datos son independientes y además en RL la distribución de los datos cambia durante el aprendizaje
- Esto es un problema para DL que supone una distribución fija

DQN

- Recientemente se hizo una combinación entre Q-learning y redes convolucionales profundas (DQN)
- Se aplicó inicialmente para aprender a jugar los video-juegos de Atari
- Relevancia del trabajo de Atari es que se usó la misma estructura para aprender todos los juegos!
- Alcanzó niveles de expertos humanos en 29 de los 46 juegos

DQN

- Fue el algoritmo que despertó el interés en Deep RL al demostrar que se puede hacer una aproximación de la función Q usando una red neuronal profunda usando Q-learning
- Utiliza *experience replay*, lo que almacena las experiencias del agente en cada paso $(e_t = (s_t, a_t, r_t, s_{t+1}))$ en una base de datos $\mathcal{D} = e_1, \ldots, e_N$
- En el algoritmo se hacen actualizaciones de la función Q muestreando $\mathcal{D}$

Experience Replay

- Ventajas:
  1. Cada paso se puede usar en varias actualizaciones
  2. Aprender de muestras consecutivas es ineficiente debido a fuertes correlaciones entre muestras
  3. Usando el promedio sobre muchos datos ayuda a suavizar el aprendizaje y evitar posibles oscilaciones o divergencias de los parámetros
- Desventajas:
  1. Se almacenan los últimos N datos, lo cual no distingue posibles transiciones importantes
  2. Requieren de mucha memoria

Copias de la función de valor

- Otra forma para reducir la variabilidad es fijar una copia de la red para actualizar e intercambiar las redes cada M pasos
- La derivada de la función de pérdida con respecto a los pesos es:

$$\nabla_{\Theta_i} L(\Theta_i) = \mathbb{E}_{s,a,r,s'}$$

$$\left[\left(r + \gamma max_{a'} Q(s', a'; \Theta_i^-) - Q(s, a; \Theta_i)\right) \nabla_{\Theta_i} Q(s, a : \Theta_i)\right]$$

- Donde Θ_i^- se refiere a la red que estima Q con pesos fijos y Θ_i se refiere a la red que se está actualizando
- Cada número determinado de pasos se intercambian las redes

Algoritmo DQN

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$

For episode = 1, M **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For t = 1,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

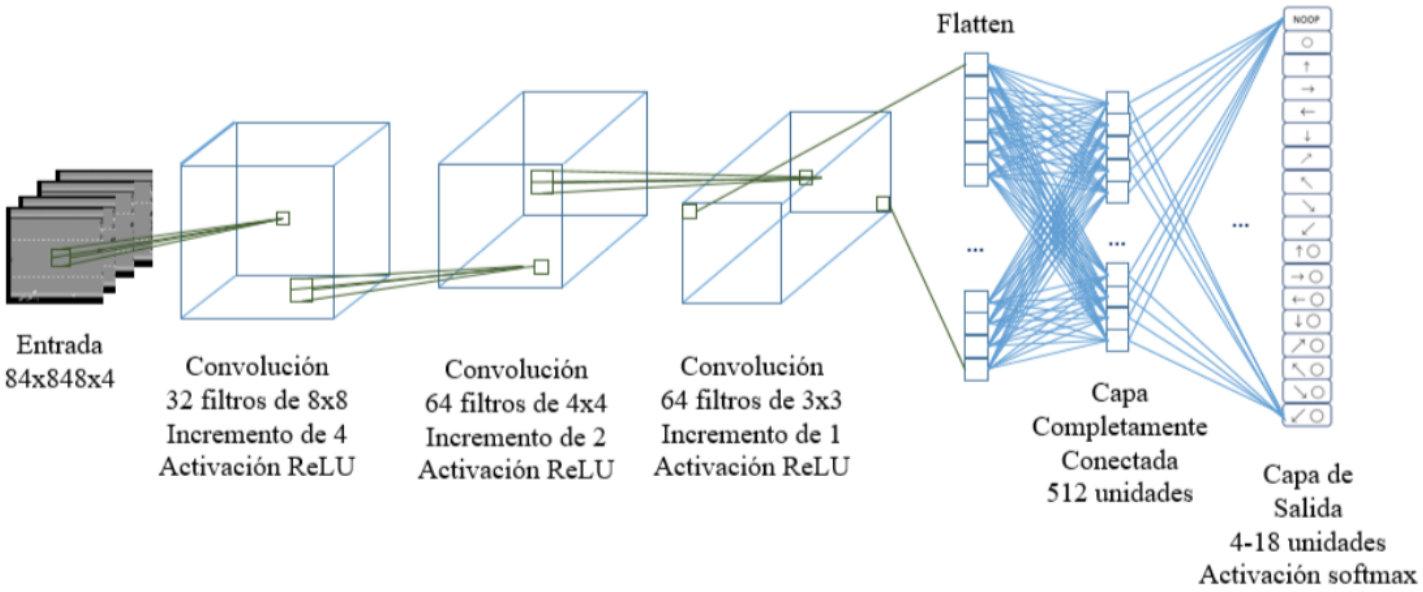
  End For

End For

DQN

- Se aprendió cada juego de 50 millones de pantallas (como 38 días de experiencia por juego)
- Las pantallas se redujeron a arreglos de 84×84 y se apilaron los últimos 4 frames (i.e., Input = $84 \times 84 \times 4$)
- La red: 3 capas convolucionales 32 20×20, 64 9×9, y 64 7×7 mapas de atributos.
- La última capa conectada a una capa de 512 unidades y de ahí a 18 unidades de salida (1 por cada posible acción)

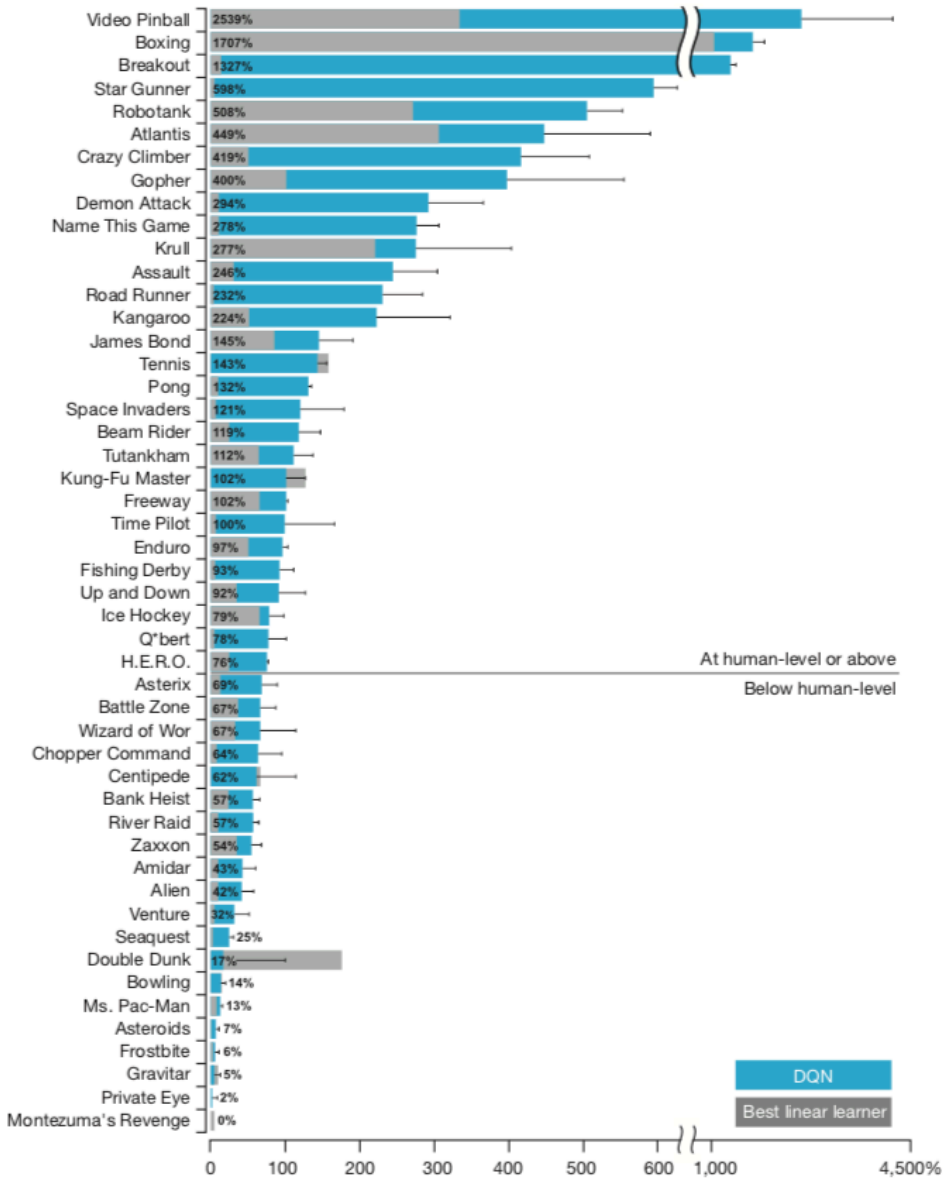
DQN



DQN

- La función de recompensa cambió de $+1/-1/0$ dependiendo de la puntuación del juego
- Exploración usando ϵ-greedy descendiendo linealmente durante el primer millón de pantallas y manteniéndose bajo después de ésto
- Usaron *mini batch*, haciendo actualización después de 32 imágenes y un algoritmo de gradiente que acelera el proceso de aprendizaje

Resultados



Prioritized Experience Replay

- Idea, algunas experiencias pueden ser más importantes que otras para entrenar, pero pueden ocurrir poco
- Al hacer *experience replay* se muestrea uniformemente
- Lo que *Prioritized Experience Replay* hace es cambiar la distribución de muestreo consierando:

$$p_t = |\delta_t| + e$$

donde $|\delta_t|$ es la magnitud del *TD error* y e es una constante para asegurar que todos tienen una probabilidad diferente de cero de ser seleccionados

Prioritized Experience Replay

- Para no ser *greedy* todo el tiempo se hace siguiendo esta distribución:

$$P(i) = \frac{p_i^a}{\sum_k p_k^a}$$

donde a indica qué tanta aleatoriedad introducir ($a = 0$ selección uniforme, $a = 1$ las de alta prioridad)

- También se pueden reforzar mucho algunas tuplas (experiencias), y para corregirlo, se usan *importance sampling weights*:

$$\left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^b$$

donde N es el tamaño del *replay buffer* y b controla qué tanto afectan los pesos al entrenamiento (b se varía (*anneal*) de 0 a 1)

DDQN

- El algoritmo de Q-learning puede sobre-estimar los valores de las acciones en ciertas condiciones
- En general, las sobre-estimaciones se pueden dar cuando las funciones de valor no son buenas
- En general, puede no ser un problema, pero si no son uniformes y no se concentran en los estados en donde queremos aprendar más, pueden tener efectos negativos
- Se prueba que DQN sobre-estima de manera importante en algunos juegos de Atari

DDQN

- Para evitar esto, DDQN descompone la función de actualización de Q-learning en dos pasos, en particular, la selección de la acción máxima
- La actualización estándar de los parámetros de la función de Q-learning es:

$$\Theta_{t+1} = \Theta_t + \alpha(Y_t^Q - Q(S_t, A_t; \Theta_t))\nabla_{\Theta_t} Q(S_t, A_t; \Theta_t)$$

Donde:

$$Y_t^Q = R_{t+1} + \gamma max_a Q(S_{t+1}, a; \Theta_t)$$

DDQN

- Se descompone, la función Q y pasa de:

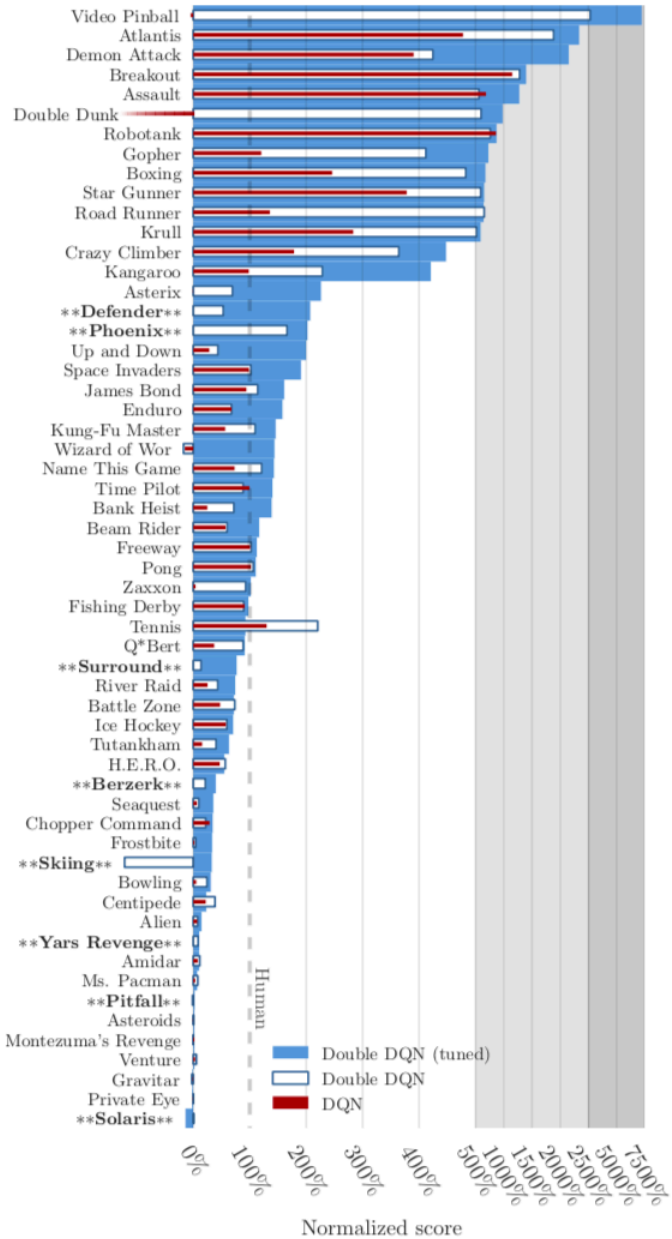
$$Y_t^Q = R_{t+1} + \gamma Q(S_{t+1}, argmax_a Q(S_{t+1}, a; \Theta_t); \Theta_t)$$

- a:

$$Y_t^{DoubleQ} = R_{t+1} + \gamma Q(S_{t+1}, argmax_a Q(S_{t+1}, a; \Theta_t); \Theta'_t)$$

- El resto del algoritmo se mantiene igual
- Tratando de mejorar los parámetros para que funcionara mejor, usaron 30,000 frames antes de copiar las Q, redujeron el nivel de exploración durante el aprendizaje y usaron un sesgo único para todas funciones de valor en la capa superior de la red

Resultados DDQN



Deep Deterministic Policy Gradients (DDPG)

- Es un algoritmo RL *model-free* para espacios de acciones contínuas (DQN es para acciones discretas)
- Mantiene dos redes:
  1. *Target policy* (actor): $\pi : S \rightarrow A$
  2. *Action-value function approximator* (critic) $Q : S \times A \rightarrow R$
- Los episodios se generan siguiendo una política de comportamiento, que es una versión ruidosa de la política objetivo: $\pi_b(s) = \pi(s) + \mathbb{N}(0, 1)$
- El critic se entrena como DQN pero los objetivos (y_t) se calculan usando las acciones generadas por el actor, i.e.: $y_t = r_t + \gamma Q(s_{t+q}, \pi(S_{t+1}))$
- El actor se entrena con un mini-batch gradient descend con loss $L_a = -\mathbb{E}_s Q(s, \pi(s))$ donde s se obtiene del replay buffer

Enfoques Distribuidos

- Varios enfoques se han usado para diseñar esquemas distribuidos que permitan escalar los algoritmos de DRL
- *Distributed Stochastic Gradient Descent*: Paraleliza el cálculo de los gradientes
- *Distributed Importance Sampling*: Muestrea datos con probabilidad proporcional al tamaño de los gradientes
- *Prioritized Experience Replay*: Parecido, pero se enfoca en las experiencias con resultados “inesperados” - útil con recompensas esparsas

A3C

- El usar *experience replay* funcionó para evitar los datos altamente correlacionados y poder hacer muestreos aleatorios
- Sin embargo, requiere de mucha memoria y es aplicable a algoritmos *off-policy*
- En A3C proponen ejecutar el aprendizaje de múltiples agentes en paralelo, pensando que cada agente va a tener diferentes experiencias
- A3C puede correrse en un CPU de varios núcleos "estándar"

A3C

- Corren diferentes políticas en diferentes hilos haciendo actualizaciones en línea en paralelo
- Cada agente tiene su propia copia del ambiente, calcula su gradiente, y se comparte la red que calcula la función de pérdida y usan mini-batches
- Cada hilo explora una política distinta
- Se usan trazas de elegibilidad pero usando *forward view*
- La política y la función de valor se actualizan después de t_{max} o al llegar a un estado terminal.
- Probaron varios algoritmos de optimización (SGD con momento, RMSProp y RMSProp compartiendo estadísticas (que les funcionó mejor))

A3C

Probaron varias opciones:

- Q-learning de un paso asíncrono
- SARSA de un paso asíncrono
- Q-learning de N pasos asíncronos (trazas de elegibilidad)
- Actor-Critic Asíncrono con *advantage* (A3C o *asynchronos advantage actor-critic*)

Algoritmo A3C

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ *and* θ_v *and global shared counter* $T = 0$
// Assume thread-specific parameter vectors θ' *and* θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial\,(R - V(s_i;\theta'_v))^2/\partial\theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

En realidad:
$\nabla_{\Theta'} log\pi(a_t|s_t;\Theta')(R_t - V(s_t;\Theta_v)) + \beta\nabla_{\Theta'} H(\pi(a_t|s_t;\Theta'))$
donde H es entropía

Ape-X

- Ape-X genera muchos datos en paralelo, usa *prioritized experience replay* y un solo agente para aprender
- Usa una memoria de *experience replay* centralizada
- Combina datos de actores que tienen diferentes políticas de exploración, lo que aumenta la diversidad de los ejemplos
- Usa Double Q-Learning con trazas de elegibilidad y una arquitectura de red *dueling*
- Para todos los elementos en el batch se calcula la función de pérdida:

$$l_t(\Theta) = \frac{1}{2}(G_t - q(S_t, A_t, \Theta))^2$$

Ape-X

Con:

$$G_t = \underbrace{R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n \overbrace{q(S_{t+n}, argmax_a q(S_{t+n}, a, \Theta), \Theta^-)}^{\text{double-Q bootstrap value}}}_{\text{multi-step return}}$$

donde:

- Θ^- son los parámetros de la red objetivo. Si el episodio acaba antes de lo n pasos se trunca. Los actores usan $\epsilon-$greedy con diferentes valores de ϵ.

Ape-X

- Para acciones contínuas se adapta a DDPG, donde es parecido, pero ahora se tiene una red para los valores Q (con parámetros ψ) y una red separada para la política (con parámetros θ)
- Las actualizaciones son:

$$l_t(\psi) = \frac{1}{2}(G_t - q(S_t, A_t, \psi))^2$$

donde:

$$G_t = \underbrace{R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n q(S_{t+n}, \pi(S_{t+n}, \theta^-), \psi^-)}_{\text{multi-step return}}$$

- La red de la política genera una acción y sus parámetros se actualizan usando un gradiente: $\nabla_\theta q(S_t, \pi(S_t, \theta), \psi)$

Algoritmos de Ape-X

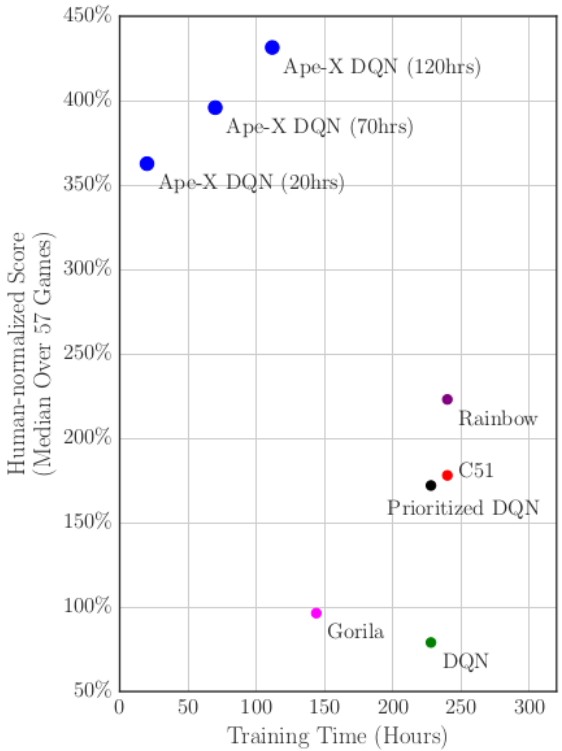
Algorithm 1 Actor

procedure ACTOR(B, T)
  $\theta_0 \leftarrow$ LEARNER.PARAMETERS( ) ▷ Remote call to obtain latest network parameters.
  $s_0 \leftarrow$ ENVIRONMENT.INITIALIZE( ) ▷ Get initial state from environment.
  for $t = 1$ **to** T **do**
    $a_{t-1} \leftarrow \pi_{\theta_{t-1}}(s_{t-1})$ ▷ Select an action using the current policy.
    $(r_t, \gamma_t, s_t) \leftarrow$ ENVIRONMENT.STEP(a_{t-1}) ▷ Apply the action in the environment.
    LOCALBUFFER.ADD($(s_{t-1}, a_{t-1}, r_t, \gamma_t)$) ▷ Add data to local buffer.
    if LOCALBUFFER.SIZE( ) $\geq B$ **then** ▷ In a background thread, periodically send data to replay.
      $\tau \leftarrow$ LOCALBUFFER.GET(B) ▷ Get buffered data (e.g. batch of multi-step transitions).
      $p \leftarrow$ COMPUTEPRIORITIES(τ) ▷ Calculate priorities for experience (e.g. absolute TD error).
      REPLAY.ADD(τ, p) ▷ Remote call to add experience to replay memory.
    end if
    PERIODICALLY($\theta_t \leftarrow$ LEARNER.PARAMETERS()) ▷ Obtain latest network parameters.
  end for
end procedure

Algorithm 2 Learner

procedure LEARNER(T) ▷ Update network using batches sampled from memory.
  $\theta_0 \leftarrow$ INITIALIZENETWORK( )
  for $t = 1$ **to** T **do** ▷ Update the parameters T times.
    $id, \tau \leftarrow$ REPLAY.SAMPLE( ) ▷ Sample a prioritized batch of transitions (in a background thread).
    $l_t \leftarrow$ COMPUTELOSS($\tau; \theta_t$) ▷ Apply learning rule; e.g. double Q-learning or DDPG
    $\theta_{t+1} \leftarrow$ UPDATEPARAMETERS($l_t; \theta_t$)
    $p \leftarrow$ COMPUTEPRIORITIES( ) ▷ Calculate priorities for experience, (e.g. absolute TD error).
    REPLAY.SETPRIORITY(id, p) ▷ Remote call to update priorities.
    PERIODICALLY(REPLAY.REMOVETOFIT()) ▷ Remove old experience from replay memory.
  end for
end procedure

Desempeño de Ape-X



- Encontraron que aumentando el número de actores mejora el desempeño (más experiencia)
- Parece como que muchos sistemas actuales están sobre-ajustando

R2D2

- Recientemente se han usado LSTMs para lidear con información parcial
- *Recurrent Replay Distributed DQN* (R2D2) entrena redes recurrentes con *experience replay* distribuido
- Parecido a Ape-X (*prioritized distributed replays*) y $n-$step double Q-Learning (con $n = 5$), generando experiencia de 256 actores, pero incluye una capa LSTM después de las convoluciones
- En lugar de guardar tuplas de transiciones (s, a, r, s'), se guardan secuencias de tamaño fijo (80) de (s, a, r) con secuencias adyacentes solapadas en 40 pasos.

R2D2

- Al entrenar usan la red en línea y la meta sobre las mismas secuencias
- Las n-pasos metas para la función Q son:

$$\hat{y}_t = h\left(\sum_{k=0}^{n-1} r_{t+k}\gamma^n h^{-1}(Q(s_{t+n}, a^*; \Theta^-))\right)$$

donde:
Θ^- son los parámetros de la red meta
$a^* = argmax_a Q(s_{t+n}, a; \Theta)$
$h(x) = sign(x)(\sqrt{|x|+1} - 1) + \epsilon x$

- A diferencia de Ape-X usa una mezcla de *max* y *mean* $n-$steps TD-errors δ_i sobre la secuencia:
$p = \eta max_i \delta_i + (1-\eta)\overline{\delta}$ con $\eta = 0.9$.

R2D2

- Para lidear con información parcial se requiere una representación de estado que codifique información acerca de la trayectoria estado-acción, además de la observación actual
- Para esto, se usan redes recurrentes, en general, LSTM, por lo que se guardan trayectorias en el *experience replay*
- Se han usado dos estrategias para entrenar una LSTM con secuencias muestreadas aleatoriamente:
  - *Zero start state*: Se inicializa la red al principio de la secuencia muetreada (sencilla, pero el estado inicial puedo no corresponder al de la secuencia)
  - Simulando las trayectorias completas de los episodios (problemas de variabilidad por usar secuencias altamente correlacionadas)

R2D2

- Se proponen dos nuevas estrategias:
  - *Stored State*: Guardar el estado recurrente en una simulación y usarlo como estado inicial en entrenamiento
  - *Burn-in*: Usar parte de la secuencia, para simular la red y producir un estado inicial, y a partir de ahí entrenar la red con el resto de la secuencia.

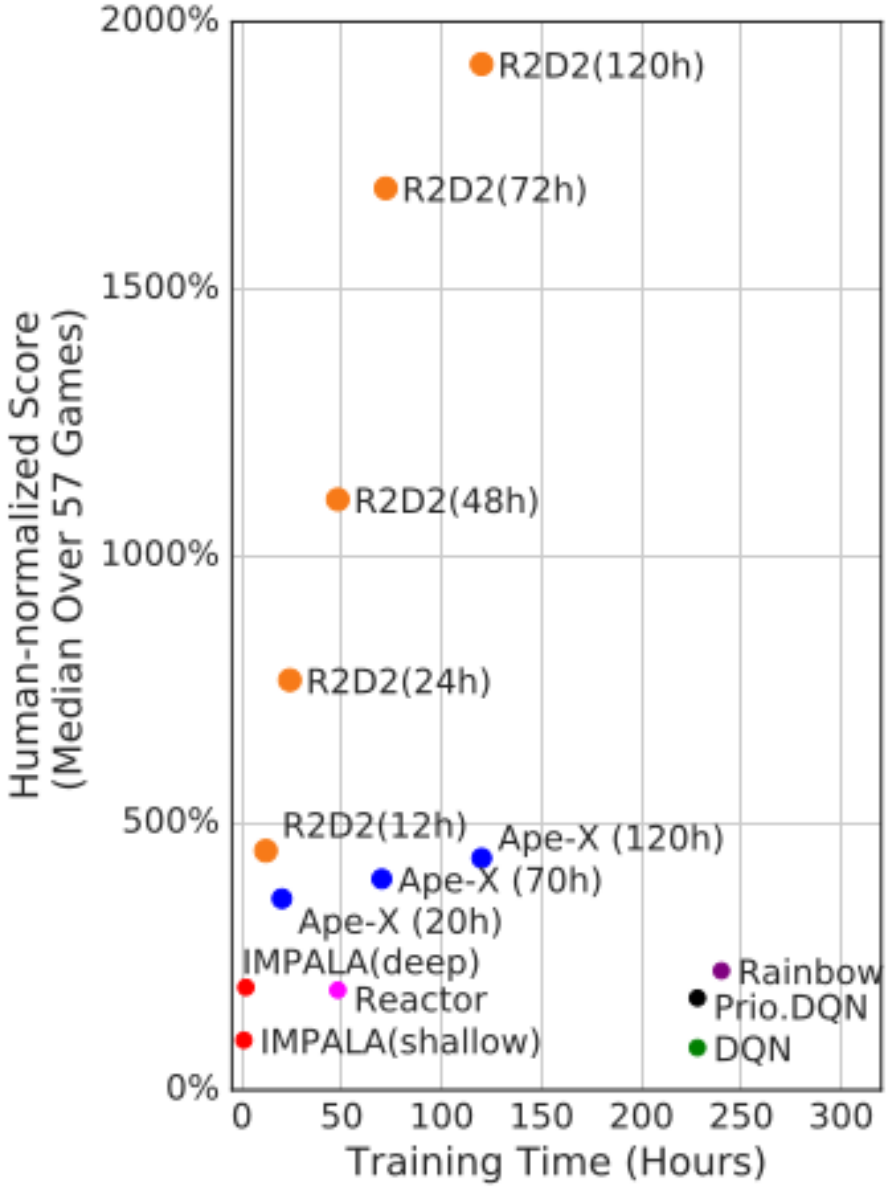
R2D2

- En cualquiera de los casos se mide la discrepancia entre los valores Q como:

$$\nabla Q = \frac{||q(\hat{h}_{t+i}; \hat{\Theta}) - q(h_{t+i}; \hat{\Theta})||_2}{|max_{a,j}(q(\hat{h}_{t+j}; \hat{\Theta}))_a|}$$

donde:
$h_{t+1} = h(o_t, h_t; \Theta)$
$\hat{h}_{t+i} = h(o_{t+i-1}, h_{t+i-1}; \hat{\Theta})$ que se obtiene al desdoblar la red con la secuencia $o_t, \ldots, o_{t+l+m-1}$, para la estrategia *stored state* ($i = l$) y para la estrategia *burn-in* ($i = l + m - 1$).

Desempeño de R2D2



PPO (Proximal Policy Optimization)

- Métodos de gradiente de política que alternan entre muestrear datos y optimizar una función objetivo surrogada usando SGD
- Usan una función objetivo con *clipped probability ratios* lo que crea un estimado pesimista (*lower bound*) del desempeño de la política
- En los métodos que buscan la política óptima y usan un gradiente, lo más común es:

$$\hat{g} = \mathbb{E}\left[\nabla_\Theta log \pi_\Theta(a_t|s_t)\hat{A}_t\right]$$

- Donde π_Θ es la política, $\hat{A}_t$ es la función *advantage*, y $\mathbb{E}$ es el promedio empírico sobre un conjunto finito de muestras de un batch.

PPO

- TRPO utiliza la función objetivo:

$$maximiza_\Theta \mathbb{E}\left[\frac{\pi_\Theta(a_t|s_t)}{\pi_{\Theta_{old}}(a_t|s_t)}\hat{A}_t\right]$$

- Sujeto a:

$$\mathbb{E}\left[KL[\pi_{\Theta_{old}}(\cdot|s_t), \pi_\Theta(\cdot|s_t)]\right] \leq \delta$$

- Lo que acaban haciendo es introduciendo una penalización, en lugar de la restricción:

$$maximiza_\Theta \mathbb{E}\left[\frac{\pi_\Theta(a_t|s_t)}{\pi_{\Theta_{old}}(a_t|s_t)}\hat{A}_t - \beta KL[\pi_{\Theta_{old}}(\cdot|s_t), \pi_\Theta(\cdot|s_t)]\right]$$

PPO

- Pero como es difícil encontrar un valor de β que funcione en general, optimizan una función simplificada (surrogada).
- Definen la razón de probabilidad: $r_t(\Theta) = \frac{\pi_\Theta(a_t|s_t)}{\pi_{\Theta_{old}}(a_t|s_t)}$ y lo que TRPO maximiza es:

$$L^{CPI}(\Theta) = \mathbb{E}\left[\frac{\pi_\Theta(a_t|s_t)}{\pi_{\Theta_{old}}(a_t|s_t)}\hat{A}_t\right] = \mathbb{E}\left[r_t(\Theta)\hat{A}_t\right]$$

- Sin la restricción, esta función objetivo surrogada puede dar actualizaciones en la política muy grandes

PPO

- PPO lo que hace es:

$$L^{CPI}(\Theta) = \mathbb{E}\left[min(r_t(\Theta)\hat{A}_t, clip(r_t(\Theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right]$$

- Lo que hace es que remueve el incentivo para mover r_t fuera de un rango $[1-\epsilon, 1+\epsilon]$

RAINBOW

Después de la publicación de DQN, se propusieron varias mejoras:

- DDQN: Reduce la sobre-estimación que puede presentar Q-learning separando la ecuación de actualización
- *Prioritized experience replay*: Muestrea transiciones con una probabilidad que depende del último valor absoluto del error de TD
- *Dueling network*: Corre dos redes, una de función de valor y otra de la función de *advantage* y al final las junta en un agregador particular

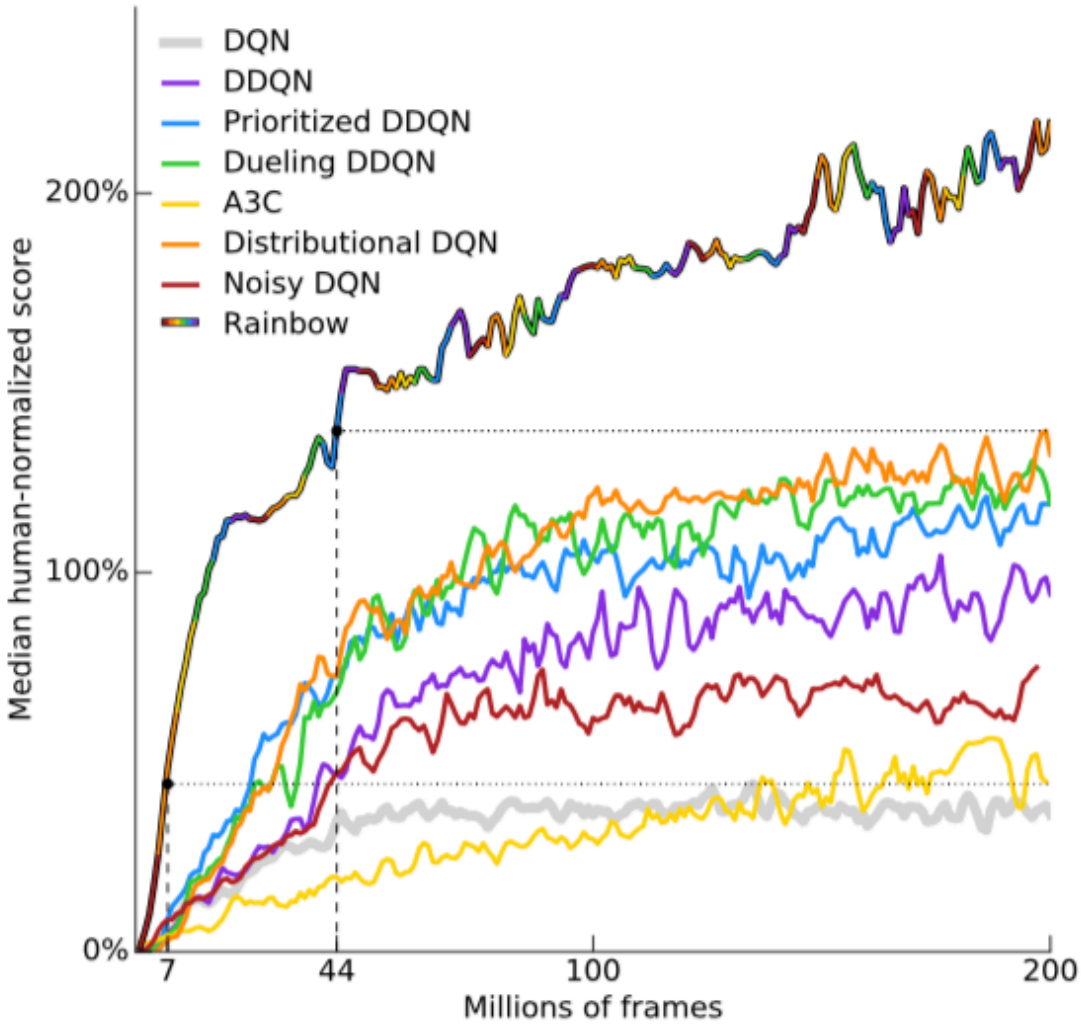
RAINBOW

Después de la publicación de DQN, se propusieron varias mejoras:

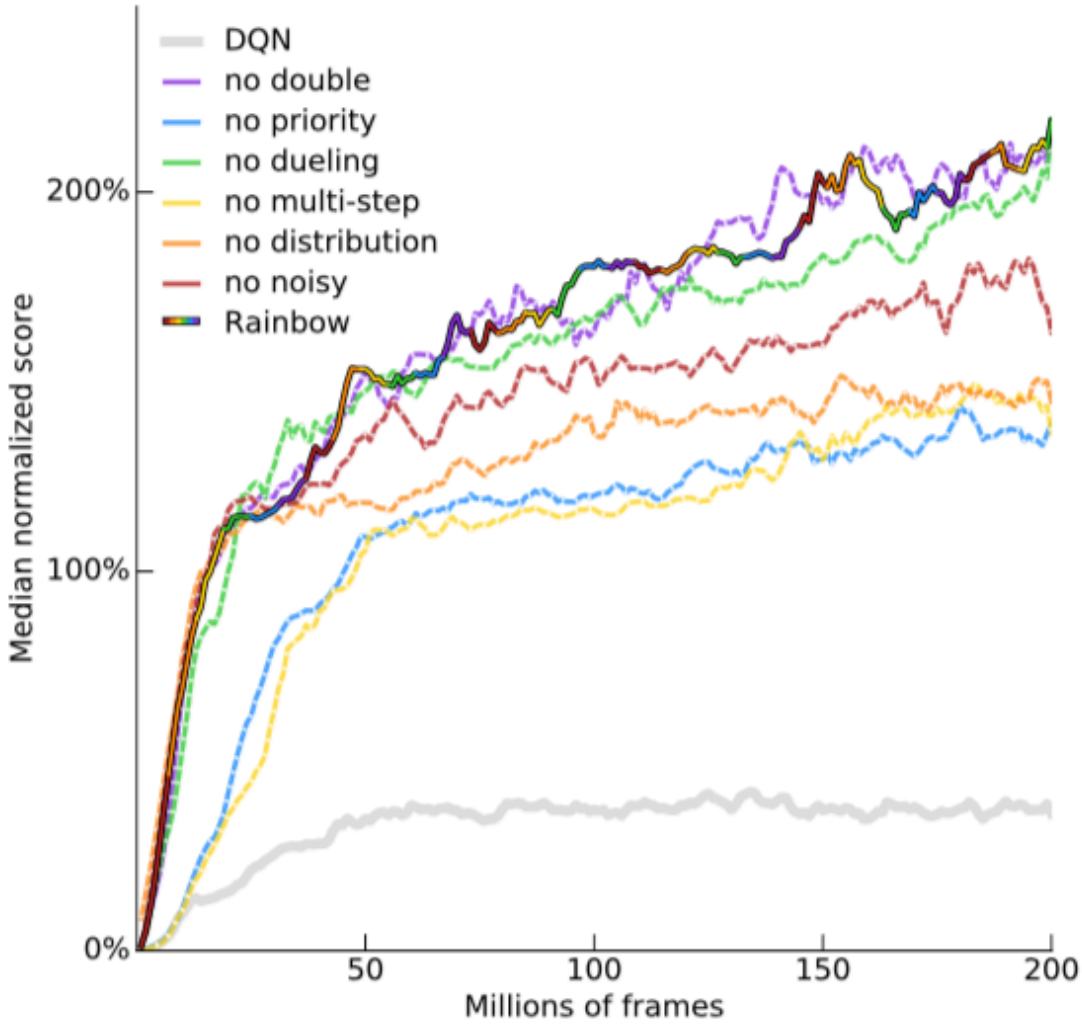
- *Multi-step learning*: Usar trazas de elegibilidad en *forward-view*
- *Distributional RL*: Aprenden a aproximar la distribución de recompensas, en lugar de la recompensa esperada
- *Noisy* DQN: Introduce ruido que con el tiempo se va disminuyendo, mejorando el proceso de exploración

RAINBOW

RAINBOW hace una combinación de todo (con algunos ajustes) y prueba que da mejores resultados que todos los demás.



Resultados



Otros Enfoques

- Finalmente RL quiere aprender a decidir cuál es la mejor acción a tomar en cada estado para cumplir un objetivo
- A veces, se pueden tener recompensas muy esparsas y por lo mismo difícil de encontrar
- Generalmente el usuario es quien define la función de recompensa para “ayudarle” al sistema
- En algunos casos, pueden existir varias metas que se quieran cumplir
- Se pueden crear metas intermedias artificiales para ayudar al agente
- Se puede buscar que el agente aprenda inicialmente las tareas sencillas y después las más complicadas

Universal Value Function Approximator (UVFA)

- Es una extensión de DQN para cuando pueden existir más de una meta que cumplir.
- Si $\mathcal{G}$ es el espacio de posibles metas, cada $g \in \mathcal{G}$ tiene una función de recompensa $r_g : S \times A \rightarrow R$.
- Cada episodio empieza muestreando un par estado-meta de alguna distribución
- La meta se mantiene fija durante todo el episodio.
- En cada paso se obtiene información del estado actual y de la meta actual, $\pi : S \times G \rightarrow A$ y obtiene de recompensa $r_t = r_g(s_t, a_t)$
- La función Q ahora depende del par estado-acción y de la meta: $Q^{\pi}(s_t, a_t, g) = \mathbb{E}[R_t | s_t, a_t, g]$

Hindsight Experience Replay

- Uno de los retos importantes en RL es cómo lidear con recompensas esparsas (pocas y alejadas)
- Muchas veces, el programador es quien tiene que diseñar una función de recompensa adecuada para el sistema funcione
- Al igual que UVFA entrena funciones de valor considerando metas, sólo que estas se definen automáticamente durante el aprendizaje.
- Después de una secuencia en un episodio $(s_0, \ldots, s_T)$, se guardan las transiciones, no sólo con la meta original sino que también con un conjunto de otras metas.
- Se hace un *replay* con la meta $m(s_T)$ (i.e., la meta que se consiguió al final del episodio).

Hindhsight Experience Replay

- Se consideraron varias estrategias para incluir submetas:
  1. *Final*: Usar el estado final al que se llegó en el episodio
  2. *Future*: *Replay* con k estados aleatorios que están en el mismo episodio y que se observaron después del estado
  3. *Episode*: *Replay* con k estados aleatorios que están en el mismo episodio
  4. *Random*: *Replay* con k estados aleatorios que se han observado en todo el proceso de entrenamiento
- En general, *future*, se comporta mejor con k=4.

Curriculum Learning

- Idea: Aprender primero las tareas fáciles
- Originalmente se le dio el orden al sistema y después surgieron esquemas para determinarlo
- Self-paced learning ...

También existe: *Curriculum-guided Hindsight Experience Replay* combinando ideas de los dos.

Self-play (Alice & Bob)

- Un tipo de auto-juego no simétrico, con dos agentes (Alice y Bob)
- Alice empieza en estado inicial (s_0) y después de una secuencia de acciones llega un estado s_t.
- En ambientes reversibles, la tarea de Bob es empezar en s_t y regresar a s_0; en ambientes "reseteables", la tarea de Bob es empezar en s_0 y llegar a s_t

Self-play (Alice & Bob)

- Lo interesante es en la definición de recompensas que incitan a Alice a ponerle tareas cada vez más difíciles a Bob, pero no imposibles

$$R_B = -\gamma t_B$$

donde R_B es la recompensa total del espisodio para Bob y t_B es el tiempo que le tomó completarla

$$R_A = \gamma max(0, t_B - t_A)$$

donde R_A es la recompensa total del espisodio para Alice y t_A es el tiempo que le tomó completarla

- El tiempo total de cada episodio está limitado por t_{max}, si Bob no acaba la tarea, $t_B = t_{max} - t_A$

Self-play (Alice & Bob)

- A Alice le conviene que Bob se tarde más que ella, pero no mucho más
- Alice tampoco se quiere tardar más que t_{max}, por lo que limita sus pasos para que sea más fácil para Bob
- Lo mejor para Alice es encontrar las tareas más sencillas (t_A pequeña) que Bob no puede resolver (t_B grande).

Go

- Comparado con ajedrez, tiene más movimientos legales por posición ($\approx$ 250 vs. $\approx$ 35) y más movimientos por juego ($\approx$ 150 vs. $\approx$ 80 en ajedrez)
- Es difícil definir una función de evaluación adecuada
- AlphaGo combina Redes Neuronales Profundas, *Monte Carlo Tree Search* (MCTS), Aprendizaje Supervisado y Aprendizaje por Refuerzo.
- Modifica MCTS usando funciones de valor
- Define (y aprende) varias redes: la red de política usando aprendizaje supervisado, la red de simulaciones Monte Carlo y la red de la función de valor.

AlphaGo

- APC-MCTS: A diferencia de MCTS expande el árbol de acuerdo a una CNN de 13 capas entrenadas previamente con 30 millones de jugadas
- La evaluación se hace combinando las estadísticas y una función de valor obtenida de otra CNN de 13 capas
- Las simulaciones para ésto se obtuvieron de otro sistema de aprendizaje sencillo

AlphaGo Zero

- Aprende sólo de auto juegos, haciendo un tipo de iteración de políticas: evaluación y mejora de la política
- Usa MCTS para seleccionar acciones y una sola CNN
- MCTS corre una simulación hasta una hoja del árbol de búsqueda actual (en lugar de hasta un estado terminal)
- La entrada a la CNN son matrices de $19 \times 19 \times 17$, representando 17 planos de atributos binarios (8 representando las piedras del jugador, 8 las del jugador contrario y 1 indicando el color que le toca jugar).
- La red aprende: $f_\Theta(s) = (p, v)$, con $p_a = Pr(a|s)$ y v = probabilidad de ganar desde la posición actual

AlphaGo Zero

- En el árbol de búsqueda cada nodo tiene la siguiente información: $\{N(s,a), W(s,a), Q(s,a), P(s,a)\}$, donde:
  - $N(s,a)$ es cuántas veces se ha visitado ($N(s,a) = N(s,a) + 1$)
  - $W(s,a)$ es la función de valor total ($W(s,a) = W(s,a) + v$)
  - $Q(s,a)$ es la función de valor promedio ($Q(s,a) = \frac{W(s,a)}{N(s,a)}$)
  - $P(s,a)$ es la función de probabilidad *a priori* (lo que aprende la red)

AlphaGo Zero

- En cada paso se selecciona la acción:
  $a_t = \text{argmax}_a(Q(s_t, a) + U(s_t, a))$
  donde:

$$U(s, a) = cP(s, a)\frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}$$

- MCTS se usa para mejorar la política

AlphaGo Zero

- La CNN tiene 41 capas convolucionales y después se divide en 2:
  1. Una parte genera 362 salidas ($19^2 + 1$) que da la probabilidad de movida ($Prob(a|s)$) en cada lugar en el tablero + *pasar* (no hacer nada)
  2. La otra parte genera una sola salida que estima la probabilidad de que el jugador gane desde la posición actual (v)
- La red se entrenó usando "batches" de ejemplos tomados aleatorios de 500 mil juegos con la mejor política actual
- Cada 1,000 pasos de entrenamiento se prueba la nueva red con la vigente. Si resulta mejor que un cierto umbral se reemplaza.

AlphaGo Zero

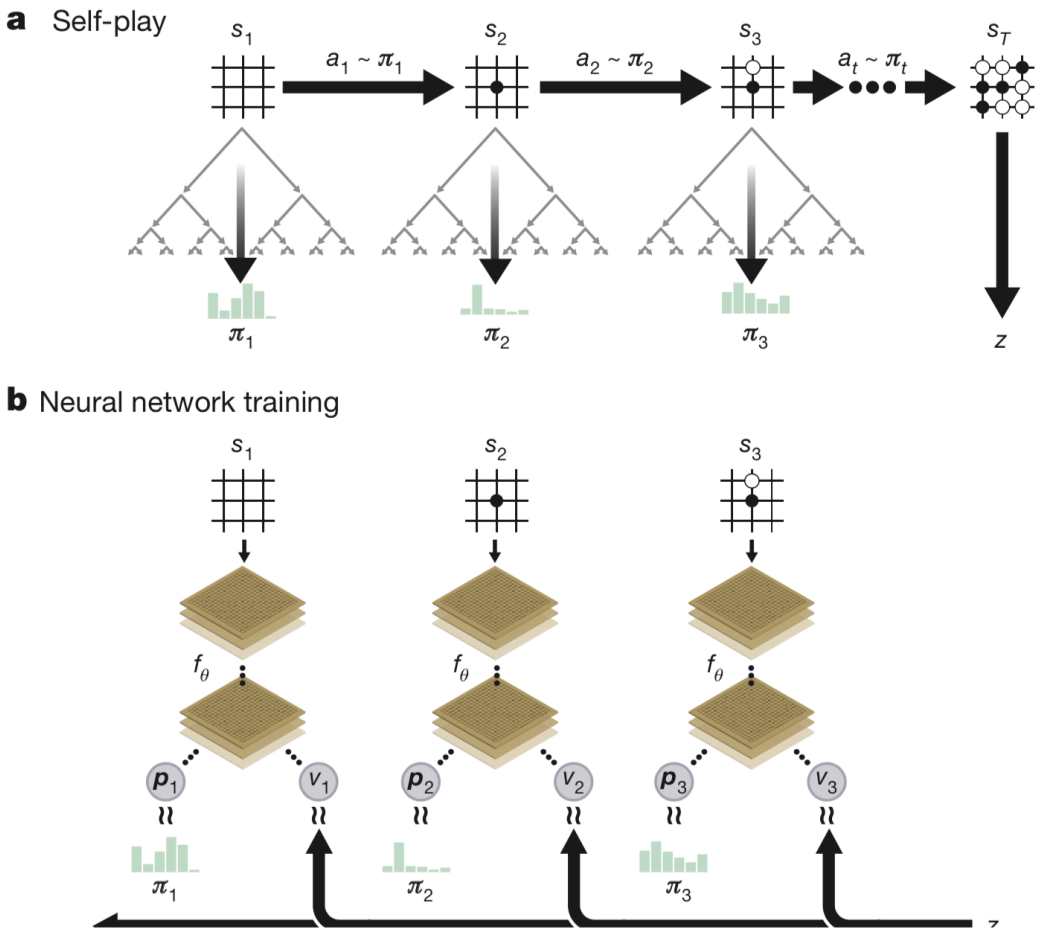
- Se entrenó con 4.9 millones de auto juegos que tomó alrededor de 3 días.
- Cada movida de cada juego se seleccionaba al correr MCTS 1,600 iteraciones (como 0.4 segundos por movida).
- Los pesos de la red se actualizaban sobre 700,000 batches cada uno con 2,048 posiciones
- En cada posición se ejecuta un MCTS guiado por la red neuronal
- El MCTS regresa una política (π) con las probabilidades de hacer cada movida y se usa como mejorador de políticas

AlphaGo Zero

- La red se entrena para maximizar la similaridad entre la predicción p y la política π obtenida con MCTS y para minimizar el error entre la predicción de quien va a ganar v y el resultado actual z.
- La función de pérdida (loss) es:

$$l = (z - v)^2 - \pi^T \log p + c||\Theta||^2$$

AlphaGo Zero



muZero

- RL lo podemos dividir en *model-based* y *model-free*
- *Model-based* aprenden un modelo (generalmente la función de transición y la función de recompensa)
- En ambientes complejos también se tiene que aprender una representación adecuada
- muZero predice la recompensa, la política y la función de valor

muZero

- Las predicciones se hacen en cada tiempo t, para cada $k = 1...K$ pasos, por un modelo μ_Θ condicionado en las observaciones pasadas $o_1, \ldots, o_t$ y acciones futuras $a_{t+1}, \ldots, a_{t+k}$
- El modelo predice:
  1. $p_t^k \approx \pi(a_{t+k+1} | o_1, \ldots, o_t, a_{t+1}, \ldots, a_{t+k})$
  2. $v_t^k \approx \mathbb{E}[u_{t+k+1} + \gamma u_{t+k+2} + \ldots | o_1, \ldots, o_t, a_{t+1}, \ldots, a_{t+k})$
  3. $r_t^k \approx u_{t+k}$
- Donde: u es la recompensa recibida, π es la política usada para seleccionar las acciones y γ es un factor de descuento

muZero (modelos y ecuaciones)

- La función del modelo:

$$\mathbf{p}^k, v^k, r^k = \mu_\Theta(o_1, \ldots, o_t, a_{t+1}, \ldots, a_{t+k})$$

- Internamente, el modelo está representado por:
  - Una función de representación de estado:

$$s^0 = h_\Theta(o_1, \ldots, o_t)$$

  - Una función de la dinámica:

$$r^k, s^k = g_\Theta(s^{k-1}, a^k)$$

  - Una función de predicción:

$$\mathbf{p}^k, v^k = f_\Theta(s^k)$$

muZero (modelos y ecuaciones)

- La función de pérdida es:

$$l_t(\Theta) = \sum_{k=0}^{K} l^r(u_{t+k}, r_t^k) + l^v(z_{t+k}, v_t^k) + l^p(\pi_{t+k}, p_t^k) + c||\Theta||^2$$

donde:

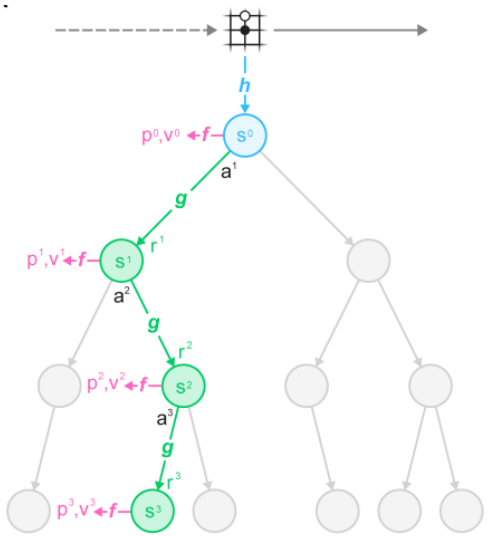
$$l^r(u, r) = \begin{cases} 0 & \text{para juegos} \\ \phi(u)^T log r & \text{para MDPs} \end{cases}$$

$$l^v(z, q) = \begin{cases} (z-q)^2 & \text{para juegos} \\ \phi(z)^T log q & \text{para MDPs} \end{cases}$$

$$l^p(\pi, p) = \pi^T log \mathbf{p}$$

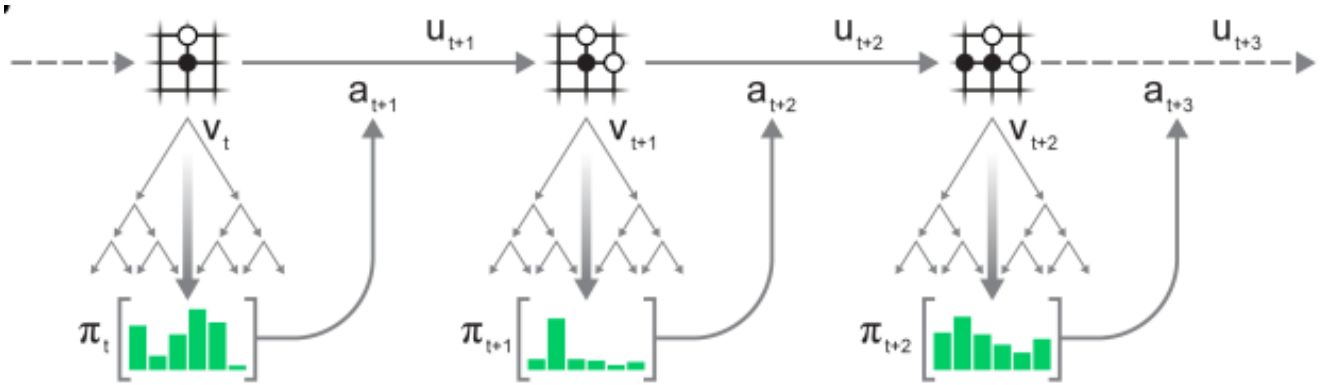
donde $\phi(x)$ es una función para representar un número real x como una combinación lineal de enteros contíguos.

muZero: Para planear



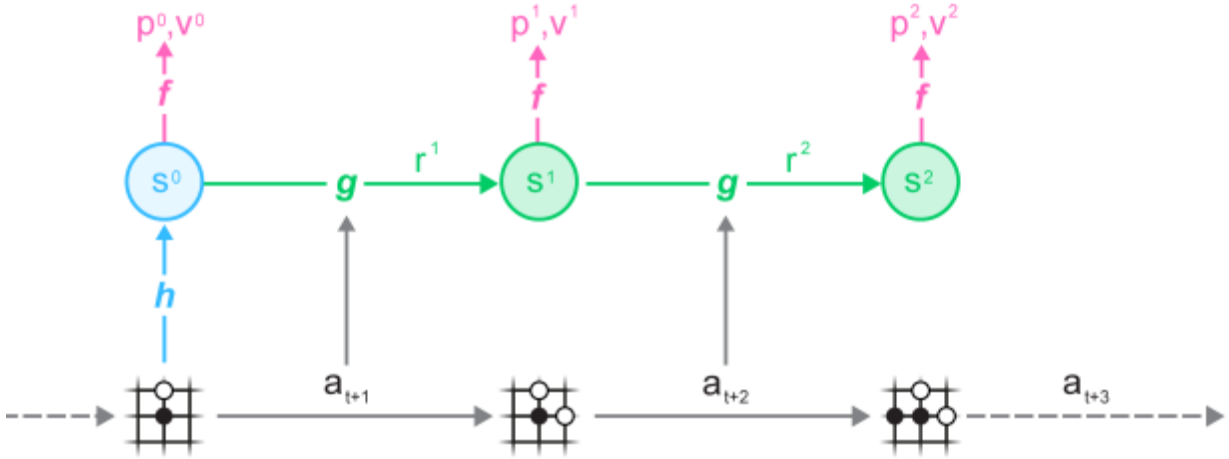
- Dado un estado s^{k-1} y una acción candidata a^k, la función de la dinámica g produce una recompensa inmediata r^k y un estado oculto s^k
- La función de política p^k y función de valor v^k se calculan del estado oculto s^k por la función de predicción f
- El estado oculto inicial s^0 se obtiene de la observación (tablero o pantalla) por la función de representación h

muZero: Para actuar en el ambiente



- En cada paso t se hace un MCTS (lámina anterior)
- La acción a_{t+1} se muestrea de la política π_t proporcional a cuántas veces se ha usado
- El ambiente genera una nueva observación o_{t+1} y recompensa u_{t+1}
- Al final del episodio la trayectoria se almacena en un *replay buffer*

muZero: Para entrenar el modelo



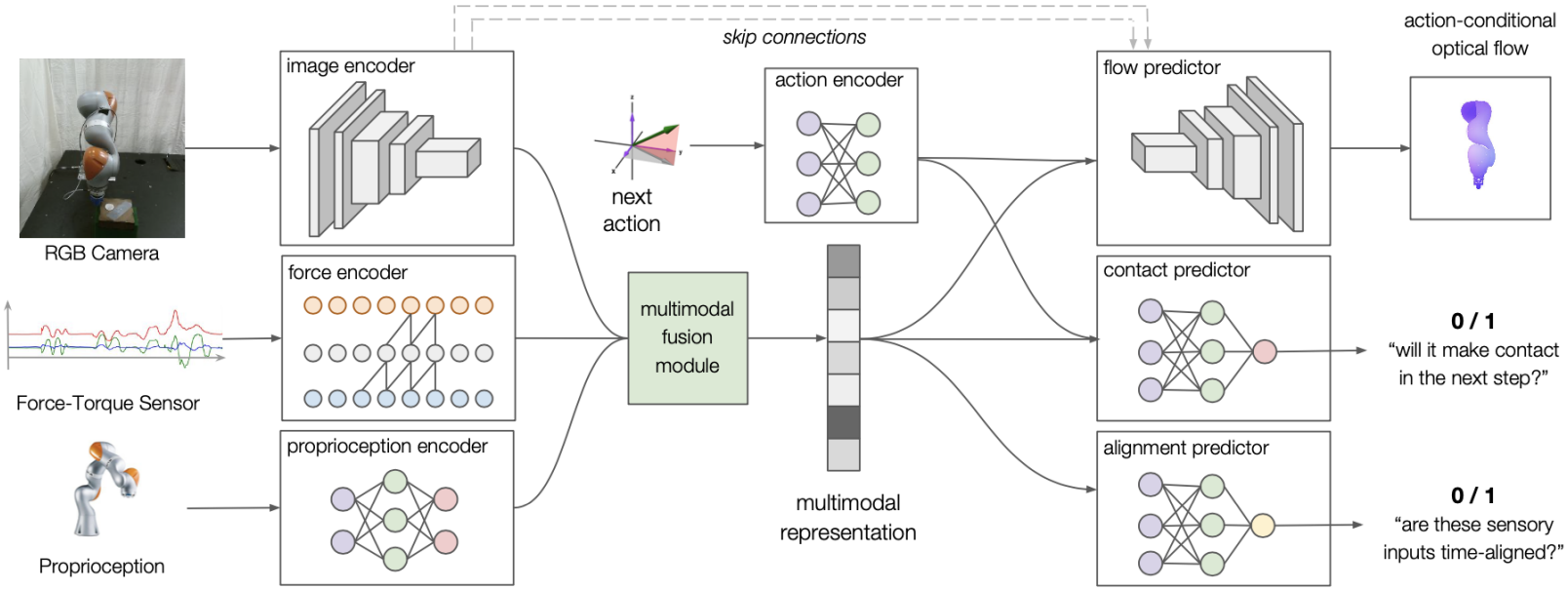
- Se muestrea una trayectoria del *buffer*
- La función de representación h recibe las observaciones de la trayectoria $(o_1, \ldots, o_t)$
- Se desdobla el modelo por K pasos, en cada paso k, la función de la dinámica g recibe el estado oculto s^{k-1} del paso anterior y la acción real a_{t+k}
- Los parámetros de las funciónes de la representación (h), la dinámica (g) y la predicción (f) se entrenan al mismo tiempo para predecir: (i) la política, la función de valor y la recompensa

Resultados

Deep RL y Robótica

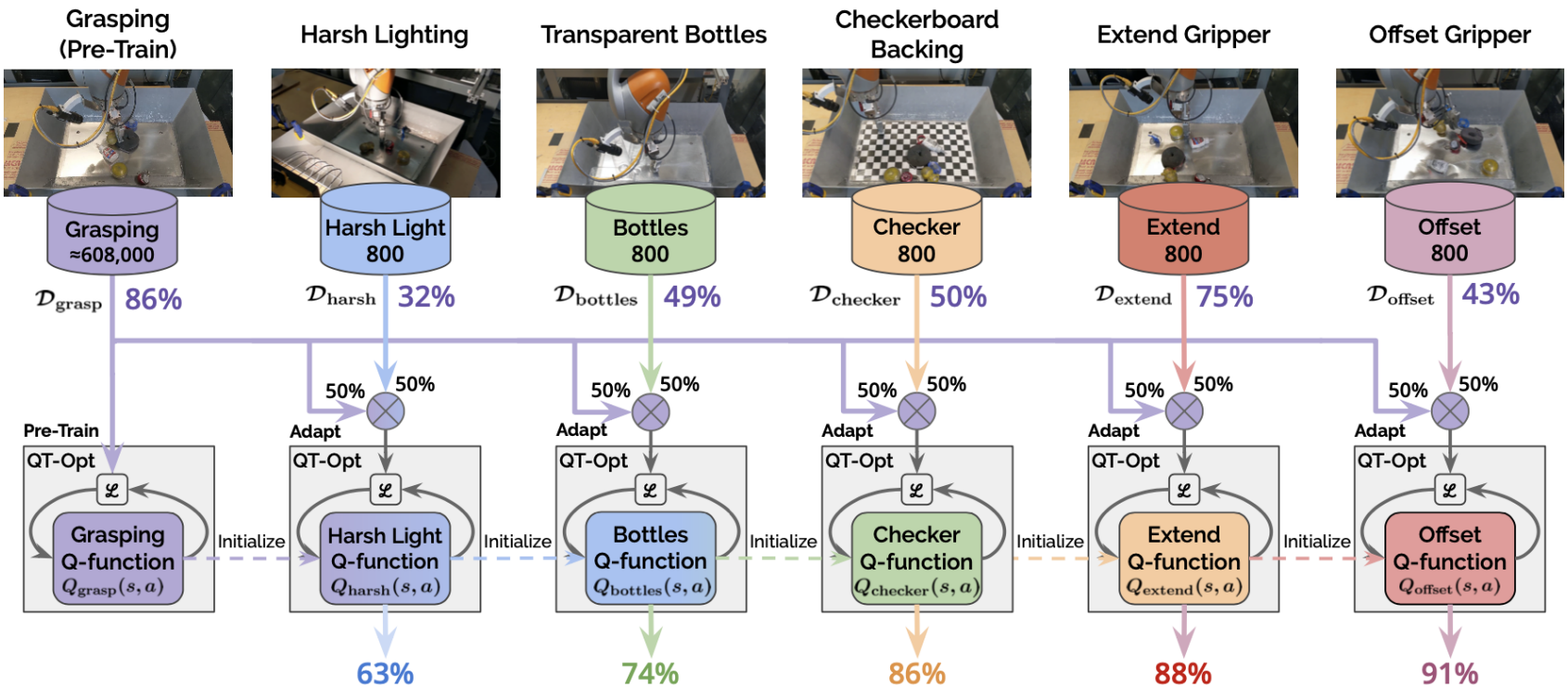
- Además de juegos, la otra área que ha recibido gran atención usando aprendizaje por refuerzo profundo es robótica
- Se ha aplicado a brazos robóticos, robóts móviles, drones, y vehículos autónomos
- Vamos sólo a ilustrar algunos casos

Making sense of vision and touch: Self-supervised learning of multimodal representations for contact-rich tasks



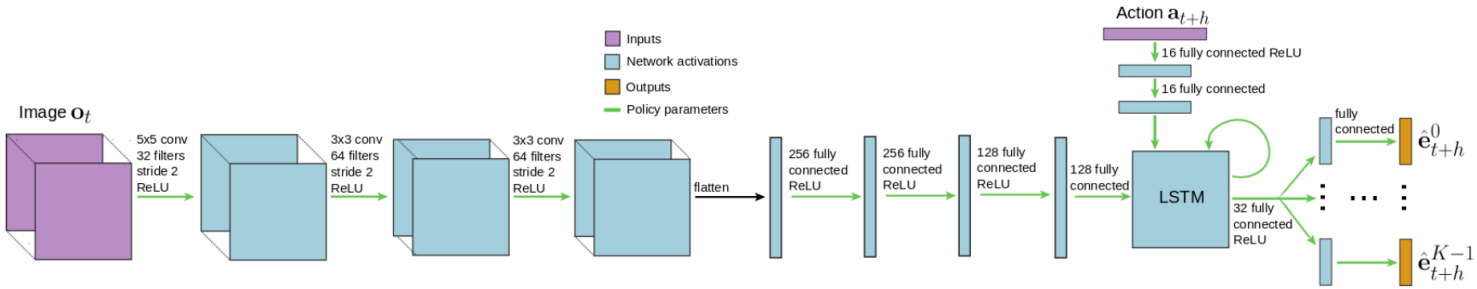
Combina información multimodal para predecir flujo óptico, contacto y alineación

Efficient addaptation for end-to-end vision-based robotic manipulation



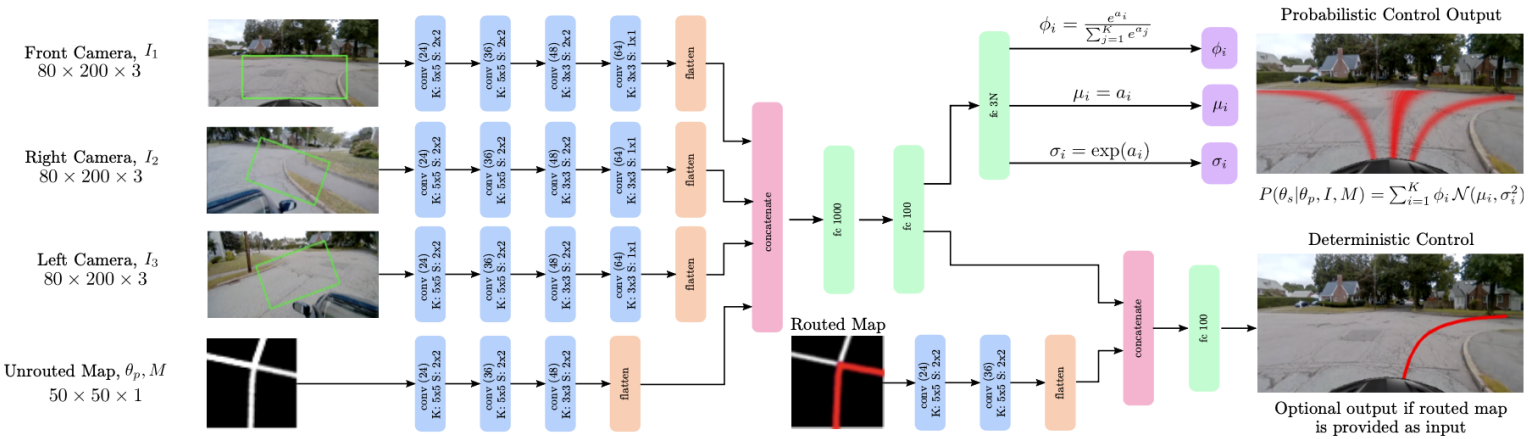
Re-entrena un modelo aprendido para ajustarse a cambios en el ambiente

BADGR: An autonomous self-supervised learning-based navigation system



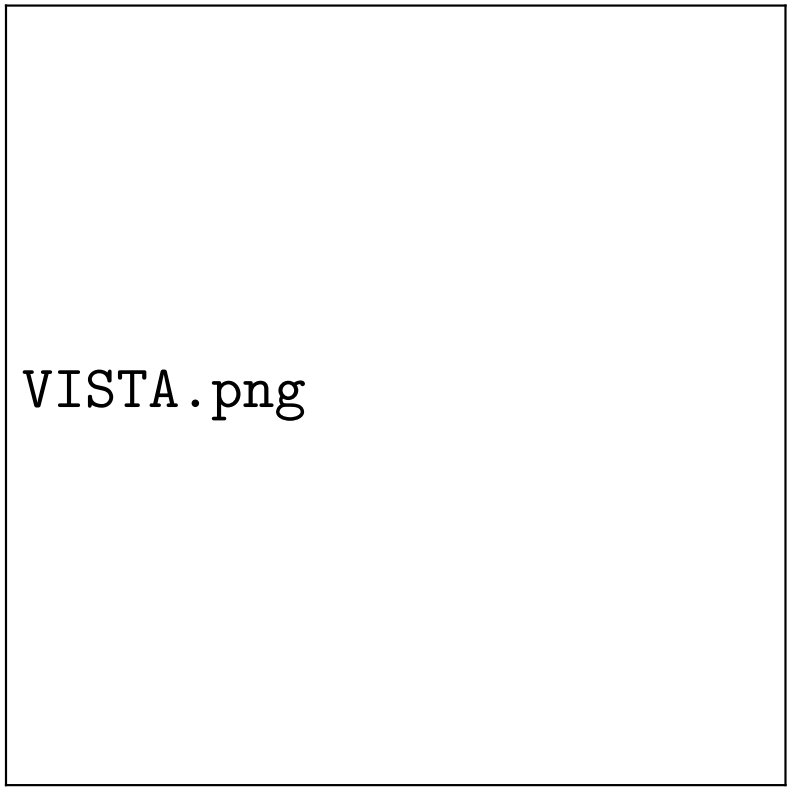
Auto-supervisado, recibe información del ambiente y aprende un modelo predictivo de eventos relevantes

Variational end-to-end navigation and localization



Usanda 3 cámaras y una ruta abstracta decide cómo moverse y/o cómo seguir una ruta

Learning robust control policies for end-to-end autonomous driving from data-driven simulations



Usa un simulador para predecir imágenes/escenarios y poder entrenar un sistema

Deep RL

- El aprendizaje por refuerzo profundo es un área de gran crecimiento
- Las aplicaciones obvias son en juegos y robótica (de todo tipo) con grandes expectativas a futuro
- DRL también se está usando en otras áreas como lenguaje natural
- Al igual que en DL se requieren algoritmos más robustos, menos costosos en términos de datos y recursos computacionales, dar explicaciones de sus resultados, ...
- Se está trabajando en hacer los algoritmos más generales (usando por ejemplo relaciones), que puedan aprender más rápido (usando por ejemplo transferencia), ...

Bibliografía

- **R. Sutton y A. Barto (2018). Reinforcement Learning: An Introduction. MIT Press (2a. edición).**
- Csaba Szepesvari (2010). Algorithms for Reinforcement Learning. Morgan & Claypool Publishers. Synthesis Lectures on Artificial Intelligence and Machine Learning. R.J. Brachman y T.G. Dietterich (editores).
- D.P. Bertsekas, J.N. Tsitsiklis (1997). Neuro-Dynamic Programming. Athena Scientific.

Cursos en Línea

- Material de Sutton y Barto adicional: http://incompleteideas.net/book/the-book-2nd.html
- Reinforcement Learning. D. Silver: http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching.html
- Deep Reinforcement Learning. S. Levine y otros: http://rail.eecs.berkeley.edu/deeprlcourse/

Herramientas y Recursos

- OpenAI Gym:
  1. https://gym.openai.com/
  2. https://github.com/openai/gym
  3. https://towardsdatascience.com/reinforcement-learning-with-openai-d445c2c687d2
  4. https://medium.com/@ashish_fagna/understanding-openai-gym-25c79c06eccb
- Mujoco:
  1. http://www.mujoco.org
  2. http://www.mujoco.org/book/

Herramientas y Recursos

- Stable baselines:
  1. https://pypi.org/project/stable-baselines/
  2. https://github.com/hill-a/stable-baselines
  3. https://stable-baselines.readthedocs.io/en/master/
- Animal AI:
  1. http://animalaiolympics.com/AAI/
  2. https://github.com/beyretb/AnimalAI-Olympics