Capítulo 11

Aprendizaje por Refuerzo

11.1 Introducción

Uno de los enfoques más usados dentro de aprendizaje es el aprendizaje supervisado a partir de ejemplos (pares entradas – salida provistos por el medio ambiente), para después predecir la salida de nuevas entradas.

Cualquier sistema de predicción puede verse dentro de este paradigma, sin embargo, ignora la estructura secuencial del mismo.

En algunos ambientes, muchas veces se puede obtener sólo cierta retroalimentación o recompensa o refuerzo (e.g., gana, pierde).

El refuerzo puede darse en un estado terminal y/o en estados intermedios.

Los refuerzos pueden ser componentes o sugerencias de la utilidad actual a maximizar (e.g., buena movida).

En aprendizaje por refuerzo (RL) el objetivo es aprender cómo mapear situaciones a acciones para maximizar una cierta señal de recompensa.

Promesa: programar agentes mediante premio y castigo sin necesidad de especificar cómo realizar la tarea.

Diferencias con otro tipo de aprendizaje:

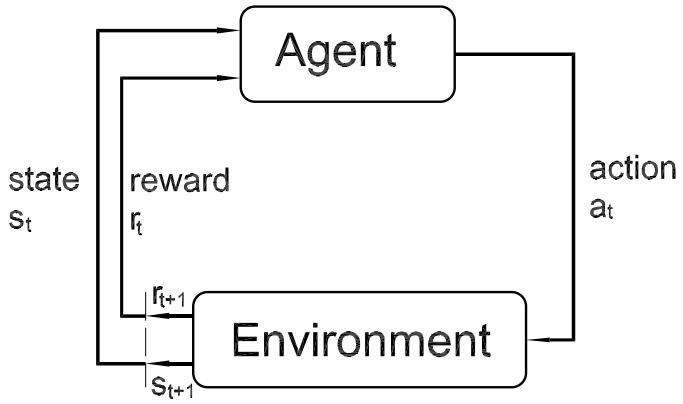

Figura 11.1: Aprendizaje por Refuerzo.

- No se le presentan pares entrada - salida.
- El agente tiene que obtener experiencia útil acerca de los estados, acciones, transiciones y recompensas de manera activa para poder actuar de manera óptima.
- La evaluación del sistema ocurre en forma concurrente con el aprendizaje.

En RL un agente trata de aprender un comportamiento mediante interacciones de prueba y error en un ambiente dinámico e incierto.

En general, al sistema no se le dice qué acción debe tomar, sino que él debe de descubrir qué acciones dan el máximo beneficio.

En un RL estandar, un agente está conectado a un ambiente por medio de percepción y acción (ver figura 11.1). En cada interacción el agente recibe como entrada una indicación de su estado actual ($s \in S$) y selecciona una acción ($a \in A$). La acción cambia el estado y el agente recibe una señal de refuerzo o recompensa ($r \in \mathcal{R}$).

El comportamiento del agente debe de ser tal que escoga acciones que tiendan a incrementar a largo plazo la suma de las recompensas totales.

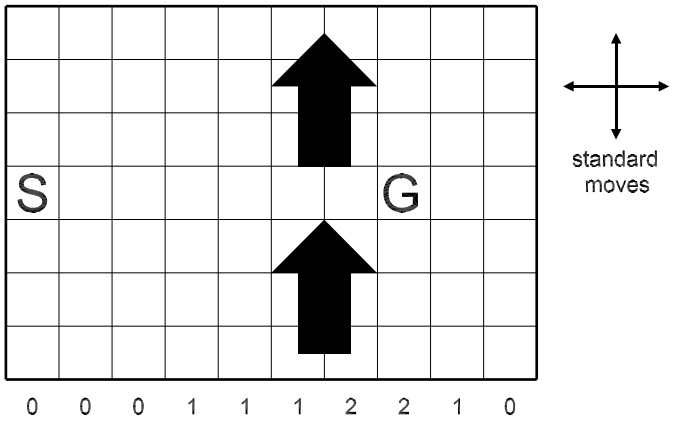

Figura 11.2: Ejemplo de problema.

El objetivo del agente es encontrar una política (π), que mapea estados a acciones que maximice a largo plazo el refuerzo.

En general el ambiente es no-determinístico (tomar la misma acción en el mismo estado puede dar resultados diferentes).

Sin embargo, se asume que el ambiente es estacionario (esto es, las probabilidades de cambio de estado no cambian o cambian muy lentamente).

Aspectos importantes: (i) se sigue un proceso de prueba y error, y (ii) la recompensa puede estar diferida.

Otro aspecto importante es el balance entre exploración y explotación. Para obtener buena ganancia uno prefiere seguir ciertas acciones, pero para saber cuáles, se tiene que hacer cierta exploración. Muchas veces depende de cuánto tiempo se espera que el agente interactue con el medio ambiente.

La caracterización de esta problemática está dada por procesos de decisión de Markov o MDP.

Un MDP modela un problema de decisión sequencial en donde el sistema evoluciona en el tiempo y es controlado por un agente.

La dinámica del sistema esta determinada por una función de transición de probabilidad que mapea estados y acciones a otros estados.

Formalmente, un MDP es una tupla $M =< S, A, \Phi, R >$. Los elementos de un MDP son:

- Un conjunto finito de estados $S(\{1, ..., n\}$.
- Un conjunto finito de acciones A, que pueden depender de cada estado.
- Función de recompensa (R): define la meta. Mapea cada estado–acción a un número (recompensa), indicando lo deseable del estado.
- Modelo del ambiente (opcional): imita el comportamiento del ambiente. Se puede usar para hacer planeación al considerar posibles situaciones futuras basadas en el modelo.

  $\Phi : A \times S \rightarrow \Pi(S)$ es una función de transición de estados dada como una distribución de probabilidad. La probabilidad de alcanzar el estado $s' \in S$ al realizar la acción $a \in A$ en el estado $s \in S$, que se puede denotar como $\Phi(a, s, s')$.
- Política (π): define cómo se comporta el sistema en cierto tiempo. Es un mapeo (a veces estocástico) de los estados a las acciones.
- Función de valor (V): indica lo que es bueno a largo plazo. Es la recompensa total que un agente puede esperar acumular empezando en ese estado (predicciones de recompensas). Se buscan hacer acciones que den los valores más altos, no la recompensa mayor.

  Las recompensas están dadas por el ambiente, pero los valores se deben de estimar (aprender) en base a las observaciones.

  Aprendizaje por refuerzo aprende las funciones de valor mientras interactua con el ambiente.

11.1.1 Modelos de Comportamiento Óptimo

Dado un estado $s_t \in S$ y una acción $a_t \in \mathcal{A}(s_t)$, el agente recibe una recompensa r_{t+1} y se mueve a un nuevo estado s_{t+1}.

El mapeo de estados a probabilidades de seleccionar una acción particular es su política (π_t). Aprendizaje por refuerzo especifica cómo cambiar la política como resultado de su experiencia.

No trata de maximizar la recompensa inmediata, sino la recompensa a largo plazo (acumulada).

La recompensa debe de mostrar lo que queremos obtener y se calcula por el ambiente.

Si las recompensas recibidas después de un tiempo t se denotan como: r_{t+1}, r_{t+2}, r_{t+3}, ..., lo que queremos es maximizar lo que esperamos recibir de recompensa (R_t) que en el caso más simple es:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_T$$

Si se tiene un punto terminal se llaman tareas *episódicas*, si no se tiene se llaman tareas *continuas*. En este último caso, la fórmula de arriba presenta problemas, ya que no podemos hacer el cálculo cuando T no tiene límite.

Podemos usar una forma alternativa en donde se van haciendo cada vez más pequeñas las contribuciones de las recompensas más lejanas:

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

donde γ se conoce como la *razón de descuento* y está entre: $0 \leq \gamma < 1$

Si $\gamma = 0$ se trata sólo de maximizar tomando en cuenta las recompensas inmediatas.

En general, podemos pensar en los siguientes modelos:

1. Horizonte finito: el agente trata de optimizar su recompensa esperada en los siguientes h pasos, sin preocuparse de lo que ocurra despues:

   $$E(\sum_{t=0}^{h} r_t)$$

   donde r_t significa la recompensa recibida t pasos en el futuro.

   Este modelo se puede usar de dos formas: (i) *política no estacionaria*: donde en el primer paso se toman los h siguientes pasos, en el siguiente los $h-1$, etc., hasta terminar. El problema principal es que no siempre se conoce cuántos pasos considerar. (ii) *receding-horizon control*: siempre se toman los siguientes h pasos.

2. Horizonte infinito: las recompensas que recibe un agente son reducidas geométricamente de acuerdo a un factor de descuento γ ($0 \leq \gamma \leq 1$):

$$E(\sum_{t=0}^{\infty} \gamma^t r_t)$$

3. Recompensa promedio: optimizar a largo plazo la recompensa promedio:

$$lim_{h \to \infty} E(\frac{1}{h} \sum_{t=0}^{h} r_t)$$

   Problema: no hay forma de distinguir políticas que reciban grandes recompensas al principio de las que no.

En general, se utiliza la de horizonte infinito.

11.1.2 Recompensa diferida y modelo Markoviano

En general, las acciones del agente determinan, no sólo la recompensa inmediata, sino también (por lo menos en forma probabilística) el siguiente estado del ambiente.

Los problemas con refuerzo diferido se pueden modelar como procesos de decisión de Markov (MDPs).

El modelo es Markoviano si las transiciones de estado no dependen de estados anteriores.

En aprendizaje por refuerzo se asume que se cumple con la propiedad Markoviana y las probabilidades de transición están dadas por:

$$\mathcal{P}^a_{ss'} = Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\}$$

El valor de recompensa esperado es:

$$\mathcal{R}^a_{ss'} = E\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\}$$

Lo que se busca es estimar las funciones de valor. Esto es, qué tan bueno es estar en un estado (o realizar una acción).

La noción de "qué tan bueno" se define en términos de recompensas futuras o recompensas esperadas.

La política π es un mapeo de cada estado $s \in S$ y acción $a \in \mathcal{A}(s)$ a la probabilidad $\pi(s,a)$ de tomar la acción a estando en estado s. El valor de un estado s bajo la política π, denotado como $V^\pi(s)$, es el refuerzo esperado estando en estado s y siguiendo la política π.

Este valor esperado se puede expresar como:

$$V^\pi(s) = E_\pi\{R_t \mid s_t = s\} = E_\pi\left\{\sum_{k=o}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right\}$$

y el valor esperado tomando una acción a en estado s bajo la política π ($Q^\pi(s,a)$):

$$Q^\pi(s,a) = E_\pi\{R_t \mid s_t = s, a_t = a\} = E_\pi\left\{\sum_{k=o}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right\}$$

Las funciones de valor óptimas se definen como:

$$V^*(s) = max_\pi V^\pi(s) \text{ y } Q^*(s,a) = max_\pi Q^\pi(s,a)$$

Las cuales se pueden expresar como las ecuaciones de optimalidad de Bellman:

$$V^*(s) = max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^*(s')]$$

y

$$Q^*(s,a) = \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^*(s')]$$

o

$$Q^*(s,a) = \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma \text{max}_{a'} Q^*(s',a')]$$

11.2 Métodos de Solución de MDPs

Existen tres formas principales de resolver MDPs: (i) usando métodos de programación dinámica, usando métodos de Monte Carlo, y (iii) usando métodos de diferencias temporales o de aprendizaje por refuerzo.

11.2.1 Programación Dinámica

Si se conoce el modelo del ambiente, osea las transiciones de probabilidad ($\mathcal{P}^a_{ss'}$) y los valores esperados de recompensas ($\mathcal{R}^a_{ss'}$), las ecuaciones de optimalidad de Bellman nos representan un sistema de $|S|$ ecuaciones y $|S|$ incognitas.

Consideremos primero como calcular la función de valor V^π dada una política arbitraria π.

$$\begin{aligned} V^\pi(s) &= E_\pi\{R_t \mid s_t = s\} \\ &= E_\pi\left\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \mid s_t = s\right\} \\ &= E_\pi\left\{r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s\right\} \\ &= \textstyle\sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')] \end{aligned}$$

donde $\pi(s,a)$ es la probabilidad de tomar la acción a en estado s bajo la política π.

Podemos hacer aproximaciones sucesivas, evaluando $V_{k+1}(s)$ en términos de $V_k(s)$.

$$V_{k+1}(s) = \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V_k(s')]$$

Podemos entonces definir un algoritmo de evaluación iterativa de políticas como se muestra en la tabla 11.1.

Una de las razones para calcular la función de valor de una política es para tratar de encontrar mejores políticas. Dada una función de valor para una política dada, podemos probar una acción $a \neq \pi(s)$ y ver si su $V(s)$ es mejor o peor que el $V^\pi(s)$.

En lugar de hacer un cambio en un estado y ver el resultado, se pueden considerar cambios en todos los estados considerando todas las acciones de cada estado, seleccionando aquella que parezca mejor de acuerdo a una política *greedy*.

Podemos entonces calcular una nueva política $\pi'(s) = \text{argmax}_a Q^\pi(s,a)$ y continuar hasta que no mejoremos.

Tabla 11.1: Algoritmo iterativo de evaluación de política.

Inicializa $V(s) = 0$ para toda $s \in S$
Repite
    $\Delta \leftarrow 0$
    Para cada $s \in S$
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')$
        $\Delta \leftarrow max(\Delta, |v - V(s)|)$
Hasta que $\Delta < \theta$ (número positivo pequeño)
Regresa $V \approx V^\pi$

Esto sugiere, partir de una política (π_0) y calcular la función de valor (V^{π_0}), con la cual encontrar una mejor política (π_1) y así sucesivamente hasta converger a π^* y V^*.

A este procedimiento se llama iteración de políticas y viene descrito en la tabla 11.2.

Uno de los problemas de iteración de políticas es que cada iteración involucra evaluación de políticas que requiere recorrer todos los estados varias veces.

Sin embargo, el paso de evaluación de política lo podemos truncar de varias formas, sin perder la garantía de convergencia. Una de ellas es pararla después de recorrer una sola vez todos los estados. A esta forma se le llama iteración de valor (*value iteration*). En particular se puede escribir combinando la mejora en la política y la evaluación de la política truncada como sigue:

$$V_{k+1}(s) = max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V_k(s')]$$

Se puede ver como expresar la ecuación de Bellman en una regla de actualización. Es muy parecido a la regla de evaluación de políticas, solo que se evalúa el máximo sobre todas las acciones (ver tabla 11.3).

Para espacios muy grandes, el ver todos los estados puede ser computacionalmente muy caro. Una opción es hacer estas actualizaciones al momento de

Tabla 11.2: Algoritmo de iteración de política.

1. Inicialización:
   $V(s) \in \mathcal{R}$ y $\pi(s) \in \mathcal{A}(s)$ arbitrariamente $\forall s \in S$
2. Evaluación de política:
   Repite
      $\Delta \leftarrow 0$
      Para cada $s \in S$
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)}[\mathcal{R}_{ss'}^{\pi(s)} + \gamma V(s')]$
         $\Delta \leftarrow max(\Delta, |v - V(s)|)$

   Hasta que $\Delta < \theta$ (número positivo pequeño)
3. Mejora de política:
   pol-estable $\leftarrow$ true
   Para cada $s \in S$:
      $b \leftarrow \pi(s)$
      $\pi(s) \leftarrow \text{argmax}_a \sum_{s'} \mathcal{P}_{ss'}^{a}[\mathcal{R}_{ss'}^{a} + \gamma V(s')]$
      if $b \neq \pi$, then *pol-estable* $\leftarrow$ false
   If *pol-estable*, then stop, else go to 2.

Tabla 11.3: Algoritmo de iteración de valor.

Inicializa $V(s) = 0$ para toda $s \in S$
Repite
   $\Delta \leftarrow 0$
   Para cada $s \in S$
      $v \leftarrow V(s)$
      $V(s) \leftarrow max_a \sum_{s'} \mathcal{P}_{ss'}^{a}[\mathcal{R}_{ss'}^{a} + \gamma V^*(s')$
      $\Delta \leftarrow max(\Delta, |v - V(s)|)$

Hasta que $\Delta < \theta$ (número positivo pequeño)
Regresa una política determinística tal que:
   $\pi(s) = argmax_a \sum_{s'} \mathcal{P}_{ss'}^{a}[\mathcal{R}_{ss'}^{a} + \gamma V^*(s')]$

Tabla 11.4: Algoritmo de Monte Carlo para estimar V^{π}.

Repite
Genera un episodio usando π
Para cada estado s en ese episodio:
    $R \leftarrow$ recompensa después de la primera ocurrencia de s
    Añade R a $recomp(s)$
    $V(s) \leftarrow$ promedio($recomp(s)$)

estar explorando el espacio, y por lo tanto determinando sobre qué estados se hacen las actualizaciones.

El hacer estimaciones en base a otras estimaciones se conoce también como *bootstrapping*.

11.2.2 Monte Carlo

Los métodos de Monte Carlo, solo requieren de experiencia y la actualización se hace por episodio más que por cada paso.

El valor de un estado es la recompensa esperada que se puede obtener a partir de ese estado.

Para estimar V^{π} y Q^{π} podemos tomar estadísticas haciendo un promedio de las recompensas obtenidas. El algoritmo para V^{π} está descrito en la tabla 11.4.

Para estimar pares estado-acción (Q^{π}) corremos el peligro de no ver todos los pares, por lo que se busca mantener la exploración. Lo que normalmente se hace es considerar solo políticas estocásticas que tienen una probabilidad diferente de cero se seleccionar todas las acciones.

Con Monte Carlo podemos alternar entre evaluación y mejoras en base a cada episodio. La idea es que después de cada episodio las recompensas observadas se usan para evluar la política y la política se mejora para todos los estados visitados en el episodio. El algoritmo viene descrito en la tabla 11.5.

Tabla 11.5: Algoritmo de Monte Carlo.

Repite
Genera un episodio usando π con exploración
Para cada par s, a en ese episodio:
    $R \leftarrow$ recompensa después de la primera ocurrencia de s, a
    Añade R a $recomp(s, a)$
    $Q(s, a) \leftarrow$ promedio$(recomp(s, a))$
Para cada s en el episodio:
    $\pi(s) \leftarrow \text{argmax}_a Q(s, a)$

Existen dos formas para asegurar que todas las acciones pueden ser seleccionadas indefinidamente:

- Los algoritmos *on-policy*: Estiman el valor de la política mientras la usan para el control. Se trata de mejorar la política que se usa para tomar decisiones.
- Los algoritmos *off-policy*: Usan la política y el control en forma separada. La estimación de la política puede ser por ejemplo *greedy* y la política de comportamiento puede ser ϵ-*greedy*. Osea que la política de comportamiento está separada de la política que se quiere mejorar.

  Esto es lo que hace Q-learning, lo cual simplifica el algoritmo.

Ejemplos de políticas de selección de acciones son:

- $\epsilon-$*greedy*: en donde la mayor parte del tiempo se selecciona la acción que da el mayor valor estimado, pero con probabilidad ϵ se selecciona una acción aleatoriamente.
- *softmax*, en donde la probabilidad de selección de cada acción depende de su valor estimado. La más común sigue una distribución de Boltzmann o de Gibbs, y selecciona una acción con la siguiente probabilidad:

$$\frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(b)/\tau}}$$

donde τ es un parámetro positivo (temperatura).

11.2.3 Diferencias Temporales (*Temporal Difference*)

Los métodos de TD combinan las ventajas de los dos anteriores: permite hacer *bootstrapping* (como DP) y no requiere tener un modelo del ambiente (como MC).

Métodos tipo TD sólo tienen que esperar el siguiente paso.

TD usan el error o diferencia entre predicciones sucesivas (en lugar del error entre la predicción y la salida final) aprendiendo al existir cambios entre predicciones sucesivas.

Ventajas:

- Incrementales y por lo tanto fáciles de computar.
- Convergen más rápido con mejores predicciones.

El más simple TD(0) es:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right]$$

El algoritmo de TD(0) viene descrito en la tabla 11.6.

La actualización de valores tomando en cuenta la acción sería:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

y el algoritmo es prácticamente el mismo, solo que se llama SARSA, y viene descrito en la tabla 11.7.

Uno de los desarrollos más importantes en aprendizaje por refuerzo fué el desarrollo de un algoritmo "fuera-de-política" (*off-policy*) conocido como Q-learning.

Tabla 11.6: Algoritmo TD(0).

Inicializa $V(s)$ arbitrariamente y π a la política a evaluar
Repite (para cada episodio):
    Inicializa s
    Repite (para cada paso del episodio):
        $a \leftarrow$ acción dada por π para s
        Realiza acción a; observa la recompensa, r, y el siguiente estado, s'
        $V(s) \leftarrow V(s) + \alpha\left[r + \gamma V(s') - V(s)\right]$
        $s \leftarrow s'$
    hasta que s sea terminal

Tabla 11.7: Algoritmo SARSA.

Inicializa $Q(s, a)$ arbitrariamente
Repite (para cada episodio):
    Inicializa s
    Selecciona una a a partir de s usando la política dada por Q
        (e.g., ϵ–greedy)
    Repite (para cada paso del episodio):
        Realiza acción a, observa r, s'
        Escoge a' de s' usando la política derivada de Q
        $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma Q(s', a') - Q(s, a)\right]$
        $s \leftarrow s'$; $a \leftarrow a'$;
    hasta que s sea terminal

Tabla 11.8: Algoritmo Q-Learning.

Inicializa $Q(s, a)$ arbitrariamente
Repite (para cada episodio):
    Inicializa s
    Repite (para cada paso del episodio):
        Selecciona una a de s usando la política dada por Q
        (e.g., ϵ–greedy)
        Realiza acción a, observa r, s'
        $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma max'_a Q(s', a') - Q(s, a)\right]$
        $s \leftarrow s'$;
    hasta que s sea terminal

La idea principal es realizar la actualización de la siguiente forma (Watkins, 89):

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

El algoritmo viene descrito en la tabla 11.8.

11.3 Trazas de Elegibilidad (*eligibility traces*)

Están entre métodos de Monte Carlo y TD de un paso.

Los métodos Monte Carlo realizan la actualización considerando la secuencia completa de recompensas observadas.

La actualización de los métodos de TD la hacen utilizando únicamente la siguiente recompensa.

La idea de las trazas de elegibilidad es considerar las recompensas de n estados posteriores (o afectar a n anteriores).

Si recordamos:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{T-t-1} r_T$$

Lo que se hace en TD es usar:

$$R_t = r_{t+1} + \gamma V_t(s_{t+1})$$

lo cual hace sentido porque $V_t(s_{t+1})$ reemplaza a los términos siguientes $(\gamma r_{t+2} + \gamma^2 r_{t+3} \ldots)$.

Sin embargo, hace igual sentido hacer:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_t(s_{t+2})$$

y, en general, para n pasos en el futuro.

En la práctica, más que esperar n pasos para actualizar (*forward view*), se realiza al revés (*backward view*). Se guarda información sobre los estados por los que se pasó y se actualizan hacia atrás las recompensas (descontadas por la distancia). Se puede probar que ambos enfoques son equivalentes.

Para implementar la idea anterior, se asocia a cada estado o par estado-acción una variable extra, representando su traza de elegibilidad (*eligibility trace*) que denotaremos por $e_t(s)$ o $e_t(s, a)$.

Este valor va decayendo con la longitud de la traza creada en cada episodio. La figura 11.3 muestra este comportamiento.

Para $TD(\lambda)$:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{si } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{si } s = s_t \end{cases}$$

Para SARSA se tiene lo siguiente:

$$e_t(s, a) = \begin{cases} \gamma\lambda e_{t-1}(s, a) & \text{si } s \neq s_t \\ \gamma\lambda e_{t-1}(s, a) + 1 & \text{si } s = s_t \end{cases}$$

El algoritmo para $SARSA(\lambda)$ viene descrito en la tabla 11.9.

Para Q-learning como la selección de acciones se hace, por ejemplo, siguiendo una política $\epsilon-greedy$, se tiene que tener cuidado, ya que a veces los movimientos, son movimientos exploratorios.

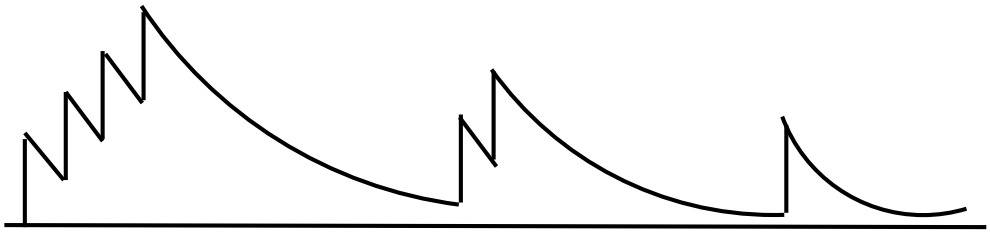

Figura 11.3: Comportamiento de las trazas de elegibilidad.

Tabla 11.9: $SARSA(\lambda)$ con trazas de elegibilidad.

Inicializa $Q(s,a)$ arbitrariamente y $e(s,a) = 0\ \forall s, a$
Repite (para cada episodio)
    Inicializa s, a
    Repite (para cada paso en el episodeo)
        Toma acción a y observa r, s'
        Selecciona a' de s' usando una política derivada de Q (e.g., $\epsilon-greedy$)
        $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
        $e(s, a) \leftarrow e(s, a) + 1$
        Para todos s, a
            $Q(s, a) \leftarrow Q(s, a) + \alpha\delta e(s, a)$
            $e(s, a) \leftarrow \gamma\lambda e(s, a)$
        $s \leftarrow s'; a \leftarrow a'$
    hasta que s sea terminal

Aquí se puede mantener historia de la traza solo hasta el primer movimiento exploratorio, ignorar las acciones exploratorias, o hacer un esquema un poco más complicado que considera todas las posibles acciones en cada estado.

11.4 Planeación y Aprendizaje

Asumamos que tenemos un modelo del ambiente, esto es, que podemos predecir el siguiente estado y la recomepensa dado un estado y una acción.

La predicción puede ser un conjunto de posibles estados con su probabilidad asociada o pouede ser un estado que es muestreado de acuerdo a la distribución de probabilidad de los estados resultantes.

Dado un modelo, es posible hacer planificación. Lo interesante es que podemos utilizar los estados y acciones utilizados en la planificación también para aprender. De hecho al sistema de aprendizaje no le importa si los pares estado-acción son dados de experiencias reales o simuladas.

Dado un modelo del ambiente, uno podría seleccionar aleatoriamente un par estad-acción, usar el modelo para predecir el siguiente estado, obtener una recompensa y actualizar valores Q. Esto se puede repetir indefinidamente hasta converger a Q^*.

El algoritmo Dyna-Q combina experiencias con planificación para aprender más rápidamente una política óptima.

La idea es aprender de experiencia, pero también usar un modelo para simular experiencia adicional y así aprender más rápidamente (ver tabla 11.10).

El algoritmo de Dyna-Q selecciona pares estado-acción aleatoriamente de pares anteriores. Sin embargo, la planificación se puede usar mucho mejor si se enfoca a pares estado-acción específicos.

Por ejemplo, enfocarnos en las metas e irnos hacia atrás o más generalmente, irnos hacia atrás de cualquer estado que cambie su valor.

Los cambios en las estimaciones de valor V o Q pueden cambiar, cuando se está aprendiendo o si el ambiente cambia y un valor estimado deja de ser

Tabla 11.10: Algoritmo de Dyna-Q.

Inicializa $Q(s,a)$ y $Modelo(s,a)$ $\forall s \in S, a \in A$
DO forever
$s \leftarrow$ estado actual
$a \leftarrow \epsilon-greedy(s,a)$
reaiza acción a onserva s' y r
$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
$Modelo(s,a) \leftarrow s', r$
Repite N veces:
$s \leftarrow$ estado anterior seleccionado aleatoriamente
$a \leftarrow$ acción aleatoria tomada en s
$s', r \leftarrow Modelo(s,a)$
$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$

cierto.

Lo que se puede hacer es enfocar la simulación al estado que cambio su valor. Esto nos lleva a todos los estados que llegan a ese estado y que también cambiarían su valor.

Esto proceso se puede repetir sucesivamente, sin embargo, algunos estados cambian mucho más que otros. Lo que podemos hacer es ordenarlos y cambiar solo los que rebacen un cierto umbral. Esto es precisamente lo que hacer el algoritmo de *prioritized sweeping* (ver tabla 11.11).

11.5 Generalización en Aprendizaje por Refuerzo

Hasta ahora hemos asumido que se tiene una representación explícita en forma de tabla (i.e., una salida por cada tupla de entradas). Esto funciona para epacios pequeños, pero es impensable para dominios como ajedrez (10^{120}) o backgammon (10^{50}).

Tabla 11.11: Algoritmo de Prioritized sweeping.

Inicializa $Q(s,a)$ y $Modelo(s,a)$ $\forall s \in S, a \in A$ y $ColaP = \emptyset$
DO forever
    $s \leftarrow$ estado actual
    $a \leftarrow \epsilon - greedy(s,a)$
    reaiza acción a onserva s' y r
    $Modelo(s,a) \leftarrow s', r$
    $p \leftarrow\mid r + \gamma \max_{a'} Q(s',a') - Q(s,a) \mid$
    if $p > \theta$, then inserta s, a a $ColaP$ con prioridad p
    Repite N veces, mientras $ColaP \neq \emptyset$:
        $s, a \leftarrow$ primero$(ColaP)$
        $s', r \leftarrow Modelo(s,a)$
        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
        Repite $\forall \overline{s}, \overline{a}$ que se predice llegan a s:
            $\overline{r} \leftarrow$ recomensa predicha
            $p \leftarrow\mid \overline{r} + \gamma \max_{a} Q(s,a) - Q(\overline{s}, \overline{a}) \mid$
            if $p > \theta$, then inserta $\overline{s}, \overline{a}$ a $ColaP$ con prioridad p

Una forma de hacerlo es con 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.

La compresión lograda por una representación implícita permite al sistema de aprendizaje, generalizar de estados visitados a estados no visitados.

Por otro lado, puede que no exista tal función. Como en todos los sistemas de aprendizaje, existe un balance entre el espacio de hipótesis y el tiempo que toma aprender una hipótesis aceptable.

Muchos sistemas de aprendizaje supervisado tratan de minimizar el error cuadrado (MSE) bajo cierta distribución P de las entradas.

Si $\vec{\Theta}_t$ representa el vector de parámetros de la función parametrizada que queremos aprender:

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

donde $P(s)$ es una distribución pesando los errores de diferentes estados.

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:

$$\begin{aligned} \vec{\Theta}_{t+1} &= \vec{\Theta}_t - \tfrac{1}{2}\alpha \nabla_{\vec{\Theta}_t}[V^{\pi(s_t)} - V_t(s_t)]^2 \\ &= \vec{\Theta}_t - \alpha[V^{\pi(s_t)} - V_t(s_t)]\nabla_{\vec{\Theta}_t} V_t(s_t) \end{aligned}$$

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

Como no sabemos $V^{\pi(s_t)}$ lo tenemos que aproximar. Podemos hacerlo con trazas de elegibilidad y actualizar la función Θ como sigue:

$$\vec{\Theta}_{t+1} = \vec{\Theta}_t + \alpha \delta_t \vec{e}_t$$

donde δ_t es el error:

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)$$

y $\vec{e}_t$ es un vector de trazas de elegibilidad, una por cada componente de $\vec{\Theta}_t$, que se actualiza como:

$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\Theta}_t} V_t(s_t)$$

con $\vec{e}_0 = 0$.

11.6 Aplicaciones a Juegos y Control

La primera aplicación en aprendizaje por refuerzo fué el programa para jugar damas de Samuel. Usó una función lineal de evaluación con pesos usando hasta 16 términos. Su programa era parecido a la ecuación de actualización de pesos, pero no usaba recompensa en los estados terminales. Esto hace que puede o no converger y puede aprender a perder.

Logró evitar ésto haciendo que el peso para ganancia de material fuera siempre positivo.

Se han hecho aplicaciones a control de robots. Una de las más conocidas es el control del péndulo invertido. Controlar la posición x para que se mantenga aproximadamente derecho ($\theta \approx \pi/2$), manteniendose en los límites de la pista. $X, \theta, \dot{X}$ y $\dot{\theta}$ son continuas. El control es de tipo bang–bang.

Boxes (Michie, Chambers '68) balanceaba el pendulo por más de una hora después de 30 intentos (no simulado). Idea: discretizar el espacio en cajas. Se corria el sistema hasta que se caía el péndulo o se salia de los límites. Entonces se daba un refuerzo negativo a la última "caja" y se propagaba a la secuencia de "cajas" por las que pasó. Sin embargo, los resultados más impresionantes (un péndulo invertido triple) se lograron derivando un algoritmo con teoría de control clásica (simulado).

TD-gammon (Tesauro '92) ilustra la potencialidad de técnicas de aprendizaje por refuerzo. Tesauro primero trató de aprender $Q(s, a)$ directamente con una red neuronal (Neurogammon) con poco éxito. Después representó una

función de evaluación con una sola capa intermedia con 40 nodos. Después de 200,000 juegos de entrenamiento mejoró notablemente su desempeño. Añadiendo atributos adicionales a una red con 80 nodos escondidos, después de 300,000 juegos de entrenamiento, juega como los 3 mejores jugadores del mundo.

Recientemente (2000), se desarrolló un algoritmo de RL que actualiza las funciones de evaluación en un árbol de búsqueda en juegos (TDLeaf(λ).

Aplicado a ajedrez, mejora el puntaje de un programa (KnightCap) de 1,650 a 2,150 después de 308 juegos en 3 días.

11.7 Algunos desarrollos recientes

Uno de los problemas principales de las técnicas usadas en aprendizaje por refuerzo, y para resolver MDP en general, es la aplicación a espacios grandes (muchos estados y acciones).

Aunque el algoritmo converge en teoría, en la práctica puede tomar un tiempo inaceptable.

Dentro de los enfoques que atacan, en parte, esta problemática, podemos mencionar:

- Agregación de estados, en donde se juntan estados "parecidos" y a todos ellos se les asigna el mismo valor, reduciendo con esto el espacio de estados. Algunos ejemplos de esto son: tile-coding, coarse coding, radial basis functions, Kanerva coding, y soft-state aggregation.
- Abstracciones basadas en máquinas de estado finito, en donde el aprendizaje por refuerzo tiene que decidir que máquina utilizar (por ejemplo, HAM y PHAM).
- Definición de jerarquías, en donde se divide el espacio en subproblemas, se aprenden políticas a los espacios de más bajo nivel y estas se usan para resolver problemas de más alto nivel (e.g., MAXQ, HEXQ). Algo parecido se usa con Macros y Options, en donde se aprenden políticas de subespacios que se usan para resolver problemas mas grandes.

- Otra opción es utilizar un sistema de planificación que decida la secuencias de submetas que se tienen que cumplir para resolver cierto problema (por ejemplo usando TOPs) y después aprender por aprendizaje por refuerzo las acciones a realizar para resolver cada submeta (e.g., RL-TOP).

- También se ha buscado utilizar representaciones relacionales dentro de aprendizaje por refuerzo, ya sea para representar las funciones de valor y/o para representar los estados y las acciones.

- También se han utilizado soluciones conocidas como guías o trazas que se usan para aprender más rápidamente las funciones de valor o para aprender un subconjunto de acciones relevantes.