

CS285: Deep Reinforcement Learning (2019)

Ramón Calvo González

7 de agosto de 2020

Tema 1

Course Introduction, Imitation Learning

Clase 1: Introduction and Course Overview

2020-06-10

1.1. ¿Qué es RL y por qué es importante?

Las máquinas inteligentes tienen que ser capaces de adaptarse y ser flexibles. El Deep Learning es muy útil para resolver problemas en entornos poco estructurados, pero con la necesidad de entrenar modelos grandes y complejos. Casi todas las tareas que resuelve el Deep Learning tienen como características que son de 'mundo abierto', son diversas y variadas (por ejemplo en NLP las reglas de los lenguajes son demasiadas como para escribirlas a mano).

RL provee de un formalismo matemático para gestionar la toma de decisiones. Una de las primera historias de éxito del RL fue la aplicación de redes neuronales para estimar el valor de los estados del juego Backgammon. Actualmente también se aplica en robótica.

Para entender mejor porque juntar RL con DL, se puede mirar al campo de la visión artificial. Al principio los pipelines eran escritos a manos y cada capa del pipeline requerían varios doctorados para ser desarrolladas. Con DL, todo este proceso queda automatizado por una red neuronal y superando con creces los resultados iniciales.

En el pasado se tenían los mismos problemas con el RL. Las características que representaban los estados tenían que cogerse a mano y consultar con expertos para saber cuáles de ellas eran más importantes a la hora de definir el valor o la política. La idea del DRL es que este proceso también queda encapsulado por una red neuronal.

1.2. ¿Que otros problemas itenen que resolvverse para habilitar un proceso de toma de decisiones en el mundo real?

- Ir más allá de la recompensa: como por ejemplo aprender la función de recompensa en sí después de haber visto a un 'profesor' (aprendizaje por refuerzo inverso).
- Transferir el conocimiento entre varios dominios.
- Aprender a predecir.

1.3. ¿De donde vienen las recompensas?

En la vida real no se tienen recompensas. Pueden haber entornos muy complejos en los que nunca se llegue a una recompensa, por lo que nunca se aprende.

En el cerebro humano, las recompensas vienen de los ganglios basales, los cuales ocupan una parte importante del cerebro. Lo que quiere decir que las funciones de recompensa no son algo sencillo, como si fueran un interruptor que se activa cuando se hace algo bien.

También las personas son buenas en saber que recompensas reciben otros individuos cuando hacen ciertas acciones y tienen ciertos objetivos.

Hay evidencia de que el razonamiento inteligente humano viene de las predicciones que hacemos del mundo.

1.4. ¿Qué se puede hacer con un modelo perfecto?

Si se aprende un modelo completo del mundo se pueden conseguir políticas muy complejas.

1.5. ¿Cómo se crean las máquinas inteligentes?

Se podría partir a partir de una máquina inteligente ya existente como la mente humana. Se podría intentar modelar de forma sencilla cada parte del cerebro. Desafortunadamente esto es extraordinariamente complejo.

Se puede hacer el proceso más sencillo partiendo de la hipótesis de que el aprendizaje es la base de la inteligencia. Por lo que se podría por ejemplo programar varias partes de un cerebro virtual para que cada parte aprendiese de su dominio.

Existe la hipótesis de que solo hay un algoritmo o unos pocos que son la base de la inteligencia. Hay varias evidencias a favor de esta hipótesis: como la flexibilidad del cerebro de los animales (p.e. visión a través de la lengua).

1.6. ¿Qué tendría que hacer una algoritmo así?

Tendría que interpretar entradas sensoriales muy ricas en información y elegir entre acciones muy complejas.

1.7. ¿Por qué usar DRL?

- Deep = puede procesar entradas sensoriales complejas (y también computar funciones complejas)
- RL: puede elegir entre acciones complejas.

Evidencias a favor del DL en la publicación: *Unsupervised learning models of primary cortical receptive fields and receptive field plasticity*. Se compara DL con las redes neuronales de animales en tareas de visión, audición y tacto y estadísticamente hay grandes coincidencias.

También hay evidencias de RL en el cerebro. (Publicación: *Reinforcement learning in the brain*).

1.8. ¿Qué puede hacer bien actualmente DRL?

- Adquirir un cierto grado de proficiencia en entornos gobernados por reglas conocidas y sencillas.
- Aprender habilidades a partir de experiencia

- Imitar el comportamiento de otros agentes.

1.9. ¿Cuáles son actualmente los desafíos de DRL?

Los métodos DRL aprenden muy lento comparado con los seres humanos. Además los humanos también podemos usar conocimiento acumulado.

También, no se tiene una noción clara de cual debería ser la función de recompensa ni cual debería ser el rol de la predicción (model free vs model based).

Clase 2: Supervised Learning of Behaviours

2020-06-11

1.10. Terminología y notación

Las observaciones serán llamadas o , las acciones a , y la distribución que calcula las acciones según las observaciones se llamará $\pi_\theta(a|o)$. Donde θ son los parámetros de la función que se tienen que aprender.

Para hacer referencia a que estos valores son secuenciales en el tiempo, se les añade t y o el subíndice $t : a_t, o_t$.

Debajo de las observaciones están los estados. Puede ser que las observaciones coincidan con el estado del entorno, pero también puede ser que la observación contenga sólo información incompleta del estado del entorno.

Por ejemplo en un robot que tenga una cámara, los píxeles que lee la cámara son una consecuencia del estado de las posiciones de todos los objetos del entorno.

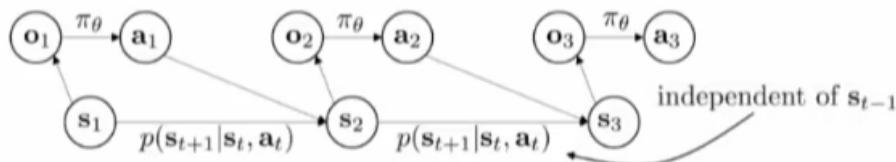


Figura 1.1: Si se observa s_1 , s_3 es totalmente independiente de s_1 (Propiedad de Markov). Esto no es cierto en las observaciones. Por ejemplo si se conoce o_2 , conocimiento de o_1 también puede ayudar a tomar una mejor decisión.

Los nombre s_t y a_t vienen de Richard Bellman, mientras que en teoría de control, sus equivalentes x_t (estado) y u_t (acción) viene del ruso.

1.11. Imitation Learning

Si se tiene por ejemplo la tarea de conducción automática, se puede crear un dataset de observaciones con las acciones de agentes humanos conduciendo y entrenar una política parametrizada por una red neuronal para que aprenda.

Generalmente, si se hace lo que se acaba de explicar, no funciona. Diferentes cosas pueden ir mal:

- Situaciones nunca vistas
- Los agentes humanos hacen acciones erróneas
- Los agentes humanos pueden tomar acciones basándose en información del pasado.
- Se tienen los mismos problemas que en problemas comunes de aprendizaje supervisado.

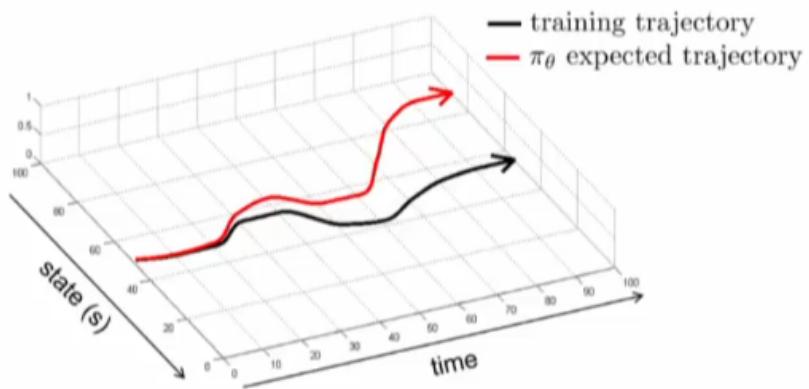
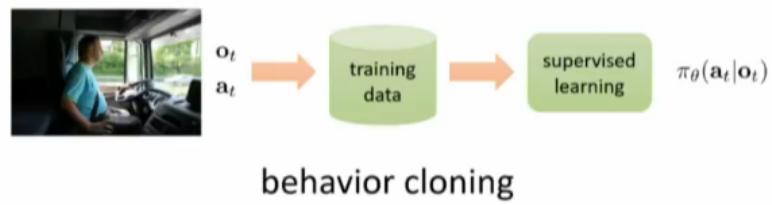
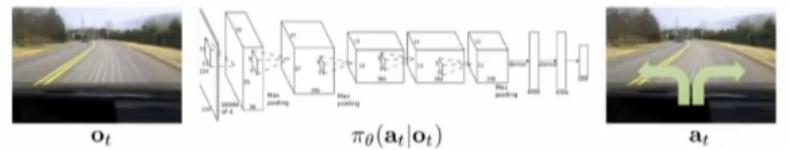


Figura 1.2: Si se considera una tarea con un estado unidimensional y se entrena un modelo para que siga la trayectoria pintada en negro, el algoritmo cometerá errores pequeños que lo irán alejando cada vez más del conjunto de entrenamiento. Esto hará que cada vez los estados que vea sean más distintos de los vistos en el entrenamiento, por lo que se cometerán errores cada vez más grandes y se divergerá de la política deseada.

En la práctica, sorprendentemente funciona más o menos (por ejemplo el trabajo de conducción autónoma de NVIDIA en 2016).

¿Por qué funciona? Hay dos formas de estabilizar el modelo:

- Se añaden tres cámaras: frontal y dos ligeramente a los lados. La frontal se entrena normal pero las laterales se entrenan con las acciones modificadas para que se gire el volante en la dirección que centre el coche en la trayectoria.
- Se puede introducir ruido estadístico para crear varias trayectorias parecidas. Lo que evita que el agente diverja en cuanto se aleje de la trayectoria original.

El problema de Imitation Learning viene de que $p_{data}(o_t) = p_{\pi_\theta}(o_t)$. Lo que se puede hacer es intentar modificar p_{data} para que se parezca a la política perfecta. Esta es la idea detrás de DAgger. Idealmente este algoritmo garantiza la solución perfecta dado un tiempo infinito. En la práctica evidentemente esto no es así.

Algoritmo 1: DAgger: Dataset Aggregation

mientras iteraciones deseadas hacer

- | Entrenar $\pi_\theta(a_t|o_t)$ a partir de datos humanos $D = \{o_1, a_1, \dots, o_N, a_N\}$
- | Ejecutar $\pi_\theta(a_t|o_t)$ para obtener el dataset $D_\pi = \{o_1, \dots, o_M\}$
- | Pedir a un humano que decida la acción de cada observación de D_π
- | Agregar: $D \leftarrow D \cup D_\pi$

fin

Se podría empezar con una política aleatoria en el principio y también se mantienen las garantías de convergencia.

Este algoritmo tiene varios problemas:

- Se requieren humanos.
- Los humanos puede que no elijan las mejores acciones ya que nos somos Markonianos. El mundo no es estático como un tablero de ajedrez.

1.11.1. ¿Se puede hacer trabajar Imitation Learning con menos datos?

- DAgger resuelve el problema de la deriva distribucional.
- Si se tiene un modelo que es tan bueno que no falla se puede eliminar casi completamente esta deriva.
- Se tiene que mimetizar comportamientos expertos muy precisamente.
- No vale con sobreentrenar.

1.11.2. ¿Por qué podemos fallar al entrenar el experto?

- Los humanos no tenemos un comportamiento Markoviano.
- Los humanos tenemos un comportamiento multimodal (ánimo, zurdo/diestro, ...).

Para arreglar lo primero, se puede entrenar una red neuronal que dependa de todas las observaciones hasta el momento. Por ejemplo una red neuronal recurrente.

Para tratar con lo segundo primero es mejor verlo con un ejemplo. Si queremos rodear un árbol, podemos elegir por la izquierda o por la derecha ya que ambas son igual de razonables. Pero una política que sea la media de ambas no sería razonable. Las acciones multimodales en algunos casos no son un problema. Por ejemplo en el caso de acciones discretas, que se puede seleccionar con una

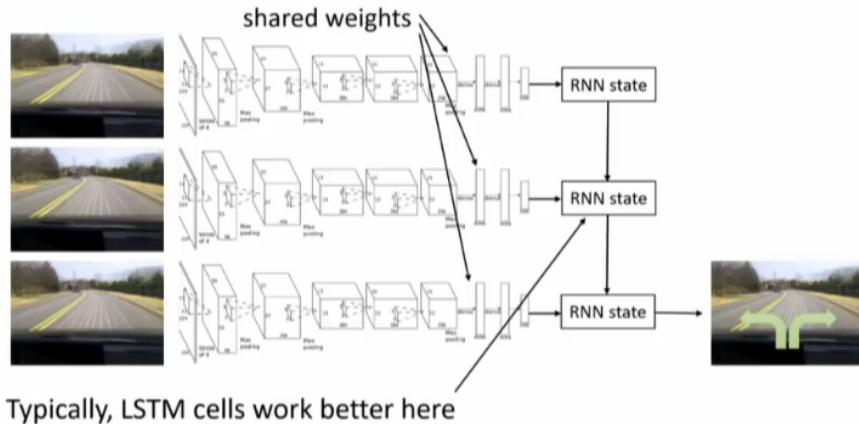


Figura 1.3: Esquema básico de un controlador con redes recurrentes.

softmax. En el ámbito continuo esto es más complejo. Actualmente se suelen usar distribuciones gaussianas y se entran con MSE. Hay varias opciones:

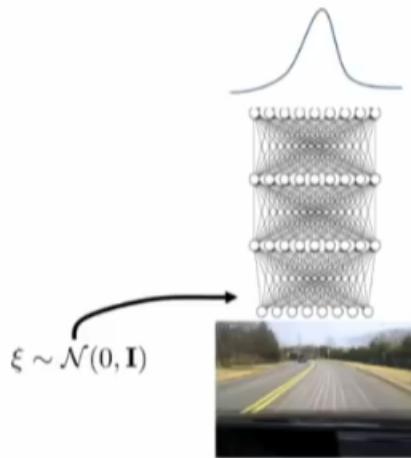
- Mezclas de gaussianas. La más sencilla pero menos expresiva.
- Latent variable models.
- Autoreressive discretization.

Mezclas de gaussianas

Se usan Mixture Density Networks. Las redes neuronales aprenden los parámetros de las distribuciones gaussianas. Esto funciona bien con dimensiones bajas.

Latent Variable Models

Es la solución más compleja pero la más general. No se cambia la salida de la red. Lo que se hace es inyectar una entrada adicional al modelo (como por ejemplo un número aleatorio a partir de una distribución gaussiana). Lo que se pretende es que la red neuronal aprenda a escoger una de las acciones multimodales a partir de esta entrada inyectada, y es precisamente donde reside la dificultad de este método.

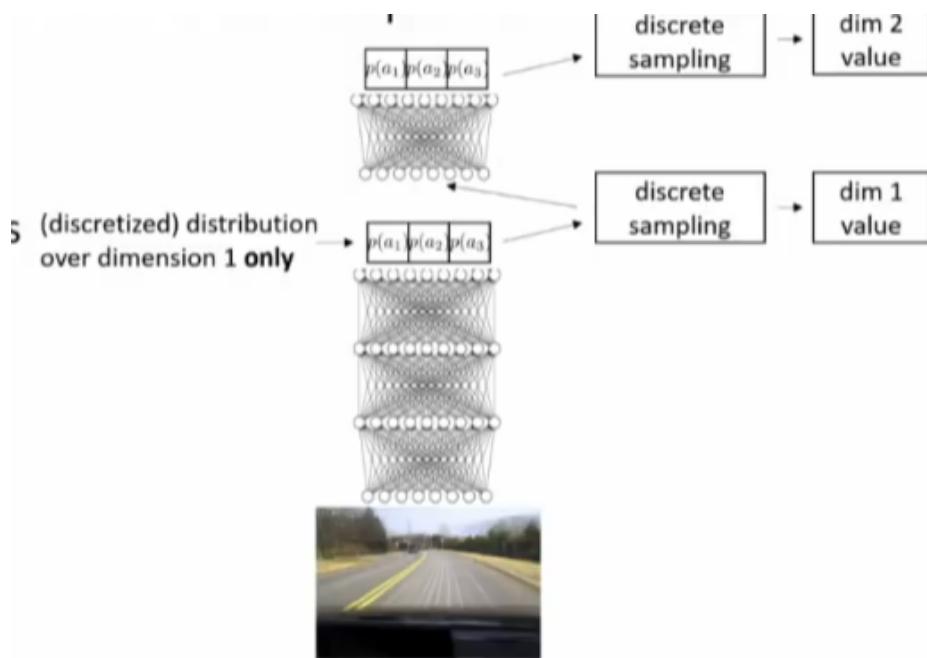


Para solventar este problema, existen las siguientes aproximaciones:

- Conditional Variational Autoencoder
- Normalizing flow / real NVP
- Stein variational gradient descent

Autoregressive discretization

Se pretende discretizar el espacio de las acciones. Como no es posible simplemente asignar celdas ya que se tiene el problema de la maldición de la dimensionalidad, lo que se hace es entrenar la red para que saque la distribución de probabilidades de todas las celdas discretas de la dimensión primera. Se saca un valor de esa distribución que será el usado en la acción. A continuación se pasa esa distribución a otra red neuronal que sacará la distribución de la segunda acción. Se saca el valor de la segunda acción. Seguidamente se pasan las dos distribuciones de la primera y segunda dimensión a una tercera red para sacar la distribución de la tercera dimensión, y así sucesivamente.



Sigue teniendo la desventaja de que los resultados son muy dependientes del tamaño de las discretizaciones.

1.11.3. ¿Qué funciones de coste son buenas para Imitation Learning?

- $r(s, a) = \log p(a = \pi^*(s)|s)$. Se convierte en coste poniéndole el signo negativo (Negative Log Likelihood).
- $c(s, a) = \begin{cases} 0 & \text{if } a = \pi^*(s) \\ 1 & \text{otherwise} \end{cases}$

DAgger funciona con ambas.

A continuación se hace un pequeño análisis. T es un tiempo finito, y se considera la función de recompensa segundaria. Se asume que se hace bien en el set de entrenamiento: $\pi_\theta(a \neq \pi^*(s)|s) \leq \epsilon \forall s \in D_{train,0}$

$$E \left[\sum_t c(s_t, a_t) \right] \leq \epsilon T + (1 - \epsilon)(\epsilon(T - 1) + (1 - \epsilon)(\dots)) \quad (1.1)$$

Esto es así porque si se falla en el primer paso, en el resto también se falla (ϵT), si en el primer paso pero en el segundo no, se tiene $(1 - \epsilon)(\epsilon(T - 1))$ y así sucesivamente.

Lo que significa que la función de coste será $O(\epsilon T^2)$, lo que es muy malo ya que el coste aumenta cuadráticamente con la duración de la trayectoria.

La asunción de que solo podemos coger estados de D_{train} es muy débil. Se sabe que los modelos de aprendizaje supervisados son capaces de generalizar, dado un número grande de datos. Por lo que podremos asumir que el modelo seguirá funcionando mientras reciba estados que provengan de la distribución de datos de los que fue entrenado. Por lo que ahora: $\pi_\theta(a \neq \pi^*(s)|s) \leq \epsilon$ para $s \sim p_{train}(s)$.

Por lo que ahora:

$$E \left[\sum_t c(s_t, a_t) \right] \leq \epsilon T \quad (1.2)$$

Y la función de coste estará acotada por $O(\epsilon T)$ y será razonable usar Imitation Learning.

Tema 2

Introducción al aprendizaje por refuerzo

Clase 3: Introducción a tensorflow

2020-06-12

Está todo en Jupyter.

Clase 4: Introducción a RL

2020-06-12

2.1. Procesos de decisión de Markov (MDP)

Los MDP quedan definidos por: $s, a, r(s, a)$ y $p(s'|s, a)$.

Una cadena de markov es un modelo de grafo sencillo que se define como $M = \{S, T\}$. Donde S es el espacio de estados, T el operador de transición. T es una matriz.

En un proceso de decisión de Markov, básicamente se le añade a la cadena de Markov la posibilidad de tomar acciones, por lo que T pasa a ser un tensor tridimensional. Entre las transiciones entre los estados se devuelve una recompensa ($r : S \times A \rightarrow \mathbb{R}$).

Un proceso de decisión de Markov parcialmente observable se define como $M\{S, A, O, T, \epsilon, r\}$. O es el espacio de observaciones y ϵ es la probabilidad de emisión: $p(o_t|s_t)$.

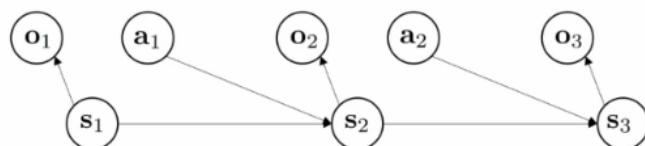


Figura 2.1: Proceso de decisión de Markov parcialmente observable

Como se puede observar, si se conocen los estados las observaciones son independientes.

2.2. Definición del problema de RL

2.2.1. El objetivo de RL

El objetivo consiste en encontrar una política que resuelva el problema. En el caso de redes neuronales, es encontrar los pesos de las neuronas que nos den las acciones correctas dados los estados.

$$p_{\theta}(s_1, a_1, \dots, s_T, a_T) = p(s_1) \prod_{t=1}^T \pi_{\theta}(a_t | s_t) p(s_{t+1} | s_t, a_t)$$

Probabilidad de pasar por una trayectoria. A partir de ahora $p_{\theta}(s_1, a_1, \dots, s_T, a_T) = p_{\theta}(\tau)$

Se pueden definir objetivos como esperanzas de esta distribución. Se puede escribir un optimizador para que nos de la solución de esta manera:

$$\theta^* = \arg \max_{\theta} E_{\tau \sim p_{\theta}(\tau)} \left[\sum_t r(s_t, a_t) \right]$$

Puede ser que algunos estados no permitan ciertas acciones. Esto realmente no importa ya que se puede implementar de forma que una acción inválida no produzca ningún cambio.

Una vez se ha escogido una política, se puede considerar que el problema se transforma en una cadena de markov pero con un espacio de estados distinto.

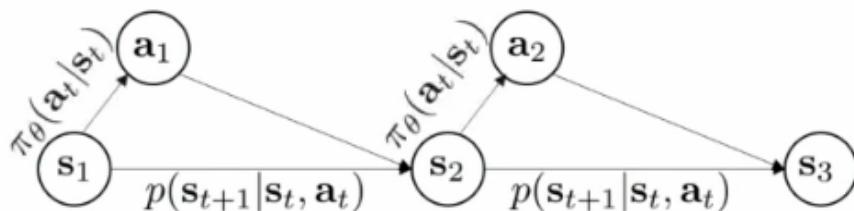


Figura 2.2: Los nuevos estados están formados por los pares (s_t, a_t)

Donde las probabilidades de transición pasan a ser: $p((s_{t+1}, a_{t+1}) | (s_t, a_t)) = p(s_{t+1} | s_t, a_t) \pi_{\theta}(a_{t+1} | s_{t+1})$

Antes de pasar a plantear el problema con una frontera infinita, se define el marginal estado-acción.

$$\theta^* = \arg \max_{\theta} E_{\tau \sim p_{\theta}(\tau)} \left[\sum_t r(s_t, a_t) \right] \quad (2.1)$$

$$= \arg \max_{\theta} \sum_{t+1}^T E_{(s_t, a_t) \sim p_{\theta}(s_t, a_t)} [r(s_t, a_t)] \quad (2.2)$$

Esto se puede hacer porque la esperanza de un sumatorio es equivalente al sumatorio de la esperanza. Como el término dentro del sumatorio sólo depende de (s_t, a_t) , se puede expresar como dependiente de (s_t, a_t) en vez de todas las posibles trayectorias.

¿Qué ocurre si $T = \infty$? Dada la condición de que todos los estados sean accesibles desde todos los otros estados, es decir, que no hayan caminos 'cortados', se demuestra que la distribución de estados visitados cuando $T \rightarrow \infty$ se vuelve estacionaria. Se considerará estacionaria cuando μ sea el vector propio de T correspondiente al valor propio 1. Recordar que μ son los estados de la cadena de markov que están definidos por (s_i, a_i) y T es la matriz de transición de todos esos estados.

Para el caso del horizonte en el infinito, hay que poner $\frac{1}{T}$ antes del sumatorio a la hora de calcular θ^* ya que si no la suma sería infinita. Esto se llama *undiscounted average return*, y no es muy usado ya que se suelen usar *discount factors*.

En RL normalmente nos interesa más las esperanzas de las recompensas que las recompensas en sí. Esto hace que incluso para estados y recompensas discretas la esperanza sea continua en θ por lo que los algoritmos de optimización lo tratarán correctamente.

2.3. Anatomía de un algoritmo de RL

En una forma u otra tienen tres partes:

- Generar samples: se consiguen ejecutando la política
- Paso de evaluación: hacer una cosa que no cambie la política pero permita evaluarla.
- Mejorar la política.

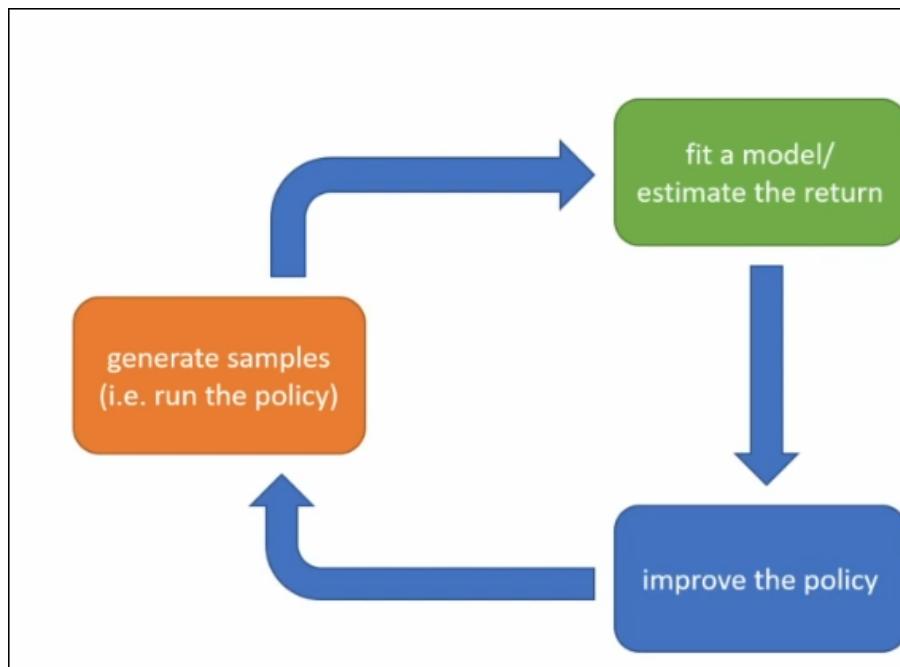


Figura 2.3: Arquitectura simple de un algoritmo RL

2.3.1. ¿Cómo se gestionan todas las esperanzas?

$$E_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^T r(s_t, a_t) \right] \quad (2.3)$$

Se puede expresar de forma conveniente y recursiva:

$$E_{s_1 \sim p(s_1)} [E_{a_1 \sim \pi(a_1|s_1)} [r(s_1, a_1) + E_{s_2 \sim p(s_2|s_1, a_1)} [E_{a_2 \sim \pi(a_2|s_2)} [r(s_2, a_2) + \dots | s_2] | s_1, a_1] | s_1]] \quad (2.4)$$

Gracias a la propiedad Markov, la esperanza de s_3 dependerá de (s_2, a_2) solamente.

Se define la siguiente función:

$$Q(s_1, a_1) = r(s_1, a_1) + E_{s_2 \sim p(s_2|s_1, a_1)} [E_{a_2 \sim \pi(a_2|s_2)} [r(s_2, a_2) + \dots | s_2] | s_1, a_1] \quad (2.5)$$

La esperanza nos queda así:

$$E_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^T r(s_t, a_t) \right] = E_{s_1 \sim p(s_1)} [E_{a_1 \sim \pi(a_1|s_1)} [Q(s_1, a_1) | s_1]] \quad (2.6)$$

De esta manera es muy fácil modificar $\pi_\theta(a_1|s_1)$ si se conoce $Q(s_1, a_1)$. Por ejemplo una forma de maximizar la esperanza sería coger la acción que maximice Q . Esta función es la que se conoce como *Q-function*, y se escribe Q^π si devuelve la esperanza que se obtiene con una política π .

Su contraparte es la función valor, la cual se define como

$$V^\pi(s_t) = \sum_{t'=t}^T E_{\pi_\theta}[r(s_{t'}, a_{t'})|s_t] \quad (2.7)$$

$$V^\pi(s_t) = E_{a_t \sim \pi(a_t|s_t)}[Q^\pi(s_t, a_t)] \quad (2.8)$$

Se discuten las siguientes ideas:

- Si se tiene una política π y se conoce Q^π , se puede mejorar la política haciendo que esta sea $\pi'(a|s) = 1$ si $\arg \max_a Q^\pi(s, a)$. Esta política es tan buena o mejor que π , da igual lo que sea π .
- Se puede usar estas dos funciones para calcular los gradientes que lleven a acciones mejores. Si $Q^\pi(s, a) > V^\pi(s)$, entonces a es mejor que la media, por lo que se puede modificar $\pi(a|s)$ para incrementar la probabilidad de a .

2.4. Resumen de los tipos de algoritmos de RL

- *Policy Gradient*
- Basados en valor. Calculan los valores óptimos de V o Q y no calculan una política.
- Es una mezcla de los dos anteriores. Estiman V o Q y optimizan la política.
- Basados en modelo: planificación, modelo usado para mejorar la política, u otras cosas. Básicamente aprenden $p(s_{t+1}|s_t, a_t)$. Una vez se tiene el modelo, se puede hacer planificación (no calcula una política), que consiste en:
 - Optimización de trayectorias o control óptimo (principalmente para espacios de estado continuos): esencialmente retropropagación para optimizar sobre las acciones.
 - En el caso discreto, se usa por ejemplo *Monte Carlo Tree Search*.

o se puede retropropagar gradientes en la política, o por último usar el modelo para aprender una función de valor mediante programación dinámica.

2.4.1. ¿Por qué hay tantos métodos distintos?

Porque no se conoce cuál es la mejor forma de resolver el problema. Según el problema, puede ser interesante que haya un alto *sampling efficiency*, para otros, se busca una mayor estabilidad o facilidad de uso. También pueden hacerse ciertas asunciones: estocástico o determinista, continuo o discreto, episódico o infinito.

Sampling efficiency hace referencia a cuantas muestras hacen falta para aprender una buena política. Es muy importante por ejemplo en casos en los que no se pueda aprender en simulación.

- *Off-policy*: el modelo es capaz de mejorar la política sin generar nuevas muestras de la política actual.
- *On-policy*: cada vez que se cambie la política, por muy poco que sea, se necesitan crear nuevas muestras a partir de dicha política.

Sample efficiency suele ir invertido con la eficiencia en la práctica. Si se tiene un simulador que sea casi instantáneo, casi siempre un algoritmo genético aprenderá mucho más rápido (en tiempo real). Esto es así porque son mucho más paralelizables y menos costosos computacionalmente.

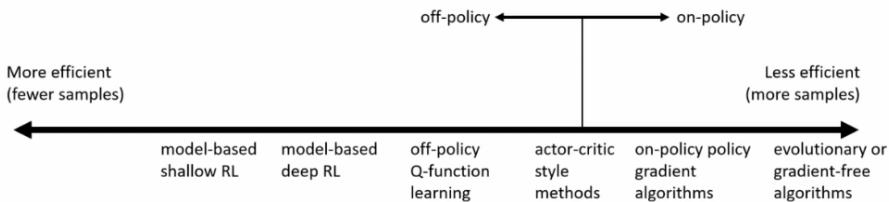


Figura 2.4: Eficiencia de varios algoritmos de RL

Hay otras consideraciones, como que los modelos que están más a la izquierda suelen hacer asunciones más fuertes que los de la derecha. Por ejemplo *Q-learning* requiere de observabilidad completa del estado markoviano.

También se tienen que hacer las siguientes preguntas:

- ¿El modelo converge?
- ¿Si es así, a qué?
- ¿Converge siempre?

Estas preguntas tienen sentido por motivos como los siguientes:

- *Q-learning* no es descenso por gradiente, es una iteración de punto fijo.
- Los algoritmos basados en modelo no optimizan la esperanza de la recompensa.
- *Policy Gradient* si que es descenso por gradiente, pero es el más ineficiente.

Tema 3

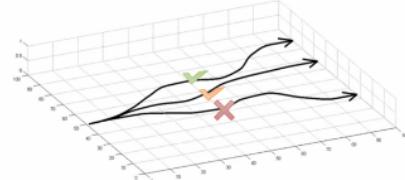
Policy Gradients

Clase 5: Policy Gradients

2020-06-13

3.1. El algoritmo Policy Gradient

$$\theta^* = \arg \max_{\theta} E_{\tau \sim p_{\theta}(\tau)} \left[\underbrace{\sum_t r(\mathbf{s}_t, \mathbf{a}_t)}_{J(\theta)} \right]$$



$$J(\theta) = E_{\tau \sim p_{\theta}(\tau)} \left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right] \approx \frac{1}{N} \sum_i \sum_t r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})$$

↑
sum over samples from π_{θ}

Figura 3.1: Para coger una estimación de la esperanza, se pueden coger N muestras y ponderarlas. Esto nos da un resultado sin bias

Teniendo en cuenta la siguiente identidad:

$$\pi_{\theta} \nabla_{\theta} \log \pi_{\theta}(\tau) = \nabla_{\theta} \pi_{\theta}(\tau) \quad (3.1)$$

Se puede hacer lo siguiente:

$$J(\theta) = E_{\tau \sim \pi_{\theta}(\tau)} [r(\tau)] = \int \pi_{\theta}(\tau) r(\tau) d\tau \quad (3.2)$$

$$r(\tau) = \sum_{t+1}^T r(s_t, a_t) \quad (3.3)$$

$$\nabla_{\theta} J(\theta) = \int \nabla_{\theta} \pi_{\theta}(\tau) r(\tau) d\tau = \int \pi_{\theta}(\tau) \nabla_{\theta} \log \pi_{\theta}(\tau) r(\tau) d\tau = E_{\tau \sim \pi_{\theta}(\tau)} [\nabla_{\theta} \log \pi_{\theta}(\tau) r(\tau)] \quad (3.4)$$

Ahora hay que conocer $\nabla_{\theta} \log \pi_{\theta}(\tau)$.

$$\pi_\theta(s_1, a_1, \dots, s_T, a_T) = p(s_1) \prod_{t=1}^T \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t) \quad (3.5)$$

$$\log \pi_\theta(\tau) = \log p(s_1) + \sum_{t=1}^T \log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t) \quad (3.6)$$

Poniendo lo anterior en la ecuación 3.4, se obtiene:

$$\nabla_\theta J(\theta) = E_{\tau \sim \pi_\theta(\tau)} \left[\left(\sum_{t=1}^T \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \left(\sum_{t=1}^T r(s_t, a_t) \right) \right] \quad (3.7)$$

Ya que el primer y último término de 3.6 no dependen de θ .

Por lo que ahora a la hora de calcular las esperanzas mediante muestras, se guardan las recompensas y los gradientes.

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \right) \left(\sum_{t=1}^T r(s_{i,t}, a_{i,t}) \right) \quad (3.8)$$

Para actualizar el modelo se hace ascensor por gradiente:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \quad (3.9)$$

Algoritmo 2: REINFORCE

mientras no finalice hacer

Muestrear $\{\tau^i\}$ a partir de $\pi_\theta(a_t s_t)$ (ejecutar la política) $\nabla_\theta J(\theta) \approx \sum_i (\sum_t \nabla_\theta \log \pi_\theta(a_{i,t}^i s_{i,t}^i)) (\sum_t r(s_{i,t}^i, a_{i,t}^i))$ $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

fin

3.2. ¿Qué hace Policy Gradient?

Si se mira la expresión de Policy Gradient se puede ver que es similar a Maximum Likelihood salvo por el término de las recompensas. Esto quiere decir que en vez de ahce todas las trayectorias más probables, se hacen más probables aquellas con mayor recompensa, y menos probables aquellas que obtengan menor recompensa.

La propiedad de Markov no se usa en este algiritmo.

El algoritmo REINFROCE descrito antes no funciona si se aplica tal cual. Policy Gradient tiene una alta varianza, lo cual es un problema para la estabilidad. Hay varios métodos para reducirla.

3.3. Reducir la varianza: Causalidad

Para este método, se asume que el pasado influye en el futuro y no al revés. Partiendo de la ecuación 3.8, se agrupan los sumatorios de la siguiente forma equivalente:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \left(\sum_{t=1}^T r(s_{i,t}, a_{i,t}) \right) \quad (3.10)$$

Si se observa detenidamente, se puede ver que en el sumatorio de la derecha se está multiplicando a los gradientes con las recompensas obtenidas en el futuro, lo cual no tiene mucho sentido. Por ello se puede hacer el siguiente cambio:

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_{i,t}|s_{i,t}) \left(\sum_{t'=t}^T r(s_{i,t'}, a_{i,t'}) \right) \quad (3.11)$$

Simplemente por el hecho de que la suma de las recompensas ahora es menor, el valor de la varianza es menor. Este método funciona tan bien en la práctica que normalmente se suele derivar Policy Gradients con el término $\hat{Q}_{i,t} = \sum_{t'=t}^T r(s_{i,t'}, a_{i,t'})$

3.4. Reducir la varianza: Baselines

La explicación de que Policy Gradient modifica θ para preferir los caminos buenos a los malos no es del todo cierta. Por ejemplo en el caso de que se tenga un escenario en el que los caminos buenos tengan como recompensa un millón más uno y los malos un millón menos uno, el ascenso por gradiente hará que todos los caminos sean escogidos por el modelo.

Para arreglar esto, se resta a las recompensas la media de todas las recompensas obtenidas.

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \nabla_{\theta} \log \pi_{\theta}(\tau) [r(\tau) - b] \quad (3.12)$$

$$b = \frac{1}{N} \sum_{i=1}^N r(\tau) \quad (3.13)$$

Resulta que matemáticamente la esperanza de esta expresión es la misma que la de original. Se demuestra calculando la esperanza del nuevo término:

$$E[\nabla_{\theta} \log \pi_{\theta}(\tau) b] = \int \pi_{\theta}(\tau) \nabla_{\theta} \log \pi_{\theta}(\tau) b d\tau = \int \nabla_{\theta} \pi_{\theta}(\tau) b d\tau = b \nabla_{\theta} \int \pi_{\theta}(\tau) d\tau = b \nabla_{\theta} 1 = 0 \quad (3.14)$$

La esperanza no cambia, pero si que lo hace la varianza. La recompensa media no es el *baseline* más óptimo, pero es suficientemente bueno. Para calcular el *baseline* óptimo, se obtiene minimizando la varianza a partir de su definición y derivando con respecto a b para obtener el mínimo.

$$\begin{aligned} \text{Var}[x] &= E[x^2] - E[x]^2 \\ \nabla_{\theta} J(\theta) &= E_{\tau \sim \pi_{\theta}(\tau)} [\nabla_{\theta} \log \pi_{\theta}(\tau) (r(\tau) - b)] \\ \text{Var} &= E_{\tau \sim \pi_{\theta}(\tau)} [(\nabla_{\theta} \log \pi_{\theta}(\tau) (r(\tau) - b))^2] - E_{\tau \sim \pi_{\theta}(\tau)} [\nabla_{\theta} \log \pi_{\theta}(\tau) (r(\tau) - b)]^2 \end{aligned} \quad (3.15)$$

$$\frac{d \text{Var}}{db} = \frac{d}{db} E[g(\tau)^2 (r(\tau) - b)^2] = \frac{d}{db} (E[g(\tau)^2 r(\tau)^2] - 2E[g(\tau)^2 r(\tau)b] + b^2 E[g(\tau)^2]) \quad (3.16)$$

$$= -2E[g(\tau)^2 r(\tau) + 2bE[g(\tau)^2]] = 0 \quad (3.17)$$

$$b = \frac{E[g(\tau)^2 r(\tau)]}{E[g(\tau)^2]} \quad (3.18)$$

Se puede ver que es la esperanza de la recompensa pero ponderada por las magnitudes de los gradientes. b es un vector para cada parámetro del modelo a optimizar, lo que proporciona un *baseline* para cada parámetro.

3.5. Derivación de Policy Gradient con Importance Sampling

$$J(\theta') = E_{\tau \sim \pi_\theta(\tau)} \left[\frac{\hat{\pi}_{\theta'}(\tau)}{\pi_\theta(\tau)} r(\tau) \right] \quad (3.19)$$

$$\nabla_{\theta'} J(\theta') = E_{\tau \sim \pi_\theta(\tau)} \left[\frac{\nabla_{\theta'} \pi_{\theta'}(\tau)}{\pi_\theta(\tau)} r(\tau) \right] = E_{\tau \sim \pi_\theta(\tau)} \left[\frac{\pi_{\theta'}(\tau)}{\pi_\theta(\tau)} \nabla_{\theta'} \log \pi_{\theta'}(\tau) r(\tau) \right] \quad (3.20)$$

En el caso de que se estime localmente: $\theta = \theta'$: $\nabla_{\theta} J(\theta) = E_{\tau \sim \pi_\theta(\tau)} [\nabla_{\theta} \log \pi_\theta(\tau) r(\tau)]$ se obtiene la expresión que nos es familiar. En el caso de que sea *off-policy*, se obtiene lo siguiente:

$$\nabla_{\theta'} J(\theta') = E_{\tau \sim \pi_\theta(\tau)} \left[\frac{\pi_{\theta'}(\tau)}{\pi_\theta(\tau)} \nabla_{\theta'} \log \pi_{\theta'}(\tau) r(\tau) \right] \quad (3.21)$$

$$= E_{\tau \sim \pi_\theta(\tau)} \left[\left(\prod_{t=1}^T \frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \right) \left(\sum_{t=1}^T \nabla_{\theta'} \log \pi_{\theta'}(a_t|s_t) \right) \left(\sum_{t=1}^T r(s_t, a_t) \right) \right] \quad (3.22)$$

En el caso de que T sea muy grande, lo que pasa es que $\frac{\pi_{\theta'}(\tau)}{\pi_\theta(\tau)}$ puede ser exponencialmente pequeño o grande, lo cual puede hacer que los gradientes sean cercanos a nulos o exploten, incrementando la varianza. Se puede intentar aplicar la causalidad, que aunque alivia esto no resuelve el problema:

$$\pi_\theta(\tau) \left[\sum_{t=1}^T \nabla_{\theta'} \log \pi_{\theta'}(a_t|s_t) \left(\prod_{t'=1}^t \frac{\pi_{\theta'}(a_{t'}|s_{t'})}{\pi_\theta(a_{t'}|s_{t'})} \right) \left(\sum_{t'=t}^T r(s_{t'}, a_{t'}) \left(\prod_{t''=t'}^{t'-1} \frac{\pi_{\theta'}(a_{t''}|s_{t''})}{\pi_\theta(a_{t''}|s_{t''})} \right) \right) \right] \quad (3.23)$$

También puede interesarnos actualizar nuestro modelo con muestras obtenidas de una política parecida a la que se tiene actualmente, para lo cual se puede usar la siguiente expresión:

$$\begin{aligned} \nabla_{\theta'} J(\theta') &\approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \frac{\pi_{\theta'}(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})}{\pi_\theta(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})} \nabla_{\theta'} \log \pi_{\theta'}(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \hat{Q}_{i,t} \\ &= \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \cancel{\frac{\pi_{\theta'}(\mathbf{s}_{i,t})}{\pi_\theta(\mathbf{s}_{i,t})}} \frac{\pi_{\theta'}(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})}{\pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})} \nabla_{\theta'} \log \pi_{\theta'}(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \hat{Q}_{i,t} \end{aligned}$$

ignore this part

3.6. Implementar Policy Gradient con diferenciación automática

Es muy ineficiente calcular los gradientes, aplicarle el logaritmo y hacer el sumatorio. En su lugar se tiene que construir un grafo de tal manera que la derivada de ese grafo es el gradiente de la política.

Como las librerías incluyen ya ML como objetivo:

$$\nabla_{\theta} J_{ML}(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_{i,t}|s_{i,t}) \quad J_{ML}(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \log \pi_{\theta}(a_{i,t}|s_{i,t}) \quad (3.24)$$

Lo que se hace es implementar una pseudo-pérdida de forma que su derivada sea el gradiente de la política:

$$\tilde{J}(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \log \pi_\theta(a_{i,t}|s_{i,t}) \hat{Q}_{i,t} \quad (3.25)$$

Un ejemplo en Tensorflow sería el siguiente:

Figura 3.2: Maximum likelihood

```
# Given:
# actions - (N*T) x Da tensor of actions
# states - (N*T) x Ds tensor of states
# Build the graph:
logits = policy.predictions(states) # This should return (N*T) x Da tensor of action logits
negative_likelihoods = tf.nn.softmax_cross_entropy_with_logits(labels=actions, logits=logits)
loss = tf.reduce_mean(negative_likelihoods)
gradients = loss.gradients(loss, variables)
```

```
# Given:
# actions - (N*T) x Da tensor of actions
# states - (N*T) x Ds tensor of states
# q_values - (N*T) x 1 tensor of estimated state-action values
# Build the graph:
logits = policy.predictions(states) # This should return (N*T) x Da tensor of action logits
negative_likelihoods = tf.nn.softmax_cross_entropy_with_logits(labels=actions, logits=logits)
weighted_negative_likelihoods = tf.multiply(negative_likelihoods, q_values)
loss = tf.reduce_mean(weighted_negative_likelihoods)
gradients = loss.gradients(loss, variables)
```

Figura 3.3: Policy Gradient

3.7. Policy Gradients en la práctica

- Los gradientes tienen una alta varianza, al contrario que en aprendizaje supervisado. Los gradientes serán muy ruidosos.
- Normalmente se usan *batches* grandes
- Optimizar los *learning rates* es difícil. ADAM y otros similares pueden aliviar un poco esto.

3.8. Resumen

- Policy Gradients es *on-policy*
- Se puede derivar una variante *off-policy*
 - Usando Importance Sampling
 - Escalado exponencial con respecto a T .
 - Se puede ignorar parte de los pesos.
- Se puede implementar con diferenciación automática.

Tema 4

Métodos Actor Critic

Clase 6: Actor Critic

2020-06-14

4.1. Mejorando Polocy Gradient con un crítico

Se pretende encontrar una mejor estimación de $\hat{Q}_{i,t} = \sum_{t'=t}^T r(s_{i,t'}, a_{i,t'})$. Lo que se quiere es tener una media de varias \hat{Q} , lo que lo acerca más a su esperanza y reduce su varianza.

$$\hat{Q}_{i,t} \approx \sum_{t'=t}^T E_{\pi_\theta}[r(s_{t'}, a_{t'})|s_t, a_t] \quad (4.1)$$

En cuanto al *baseline*, se coge como la media de todas las recompensas obtenidas:

$$b_t = \frac{1}{N} \sum_i Q(s_{i,t}, a_{i,t}) \quad (4.2)$$

También se puede usar un *baseline* que dependa del estado, ya que se puede demostrar que el bias sigue siendo 0.

$$V(s_t) = E_{a_t \sim \pi_\theta(a_t|s_t)}[Q(s_t, a_t)] \quad (4.3)$$

Por lo que el algoritmo Policy Gradient con estas modificaciones queda como:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) (Q(s_{i,t}, a_{i,t}) - V(s_{i,t})) \quad (4.4)$$

La expresión $Q(s_{i,t}, a_{i,t}) - V(s_{i,t})$ significa cuánto mejor es una acción que la media de las acciones que se pueden tomar en ese estado, y se denomina como *advantage*.

Recordatorio.

$$Q^\pi(s_t, a_t) = \sum_{t'=t}^T E_{\pi_\theta}[r(s_{t'}, a_{t'})|s_t, a_t] : \text{recompensa total tomando } a_t \text{ en } s_t \quad (4.5)$$

$$V^\pi(s_t) = E_{a_t \sim \pi_\theta(a_t|s_t)}[Q^\pi(s_t, a_t)] : \text{recompensa total de } s_t \quad (4.6)$$

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t) : \text{cuánto mejor es } a_t \quad (4.7)$$

$$(4.8)$$

Por lo que se puede escribir el algoritmo de la siguiente forma (el cual tendrá una varianza baja):

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_{i,t}|s_{i,t}) A^{\pi}(s_{i,t}, a_{i,t}) \quad (4.9)$$

Cuanto mejor sea la estimación de A^{π} , la varianza será menor.

En el algoritmo con *baseline* que se vio en el tema anterior la estimación del *advantage* no es del todo buena. No tiene bias, pero tiene una varianza mayor que si se calculase una b para cada estado.

Lo que se puede hacer es tener una red neuronal que estime una de estas tres: $Q^{\pi}, V^{\pi}, A^{\pi}$. Cada una tiene sus ventajas e inconvenientes, por ejemplo Q^{π} es más difícil de aprender que V^{π} ya que depende de s_t y a_t , o A^{π} es mucho más dependiente de la política que las otras dos opciones. El algoritmo Actor Critic clásico estima V^{π} .

$$Q^{\pi}(s_t, a_t) = r(s_t, a_t) + \sum_{t'=t+1}^T E_{\pi_{\theta}}[r(s_{t'}, a_{t'})|s_t, a_t] \quad (4.10)$$

$$= r(s_t, a_t) + E_{s_{t+1} \sim p(s_{t+1}|s_t, a_t)}[V^{\pi}(s_{t+1})] \quad (4.11)$$

$$\approx r(s_t, a_t) + V^{\pi}(s_{t+1}) \quad (4.12)$$

El último paso es una aproximación tomando una muestra, esto se puede hacer ya que la función valor tiene en cuenta todas las actualizaciones hechas previamente. Por lo que el advantage queda como:

$$A^{\pi}(s_t, a_t) \approx r(s_t, a_t) + V^{\pi}(s_{t+1}) - V^{\pi}(s_t) \quad (4.13)$$

Por tanto se define una red neuronal que dado s estima un valor $\hat{V}^{\pi}(s)$. Esta red tiene como parámetros ϕ (ya que la red que estima las acciones tiene los parámetros representados con θ).

4.2. El problema de evaluación de la política

Consiste en dada una política, calcular los valores de los estados. Para estimar los valores, se tiene que ejecutar la políticas numerosas veces en el entorno (Monte Carlo).

$$V^{\pi}(s_t) \approx \sum_{t'=t}^T r(s_{t'}, a_{t'}) \quad (4.14)$$

Si se pudiese retroceder en el tiempo (que en el mundo real no se puede), se podrían estimar de la siguiente manera, que daría mejores resultados:

$$V^{\pi}(s_t) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t'=t}^T r(s_{t'}, a_{t'}) \quad (4.15)$$

Pero al no ser siempre posible, la estimación de 4.14 sigue siendo bastante buena. Esto se debe a que al trabajar con un aproximador (red neuronal), no se puede pretender tener valores exactos para cada estado, sino que estados similares acabarán con valores similares.

Para entrenar al crítico, se crean los datos de entrenamiento para hacer aprendizaje supervisado:

$$\{(s_{i,t}, \sum_{t'=t}^T r(s_{i,t'}, a_{i,t'}))\} \quad (4.16)$$

Y se hace la regresión:

$$L(\phi) = \frac{1}{2} \sum_i \|\hat{V}_\phi^\pi(s_i) - y_i\|^2 \quad (4.17)$$

¿Esto se puede mejorar? Sí. Se puede hacer un rollout de la siguiente manera:

$$y_{i,t} = \sum_{t'=t}^T E_{\pi_\theta}[r(s_{t'}, a_{t'})|s_{i,t}] \approx r(s_{i,t}, a_{i,t}) + V^\pi(s_{i,t+1}) \approx r(s_{i,t}, a_{i,t}) + \hat{V}_\phi^\pi(s_{i,t+1}) \quad (4.18)$$

Se puede ver que en el último paso lo que se hace es coger la estimación de \hat{V}^π obtenida por nuestra red neuronal, que es exactamente lo que estamos intentando optimizar. Esto produce un valor ligeramente erróneo, pero reduce la varianza ya que ese valor ha sido calculado tras varios pasos por ese estado. Al hacer esto, los datos de entrenamiento pasan a ser:

$$\{(s_{i,t}, r(s_{i,t}, a_{i,t}) + \hat{V}_\phi^\pi(s_{i,t+1}))\} \quad (4.19)$$

Estos datos tienen bias, pero son una estimación mejor porque reducen la varianza. Cuando se hace esto se le suele llamar *bootstrapping*.

Reordenando las ideas anteriores, se define el algoritmo Actor Critic:

Algoritmo 3: Actor-Critic

mientras seguir con el entrenamiento hacer

- | Muestrear $\{s_i, a_i\}$ de $\pi_\theta(a|s)$
- | Optimizar $\hat{V}_\phi^\pi(s)$ a la suma de recompensas muestreadas
- | Evaluar $\hat{A}^\pi(s_i, a_i) = r(s_i, a_i) + \hat{V}_\phi^\pi(s'_i) - \hat{V}_\phi^\pi(s_i)$
- | $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(a_i|s_i) \hat{A}^\pi(s_i, a_i)$
- | $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

fin

4.3. Factores de descuento

En el caso de que el algoritmo 3 se use en un entorno sin horizonte y en el que las recompensas no dependan del tiempo, por ejemplo recompensa 1 siempre mientras el agente siga vivo, V^π puede llegar a diverger.

Para arreglar esto se pueden usar factores de descuento.

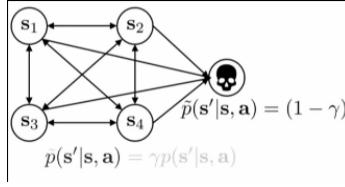
Para esto, se hace la asunción de que las recompensas es mejor obtenerlas cuanto antes. Matemáticamente esto se formula así:

$$y_{i,t} \approx r(s_{i,t}, a_{i,t}) + \gamma \hat{V}_\phi^\pi(s_{i,t+1}) \quad (4.20)$$

Donde γ es el factor de descuento y está entre 0 y 1 (normalmente 0.99 funciona bien).

Al introducir el factor de descuento en un MDP, lo que se está haciendo efectivamente es crear otro MDP en el que hay otro estado nuevo. Este estado nuevo se puede considerar como 'la muerte'. Ahora el agente puede 'morir' con probabilidad $(1 - \gamma)$ desde cualquier estado.

Por ejemplo, la aplicación de esto a Policy Gradient (Monte Carlo) puede quedar de estas formas:



- Opción 1

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_{i,t}|s_{i,t}) \left(\sum_{t'=t}^T \gamma^{t'-t} r(s_{i,t'}, a_{i,t'}) \right) \quad (4.21)$$

- Opción 2: Usarlo antes de aplicar el truco de la causalidad (sección 3.3)

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_{i,t}|s_{i,t}) \right) \left(\sum_{t=1}^T \gamma^{t-1} r(s_{i,t}, a_{i,t}) \right) \quad (4.22)$$

Estos dos no son lo mismo, calculan gradientes diferentes. Para la opción 1, al introducir el crítico se queda de la siguiente forma:

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_{i,t}|s_{i,t}) (r(s_{i,t}, a_{i,t}) + \gamma \hat{V}_{\phi}^{\pi}(s_{i,t+1}) - \hat{V}_{\phi}^{\pi}(s_{i,t})) \quad (4.23)$$

En la opción 2, si se aplica causalidad se obtiene la expresión:

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_{i,t}|s_{i,t}) \left(\sum_{t'=t}^T \gamma^{t'-1} r(s_{i,t'}, a_{i,t'}) \right) \quad (4.24)$$

Lo único que cambia con respecto a la opción 1 es el exponente de γ . Si se reordena, queda más claro cual es la diferencia con respecto a la opción 1:

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \gamma^{t-1} \nabla_{\theta} \log \pi_{\theta}(a_{i,t}|s_{i,t}) \left(\sum_{t'=t}^T \gamma^{t'-t} r(s_{i,t'}, a_{i,t'}) \right) \quad (4.25)$$

Como se puede ver todo queda multiplicado por γ^{t-1} . Lo que hace que los pasos tomados más tarde importen menos que los tomados antes para el gradiente. La opción 2 es la opción correcta si se quiere mantener la misma representación de un MDP con el estado de 'muerte'. Pero en la práctica se suele usar la opción 1, que no resuelve el MDP con el estado de 'muerte'.

La opción 1 en su lugar está optimizando un problema de frontera infinita pero previniendo las sumas infinitas. Estas sumas infinitas hacen que la varianza crezca mucho, por lo que se puede interpretar que γ lo que hace es intercambiar varianza por la introducción de bias (valores bajos de γ producen menos varianza pero más bias).

Demostración: *Bias in natural actor-critic algorithms*. Philip Thomas. ICML 2014.

4.4. El algoritmo Actor Critic

Con descuento, se los algoritmos actor-critic quedan:

Se puede derivar una versión *online*. Con Policy Gradients se necesita una trayectoria completa para estimar el gradiente. Pero con Actor-Critic esto no hace falta, se puede ir estimando cada paso que se toma.

El algoritmo 5 aplicado así no funciona, hay que aplicar algunos ajustes.

Algoritmo 4: Batch Actor-Critic con descuento

mientras seguir con el entrenamiento **hacer**

Muestrear $\{s_i, a_i\}$ de $\pi_\theta(a|s)$

Optimizar $\hat{V}_\phi^\pi(s)$ a la suma de recompensas muestreadas

Evaluar $\hat{A}^\pi(s_i, a_i) = r(s_i, a_i) + \gamma \hat{V}_\phi^\pi(s'_i) - \hat{V}_\phi^\pi(s_i)$

$\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(a_i|s_i) \hat{A}^\pi(s_i, a_i)$

$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

fin

Algoritmo 5: Online Actor-Critic

mientras se siga entrenando **hacer**

Tomar accion $a \sim \pi_\theta(a|s)$, tomar (s, a, s', r)

Actualizar \hat{V}_ϕ^π usando el objetivo $r + \gamma \hat{V}_\phi^\pi(s')$

Evaluar $\hat{A}^\pi(s, a) = r(s, a) + \gamma \hat{V}_\phi^\pi(s') - \hat{V}_\phi^\pi(s)$

$\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(a|s) \hat{A}^\pi(s, a)$

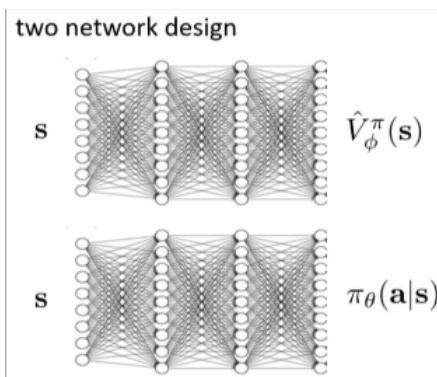
$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

fin

4.4.1. Diseño de la arquitectura

Hay varias formas de crear las redes neuronales para las estimaciones:

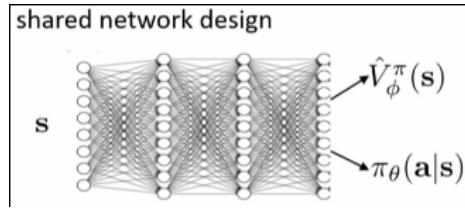
- Dos redes separadas:
 - Es más estable y simple
 - Como desventaja, no se comparten características entre el actor y el crítico, por lo que se tiene que aprender desde 0 en ambos.



- Red compartida
 - Se comparten los pesos, la desventaja es que se tiene que optimizar dos objetivos lo que puede hacer que un gradiente pese más que el otro.

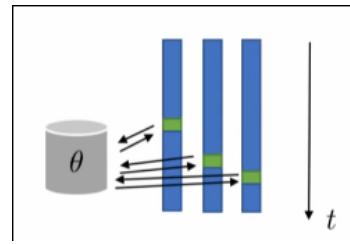
4.4.2. Online Actor-Critic en la práctica

El problema comentado en el algoritmo 5 hace referencia a que en el caso de entrenar de forma continua se tiene un batch de tamaño 1. Esto es mala idea hasta en aprendizaje supervisado, lo que se puede hacer es tener a varios agentes interactuando de manera paralela y recoger un batch a partir de las observaciones de cada uno de ellos.



Una forma de hacer esto es con *synchronized parallel actor-critic*, donde se obtiene un batch, se actualiza θ y así sucesivamente.

También se puede implementar *asynchronous parallel actor-critic*. Donde los parámetros están guardados en un servidor y las distintas instancias se comunican con él cuando lo necesitan.



El caso asíncrono suele ir más rápido, por eso se usa más en la práctica. Lo único paralelo son las simulaciones.

Cabe recordar que esto no se hace para que el algoritmo entrene más rápido, sino para que funcione. Si se quiere entrenar más rápido, habría que tener también a varios trabajadores paralelos calculando gradientes.

4.4.3. Críticos como baselines dependientes del estado

En Actor-Critic, el Critic se usa para estimar el *advantage*. Aunque esto reduce la varianza, incrementa el bias por dos posibles razones:

- Puede ser que se use el critic anterior
- El critic da valores erróneos

Por otro lado, Policy Gradients (que es un método Monte Carlo), no tiene bias, pero padece de una alta varianza.

Se puede calcular \hat{V}_ϕ^π para que sea b y hacer que la varianza se reduzca manteniendo el bias a 0.

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \left(\left(\sum_{t'=t}^T \gamma^{t'-t} r(s_{i,t'}, a_{i,t'}) \right) - \hat{V}_\phi^\pi(s_{i,t}) \right) \quad (4.26)$$

Si se quisiera hacer lo mismo pero con los valores de las acciones en vez del estado, se tiene que sumar la esperanza de Q ya que su media no es 0. La esperanza se puede obtener aproximada (Para más información mirar *Q-prop* de Gu, et. al).

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) (\hat{Q}_{i,t} - Q_\phi^\pi(s_{i,t}, a_{i,t})) + \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta E_{a \sim \pi_\theta(a_t|s_{i,t})} [Q_\phi^\pi(s_{i,t}, a_t)] \quad (4.27)$$

4.5. Eligibility traces y n-step returns

Como se ha visto, se tienen las dos aproximaciones:

$$\hat{A}_C^\pi(s_t, a_t) = r(s_t, a_t) + \gamma \hat{V}_\phi^\pi(s_{t+1}) - \hat{V}_\phi^\pi(s_t)$$

$$\hat{A}_{MC}^\pi(s_t, a_t) = \sum_{t'=t}^{\infty} \gamma^{t'-t} r(s_{t'}, a_{t'}) - \hat{V}_\phi^\pi(s_t)$$

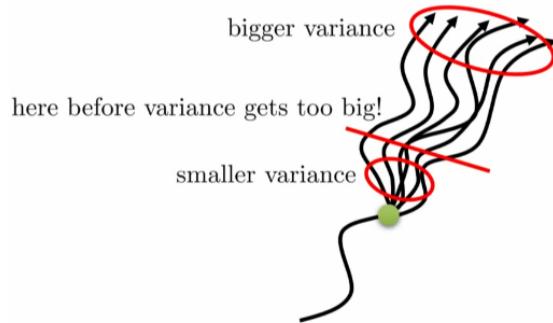
- + lower variance
- higher bias if value is wrong (it always is)
- + no bias
- higher variance (because single-sample estimate)

Se pretende buscar una forma de obtener las ventajas de los dos métodos y controlar la relación bias/varianza.

Cuando se proyecta una trayectoria en el futuro, la varianza de los estados inmediatos es baja, pero la de tiempos lejanos es muy alta. Por ejemplo, si me levanto mañana se que tengo que ir al trabajo en un 99% pero puede ser que me ponga malo con un 1%. Sin embargo, si pienso en mi futuro profesional dentro de 30 años lo más seguro es que me equivoque.

Monte Carlo es útil para obtener valores de tiempos cercanos, pero inútiles a largo plazo. Y los basados en valor son más o menos al revés. Su bias puede ser perjudicial al comienzo pero en el futuro su baja varianza hace que hagan predicciones más o menos buenas.

Lo que se puede hacer es elegir donde cortar:



Lo que se puede hacer es tener una estimación que considere los n pasos más recientes:

$$\hat{A}_n^\pi(s_t, a_t) = \sum_{t'=t}^{t+n} \gamma^{t'-t} r(s_{t'}, a_{t'}) - \hat{V}_\phi^\pi(s_t) \quad (4.28)$$

Esto puede tener sentido en entornos muy estocásticos, donde predecir el futuro no tenga mucho sentido. Pero lo normal es dejar a \hat{V}_ϕ^π para predecir el futuro.

$$\hat{A}_n^\pi(s_t, a_t) = \sum_{t'=t}^{t+n} \gamma^{t'-t} r(s_{t'}, a_{t'}) - \hat{V}_\phi^\pi(s_t) + \gamma^n \hat{V}_\phi^\pi(s_{t+n}) \quad (4.29)$$

Normalmente elegir $n > 1$ da mejores resultados.

4.5.1. Generalized Advantage Estimation

En vez de elegir una n donde cortar en la ecuación 4.29, se puede calcular el *advantage* como una combinación ponderada de infinitos *n-step returns*.

$$\hat{A}_{GAE}^{\pi}(s_t, a_t) = \sum_{n=1}^{\infty} w_n \hat{A}_n^{\pi}(s_t, a_t) \quad (4.30)$$

Los pesos interesa escogerlos de tal forma que los *n-steps* donde la n es menor tengan más preferencia, por lo que se escoge un decrecimiento exponencial:

$$w_n \propto \lambda^{n-1} \quad (4.31)$$

Por lo que queda:

$$\hat{A}_{GAE}^{\pi}(s_t, a_t) = r(s_t, a_t) + \gamma((1 - \lambda)\hat{V}_{\phi}^{\pi}(s_{t+1}) + \lambda(r(s_{t+1}, a_{t+1}) + \gamma((1 - \lambda)\hat{V}_{\phi}^{\pi}(s_{t+2}) \dots \quad (4.32)$$

Que se puede simplificar a:

$$\hat{A}_{GAE}^{\pi}(s_t, a_t) = \sum_{t'=t}^{\infty} (\gamma \lambda)^{t'-t} \delta_{t'} \quad (4.33)$$

Donde d_t es:

$$\delta_{t'} = r(s_{t'}, a_{t'}) + \gamma \hat{V}_{\phi}^{\pi}(s_{t'+1}) - \hat{V}_{\phi}^{\pi}(s_{t'}) \quad (4.34)$$

Se puede observar que γ y λ estan juntas como base de la exponencial, por lo que tienen el mismo significado. Como λ se introdujo para reducir la varianza, se comprueba que se estaba en lo cierto cuando anteriormente en la opción 1 se pensaba que γ reducía la varianza.

Tema 5

Métodos de Funciones Valor

Clase 7: Value Function Methods

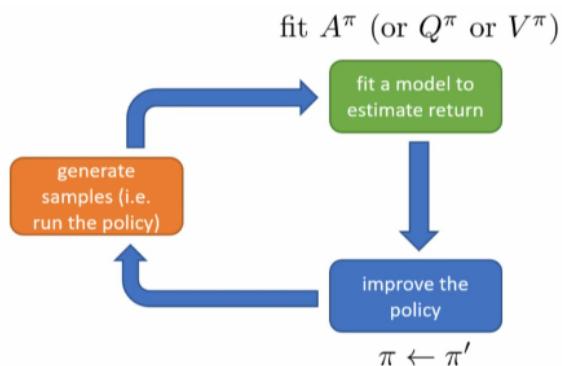
2020-06-15

5.1. Usando solamente un crítico, sin un actor

Los algoritmos basados en Policy Gradient suelen tener una varianza alta. Para contrarrestarlo se introducen *baselines* y un crítico. Se puede pensar que para deshacernos de la varianza completamente se puede eliminar el actor.

Esto se tiene que hacer en la definición del *advantage* $A^\pi(s_t, a_t)$. Se pueden escoger las acciones siguiendo la política $\pi = \arg \max_{a_t} A^\pi(s_t, a_t)$. En este caso, la política será al menos tan buena como cualquier $a_t \sim \pi(a_t, s_t)$ a no ser que π sea la mejor política posible.

$$\pi'(a_t|s_t) = \begin{cases} 1 & \text{si } a_t = \arg \max_{a_t} A^\pi(s_t, a_t) \\ 0 & \text{en cualquier otro caso} \end{cases} \quad (5.1)$$



Para entender estos métodos, se tiene que introducir el concepto de iteración de la política:

Algoritmo 6: Policy Algorithm

```
mientras se entrene hacer
    |   Evaluar  $A^\pi(s, a)$ 
    |    $\pi \leftarrow \pi'$ 
fin
```

Como antes, el *advantage* sigue siendo:

$$A^\pi(s, a) = r(s, a) + \gamma E[V^\pi(s')] - V^\pi(s) \quad (5.2)$$

Para calcular $A^\pi(s, a)$, se tiene que estimar primero V^π . Para estimarlo se puede usar Programación Dinámica.

5.1.1. Programación Dinámica

Se asume que se conoce $p(s'|s, a)$, y s y a son ambos discretos y pequeños.

Como ejemplo, se propone un mundo con 16 estados, 4 acciones (moverse en las direcciones). Por lo que se puede guardar V^π en una tabla. T será un tensor de 16x16x4.

0.2	0.3	0.4	0.3
0.3	0.3	0.5	0.3
0.4	0.4	0.6	0.4
0.5	0.5	0.7	0.5

Para estimar los valores de los estados, se utiliza la técnica llamada *bootstrapped update*, que consiste en:

$$V^\pi(s) \leftarrow E_{a \sim \pi(a|s)}[r(s, a) + \gamma E_{s' \sim p(s'|s, a)}[V^\pi(s')]] \quad (5.3)$$

Donde en $V^\pi(s')$ se usa la estimación que se tenga en ese momento.

Como se usa una política determinista (ecuación 5.1), se puede expresar la política como $\pi(s) = a$, por lo que el *bootstrapped update* queda simplificado a:

$$V^\pi(s) \leftarrow r(s, \pi(s)) + \gamma E_{s' \sim p(s'|s, \pi(s))}[V^\pi(s')] \quad (5.4)$$

En MDP totalmente observables (se tiene el estado en vez de una observación) siempre existe una política determinista que es mejor que cualquier otra política. En el caso de los MDP parcialmente observables esto deja de ser cierto.

Simplificación

Partiendo de la ecuación 5.1, y viendo la definición de $A^\pi(s, a)$ en la ecuación 5.2, se concluye que $\arg \max_{a_t} A^\pi(s_t, a_t) = \arg \max_{a_t} Q^\pi(s_t, a_t)$ ya que el único término de $A^\pi(s, a)$ que depende de a es $r(s, a)$. Esto es de interés porque:

$$Q^\pi(s, a) = r(s, a) + \gamma E[V^\pi(s')] \quad (5.5)$$

Que es una expresión más sencilla que la anterior.

Algoritmo 7: Value Iteration simplificado

mientras $V(s)$ no converja **hacer**

$$\begin{aligned} & | \quad Q(s, a) \leftarrow r(s, a) + \gamma E[V(s')] \\ & | \quad V(s) \leftarrow \max_a Q(s, a) \end{aligned}$$

fin

a			
$Q(s, a)$	$Q(s, a)$	$Q(s, a)$	$Q(s, a)$
$Q(s, a)$	$Q(s, a)$	$Q(s, a)$	$Q(s, a)$
$Q(s, a)$	$Q(s, a)$	$Q(s, a)$	$Q(s, a)$
s	$Q(s, a)$	$Q(s, a)$	$Q(s, a)$
$Q(s, a)$	$Q(s, a)$	$Q(s, a)$	$Q(s, a)$
$Q(s, a)$	$Q(s, a)$	$Q(s, a)$	$Q(s, a)$

Figura 5.1: Se guardan todos los valores de $Q(s, a)$ en una tabla. Los valores $V(s)$ se corresponderán a los máximos por cada fila.

5.1.2. Fitted Value Iteration

La programación dinámica no se puede aplicar a problemas reales por culpa de la maldición de la dimensionalidad, por lo que se puede intentar usar una red neuronal para estimar las funciones de valores.

Se tiene una red neuronal $V : S \mapsto \mathbb{R}$, donde ϕ son sus parámetros, y se entrena con el siguiente objetivo:

$$L(\phi) = \frac{1}{2} \|V_\phi(s) - \max_a Q^\pi(s, a)\|^2 \quad (5.6)$$

Algoritmo 8: Fitted value iteration

```

mientras  $V(s)$  no haya收敛ido hacer
    |    $y_i \leftarrow \max_{a_i}(r(s_i, a_i) + \gamma E[V_\phi(s'_i)])$ 
    |    $\phi \leftarrow \arg \min_\phi \frac{1}{2} \sum_i \|V_\phi(s_i) - y_i\|^2$ 
fin

```

Se tiene que reformular el problema para que la red neuronal no reciba todos los posibles estados, ya que estos pueden crecer exponencialmente con el tamaño del problema.

Todavía se asume que se conoce $p(s'|s, a)$.

Para deshacerse de la necesidad de enumerar todos los estados, se pueden muestrear un subconjunto.

Para deshacerse de la necesidad de conocer la dinámica de transición $p(s'|s, a)$ se puede cambiar el primer paso de *policy iteration*. Donde antes se iteraba:

$$V^\pi(s) \leftarrow r(s, \pi(s)) + \gamma E_{s' \sim p(s'|s, \pi(s))}[V^\pi(s')] \quad (5.7)$$

Ahora pasará a iterarse Q directamente:

$$Q^\pi(s, a) \leftarrow r(s, a) + \gamma E_{s' \sim p(s'|s, a)}[Q^\pi(s', \pi(s'))] \quad (5.8)$$

Por lo que ahora se guarda un valor por cada par estado-acción en vez de solo de por cada acción. La esperanza puede estimarse mediante muestras.

En el algoritmo 8 hay una maximización escondida en $E[V_\phi(s'_i)] \approx \max_{a'} Q_\phi(s'_i, a'_i)$. Pero esta maximización no requiere de conocer las probabilidades de transición, ya que solo se tienen que probar las acciones en el aproximador de funciones (red neuronal). En el caso de que las acciones sean continuas, para encontrar la a' que maximice la Q se tiene que plantear un problema de optimización.

Este método funciona incluso para muestras *off-policy*, y solo se necesita una red neuronal. Como desventaja, se pierden las garantías de convergencia.

Algoritmo 9: Fitted Q-Iteration

Entrada: K: Número de veces que se entrenará con un mismo dataset

Entrada: N: Tamaño del dataset

Entrada: S: Número de pasos del gradiente

mientras se siga entrenando **hacer**

 Crear un dataset $\{(s_i, a_i, s'_i, r_i)\}$ usando una política

mientras Repetir K veces **hacer**

 Se crean los targets usando *bootstrapping*: $y_i \leftarrow r(s_i, a_i) + \gamma \max_{a'_i} Q_\phi(s'_i, a'_i)$

 Regresión: $\phi \leftarrow \arg \min_\phi \frac{1}{2} \sum_i \|Q_\phi(s_i, a_i) - y_i\|^2$

fin

fin

La razón por la que está el bucle exterior es que la política inicialmente puede ser bastante mala y no explorar todos los valores.

En el paso final, se está minimizando el error de Bellman:

$$\epsilon = \|Q_\phi(s_i, a_i) - y_i\|^2 = \frac{1}{2} E_{(s,a) \sim \beta} [(Q_\phi(s, a) - [r(s, a) + \gamma \max_{a'} Q_\phi(s', a')])^2] \quad (5.9)$$

Pero en los otros dos pasos esto no es así (incluso en el paso 2, se incrementa el error de Bellman). Cuando el error de Bellman es 0, $Q_\phi(s, a) = r(s, a) + \gamma \max_{a'} Q_\phi(s', a')$.

Malas noticias: no se sabe realmente lo que se está optimizando con este algoritmo.

5.2. Q-Learning

5.2.1. Online Q-Learning

Algoritmo 10: Fitted Q-Iteration

mientras se siga entrenando **hacer**

 Tomar una acción a_i y observar (s_i, a_i, s'_i, r_i) .

$y_i \leftarrow r(s_i, a_i) + \gamma \max_{a'_i} Q_\phi(s'_i, a'_i)$

$\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(s_i, a_i)(Q_\phi(s_i, a_i) - y_i)$

fin

El paso 1 sigue siendo *off-policy* por lo que se puede usar cualquier política. Para el caso de aproximadores de funciones, no es buena idea entrenar con un batch de tamaño 1 ya que eso genera inestabilidad.

5.2.2. Exploración con Q-Learning

Para el paso 1 del algoritmo anterior, se puede usar la política de la ecuación 5.1. Esta política inicialmente puede ser mala ya que al ser determinista puede ser que no se llegue a ciertos estados necesarios para resolver el problema.

Hay varias formas de arreglar esto. Una de ellas es *epsilon-greedy*, donde la política pasa a ser:

$$\pi(a_t | s_t) = \begin{cases} 1 - \epsilon & \text{si } a_t = \arg \max_{a_t} Q_\phi(s_t, a_t) \\ \frac{\epsilon}{|A|-1} & \text{en caso contrario} \end{cases} \quad (5.10)$$

Otra forma de solucionar esto es la llamada exploración de Boltzmann. La transformación a aplicar tiene que mapear cada valor a un número positivo, la exponencial suele funcionar bien. Este método

tiende a alejar el agente de las decisiones de las que está muy seguro. Si un valor de Q es muy bajo, es muy improbable que lo coja. También puede ser mejor aplicarlo cuando el espacio de acciones es grande.

$$\pi(a_t|s_t) \propto \exp(Q_\phi(s_t, a_t)) \quad (5.11)$$

Tema 6

Deep RL con funciones Q

Clase 8: DRL con funciones Q

2020-06-16

6.1. Hacer funcionar Q-Learning con redes neuronales

Al usar aproximadores de funciones se pierden las garantías de convergencia que se tienen en el caso tabular.

En el caso de *online* Q-Learning, a parte de la demostración teórica de la no convergencia, se tienen también los siguientes problemas:

- La política inicial puede hacer que no se exploren algunos estados nunca.
- (Cierto parecido con la inestabilidad de las GAN)
- Con un batch de tamaño 1, la optimización es muy difícil.
- y_i depende de Q_ϕ , por lo que el algoritmo no es exactamente descenso por gradiente. Para que fuese descenso por gradiente tendría que ser:

$$\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(s_i, a_i)(Q_\phi(s_i, a_i) - r(s_i, a_i) + \gamma \max_{a'} Q_\phi(s'_i, a'_i)) \quad (6.1)$$

Se puede ver que en algoritmo original los gradientes no 'fluyen' por la parte última de la ecuación. Para hacerlo, habría que optimizar el máximo (no lineal) y aún así no sería descenso por gradiente y no convergería (el profesor lo ha dicho).

- Las muestras no son i.i.d, ya que muestras sucesivas están relacionadas.

6.1.1. Muestras relacionadas en *online* Q-Learning

En el algoritmo original, lo que pasa es que se hace *overfitting* de cada conjunto de muestras que están relacionadas entre ellas, olvidando las muestras ya vistas anteriormente.

Para resolver el problema:

- Q-Learning paralelo sincronizado (como con Actor Critic). No resuelve el problema del todo. A diferencia de Actor Critic, Q-Learning es off-policy.
- *Replay buffer*. Se tiene un dataset de transiciones creadas por una política y se entrena el modelo con esos datos. Esto consigue que las muestras no estén relacionadas entre ellas. También hace que el tamaño del batch sea mayor que 1. Cuando se llena el buffer se puede usar cualquier política, como por ejemplo una ϵ -greedy de la política actual.

Algoritmo 11: Q-Learning con Replay Buffer

Crear dataset $\{(s_i, a_i, s'_i, r_i)\}$ usando una política, añadirlo a B
mientras no se hayan hecho K iteraciones **hacer**
 | Muestrear un batch de B
 | $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(s_i, a_i)(Q_\phi(s_i, a_i) - [r(s_i, a_i) + \gamma \max_{a'} Q_\phi(s'_i, a'_i)])$
fin

Normalmente se usa $K = 1$, pero valores mas grandes de K pueden ser más eficientes.

Todavía se tiene el problema de que no se está haciendo gradient descent real.

6.1.2. Q-Learning con Target Networks

Para lidiar con el problema de que no se está haciendo una regresión real, se cambia en el paso 3 del algoritmo anterior el Q_ϕ del final por Q'_ϕ . Esto hace que no hayan gradientes que tengan que 'fluir' por esa dirección y soluciona el problema. Evidentemente tiene que cogerse una función Q'_ϕ razonable por lo que se usa la política que se tenía hace muchas iteraciones. Esto convierte el problema efectivamente en un problema de regresión supervisada.

Algoritmo 12: DQN: Q-Learning con Target Networks y Replay Buffer

mientras se siga entrenando **hacer**
 | Guardar los parámetros de la red neuronal: $\phi' \leftarrow \phi$
 | **mientras** no se hayan hecho N iteraciones **hacer**
 | | Crear dataset $\{(s_i, a_i, s'_i, r_i)\}$ usando una política, añadirlo a B
 | | **mientras** no se hayan hecho K iteraciones **hacer**
 | | | Muestrear un batch de B
 | | | $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(s_i, a_i)(Q_\phi(s_i, a_i) - [r(s_i, a_i) + \gamma \max_{a'} Q_\phi(s'_i, a'_i)])$
 | | **fin**
 | **fin**
fin

El algoritmo clásico DQN es una instancia del algoritmo 12 donde $N = 1$ y $K = 1$.

Esta es una solución heurística a un problema intratable por lo que no se consiguen las garantías de convergencia.

El target network se puede inicializar como se quiera pero una buena idea es hacerlo con pesos pequeños para que la salida sea pequeña e influya más la recompensa en las etapas iniciales.

Esta técnica puede parecer que sea muy brusca en los cambios $\phi' \leftarrow \phi$, por lo que se puede utilizar una promediación de Polyak cada vez que se actualice ϕ :

$$\phi' : \phi' \leftarrow \tau\phi' + (1 - \tau)\phi \quad \tau = 0,999 \text{ funciona bien} \quad (6.2)$$

6.2. Una vista generalizada de los algoritmos Q-Learning

Se puede ver DQN como Fitted Q-Learning pero del revés. Ambos son dos formas de expresar el mismo proceso:

- El algoritmo Online Q-Learning explicado en el tema anterior es un caso particular de esta vista general en el cual los datos se eliminan conforme llegan y los 3 procesos corren a la misma velocidad.

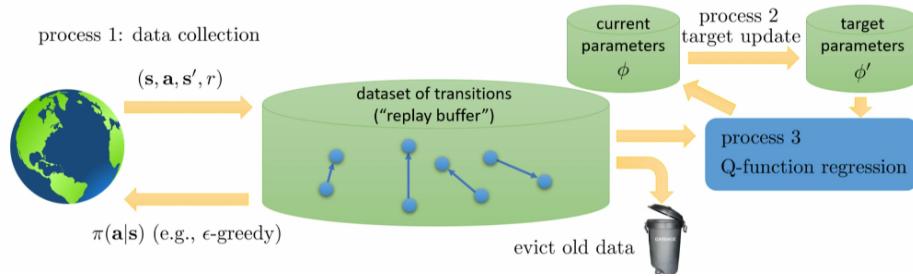


Figura 6.1: Vista general de Q-Learning. El proceso 1 es la obtención del Replay Buffer, que puede ser implementado con una FIFO, ya que se llenará de datos y habrá que ir vaciándola. El proceso 2 es la actualización de la Target Network, que puede ser repentina cada ciertos pasos o continuada con la promediación de Polyak. El proceso 3 es la regresión.

- DQN: los procesos 1 y 3 van a la misma velocidad y el proceso 2 es más lento. El tamaño del buffer es mayor que 1.
- Fitted Q-Iteration: el proceso 3 está en el interior del bucle del proceso 2, el cual a su vez está en el interior del proceso 1.

Se puede acelerar el proceso de aprendizaje si se usa un Prioritized Replay Buffer. En el cual se eligen con más probabilidad las muestras que hacen que se genere el mayor error.

6.3. Trucos prácticos para mejorar Q-Learning

6.3.1. ¿Son los valores Q precisos?

La función Q es una función numérica con significado: la esperanza de la recompensa si empiezas en un estado, tomas una acción y sigues con la política π . La pregunta es si las redes neuronales aprenden los valores de estas predicciones.

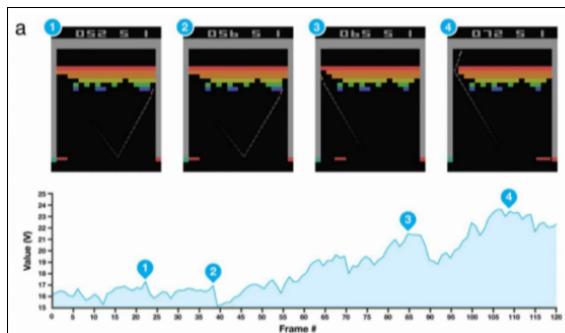


Figura 6.2: En este gráfico se pueden ver los máximos locales de Q cuando el algoritmo piensa que está en un buen estado. Se puede ver que el mayor de estos máximos es cuando rompe la barrera.

Esto parece razonable, pero ¿son numéricamente precisos?

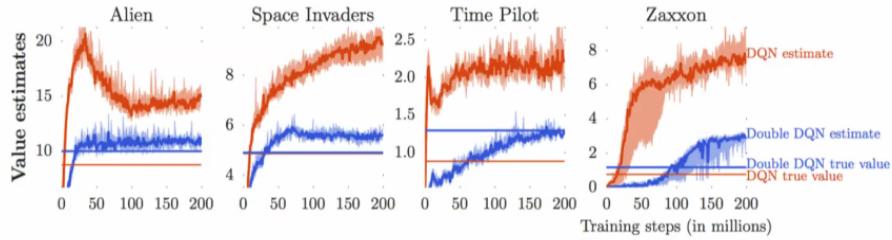


Figura 6.3: (Ingorar por ahora las líneas azules) Se puede ver que las estimaciones de DQN son mayores que la recompensa real obtenida (lineas rojas horizontales).

Se ve que todos los números son muy grandes, DQN es muy 'optimista'. Una intuición de por qué esto pasa es la siguiente: los valores Q estimados vienen con mucha varianza (ruido), por lo que al aplicar el operador máx, se están quedando con esos valores máximos que realmente son ruido.

Más concretamente, el target-value se calcula:

$$y_j = r_j + \gamma \max_{a'_j} Q_{\phi'}(s'_j, a'_j) \quad (6.3)$$

El término máx es el problema. Imaginemos que tenemos dos variables aleatorias X_1 y X_2 (se puede pensar que son valores Q para dos acciones distintas). Se puede demostrar que:

$$E[\max(X_1, X_2)] \geq \max(E[X_1], E[X_2]) \quad (6.4)$$

Lo que significa que si se maximizan estas variables, siempre se coge la mayor + el ruido. Por lo que da un valor siempre mayor que la esperanza de estos valores. $Q_{\phi'}(s', a')$ no es perfecta (se 've' con ruido). Por lo que $\max_{a'} Q_{\phi'}(s', a')$ está sobreestimando el siguiente valor. Otra manera de pensarlo es escribirlo de esta manera:

$$Q_{\phi'}(s', a') = Q_{\phi'}(s', \arg \max_{a'} Q_{\phi'}(s', a')) \quad (6.5)$$

Por lo que si se escoge la acción que maximiza $Q_{\phi'}$ y resulta que esa acción tiene un valor Q erróneamente alto por culpa del ruido, se estará sobreestimando su valor.

6.3.2. Double Q-Learning

El problema viene de que el mismo ruido afecta al argmax y a $Q_{\phi'}$. Por lo que si el ruido aplicado a ambos se decorrelase, el problema se solucionaría. Para conseguir esto se pueden usar dos redes neuronales (cada una con su ruido correspondiente pero no relacionados entre ellos): una para elegir la acción y otra para evaluar el valor:

$$Q_{\phi_A}(s, a) \leftarrow r + \gamma Q_{\phi_B}(s', \arg \max_{a'} Q_{\phi_A}(s', a')) \quad (6.6)$$

$$Q_{\phi_B}(s, a) \leftarrow r + \gamma Q_{\phi_A}(s', \arg \max_{a'} Q_{\phi_B}(s', a')) \quad (6.7)$$

No hace falta aplicar muchos cambios a los algoritmos que ya se han deducido anteriormente en el tema ya que se tiene Q_{ϕ} y $Q_{\phi'}$. Por lo que el único cambio que habría que hacer sería cambiar:

$$y = r + \gamma Q_{\phi'}(s', \arg \max_{a'} Q_{\phi'}(s', a')) \quad (6.8)$$

por:

$$y = r + \gamma Q_{\phi'}(s', \arg \max_{a'} Q_{\phi}(s', a')) \quad (6.9)$$

Esto no es del todo correcto ya que ambas redes no son del todo independientes, pero en la práctica están lo suficientemente decorreladas para que funcione.

En la práctica, el problema de los valores sobreestimados no es tan malo, pero solucionarlo nos lleva a políticas ligeramente mejores.

Se puede ver el efecto de Double Q Learning en la figura 6.3 (color azul).

6.3.3. Multi-step returns

Consiste en desenrollar la función de Bellman. La intuición es que si en el estado actual no se tiene una buena estimación, se puede ir más rápido a un valor cercano si se miran las recompensas más cercanas (parecido al caso de actor-critic en la sección 4.5). Por lo que el target queda como:

$$y_{j,t} = \sum_{t'=t}^{t+N-1} \gamma^{t'-t} r_{j,t'} + \gamma^N \max_{a_{j,t+N}} Q_{\phi'}(s_{j,t+N}, a_{j,t+N}) \quad (6.10)$$

Cuando ya se tenga un valor bastante aproximado, valores altos de N introducirán más varianza por lo que hay que jugar con el valor. Normalmente se usan valores entre 4 y 10.

La desventaja es que las recompensas vienen de una trayectoria, lo que hace que el modelo deje de ser *off-policy*.

Para solucionar este problema se puede hacer lo siguiente:

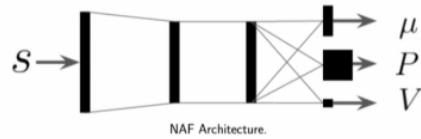
- Ignorarlo (para valores pequeños de N (como por ejemplo 4)) suele funcionar bien.
- Cortar la traza: seleccionar N adaptativamente. Por ejemplo si se está jugando a Atari (dimensiones del espacio de acciones bajo) puede ser que durante varios pasos se elija la misma acción y se puedan meter en un batch.
- Importance Sampling

6.4. Métodos Q-Learning continuos

Para el caso discreto, escoger la acción que maximiza Q es trivial. En el caso continuo, hay varias opciones para aproximar la maximización.

La maximización ocurre en el bucle interno del algoritmo por lo que se quiere evitar que la optimización sea cara computacionalmente.

- Opción 1: optimización
 - correr SGD con respecto a a . No es una buena idea porque es lento.
 - Si el espacio de acciones tiene una dimensionalidad baja, se puede usar optimización estocástica (Monte Carlo: coger aleatoriamente varias a y coger la mayor, por ejemplo). Una solución más precisa es usar *Cross-Entropy Method (CEM)*. Consiste en muestrear unas cuantas acciones, evaluar sus valores y en vez de coger la que tenga el mayor valor, se crea una distribución en las mejores obtenidas y se muestrean acciones de esa distribución, repitiendo el proceso. También se pueden aplicar otros algoritmos como CMA-ES. Estos métodos funcionan para espacios de acciones de dimensionalidad menor que 40.
- Opción 2: usar una clase de funciones que sean fáciles de optimizar. En vez de usar una red neuronal, se puede por ejemplo aproximar los valores Q con una función cuadrática. En el caso de que se tenga un espacio de estados muy complejo, se puede crear una arquitectura que sea cuadrática en las acciones pero no lineal en los estados:



$$Q_\phi(s, a) = -\frac{1}{2}(a - \mu_\phi(s))^T P_\phi(s)(a - \mu_\phi(s)) + V_\phi(s) \quad (6.11)$$

La acción que da el mayor valor es μ . Esta arquitectura se llama NAF (*Normalized Advantage Functions*). Como ventajas tiene que no cambia el algoritmo y es tan eficiente como Q-Learning, pero a costa de perder poder de representación. Se aplica solo en casos en que el espacio de acciones sea muy sencillo.

- Opción 3: entrenar otra red neuronal que prediga la acción que optimice el valor. Un algoritmo de esta categoría sería DDPG (*Deep Deterministic Policy Gradients*). En la publicación del algoritmo los autores lo describen como un método Actor-Critic pero realmente se puede ver como una aproximación de Q-Learning.

Partiendo de la ecuación 6.5 se puede ver que el problema está en calcular el argmax de a . Por lo que se podría entrenar a una red neuronal para que aproxime este valor:

$$\mu_\theta(s) \approx \arg \max_a Q_\phi(s, a) \quad (6.12)$$

Para entrenar a esta red neuronal, simplemente se tiene que resolver $\theta \leftarrow \arg \max_\theta Q_\phi(s, \mu_\theta(s))$. Mediante la regla de la cadena se puede ver que $\frac{dQ_\phi}{d\theta} = \frac{da}{d\theta} \frac{dQ_\phi}{da}$ (un paquete de diferenciación automática hace esto automáticamente). Ahora cuando se calculen los target-values, en vez de usar argmax, se usa μ_θ :

$$y_j = r_j + \gamma Q_{\phi'}(s'_j, \mu_\theta(s'_j)) \approx r_j + \gamma Q_{\phi'}(s'_j, \arg \max_{a'} Q_{\phi'}(s'_j, a'_j)) \quad (6.13)$$

Algoritmo 13: DDPG

mientras se siga entrenando **hacer**

Tomar una acción a_i y observar (s_i, a_i, s'_i, r_i) , añadirlo a B Muestrear un mini-batch de $\{s_j, a_j, s'_j, r_j\}$ de B de manera uniforme. Calcular $y_j = r_j + \gamma Q_{\phi'}(s'_j, \mu_{\theta'}(s'_j))$ usando $Q_{\phi'}$ y $\mu_{\theta'}$ $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(s_j, a_j)(Q_\phi(s_j, a_j) - y_j)$ $\theta \leftarrow \theta + \beta \sum_j \frac{d\mu}{d\theta}(s_j) \frac{dQ_\phi}{da}(s_j, a)$ Actualizar ϕ' y θ' (por ejemplo con Polyak)

fin

Se podría entrenar μ_θ hasta la convergencia, pero Q está cambiando constantemente, por lo que se tiene que ir actualizando.

6.5. Consejos prácticos para Q-Learning

Es buena idea empezar en tareas sencillas primero, ya que el algoritmo no es muy estable. Puede tardar mucho en aprender.

Replay Buffers grandes ayudan a mejorar la estabilidad.

Es recomendable empezar con una alta exploración (ϵ).

Los errores de Bellman pueden hacer que los gradientes sean grandes, por lo que se deben cortar los gradientes o usar *Huber loss* (como MSE en el origen pero lineal lejos del origen).

Usar Double Q Learning, ayuda mucho y es muy sencillo.

N-Steps ayuda pero tienen sus inconvenientes.

Testear los modelos con varias semillas aleatorias, ya que los resultados pueden variar mucho.

Tema 7

Policy Gradient Avanzado

Clase 9: Advanced Policy Gradient

2020-06-17

7.1. ¿Por qué los métodos Policy Gradient funcionan?

Una vista de alto nivel de los algoritmos Policy Gradient es la ejecución en bucle de estos dos pasos:

1. Estimar $\hat{A}^\pi(s_t, a_t)$ para la política π actual.
2. Usar $\hat{A}^\pi(s_t, a_t)$ para obtener una política mejorada π' .

Esto es similar al algoritmo de iteración de política (*Policy Iteration Algorithm*):

1. Evaluar $A^\pi(s, a)$.
2. $\pi \leftarrow \pi'$

Por lo que se puede enmarcar Policy Gradient dentro de Policy Iteration.

7.2. Policy Gradient es un tipo de iteración de política

Siendo:

$$J(\theta) = E_{\tau \sim p_\theta(\tau)} \left[\sum_t \gamma^t r(s_t, a_t) \right] \quad (7.1)$$

Se analiza la mejora del objetivo de la política (diferencia entre dos políticas) (θ' es lo que se busca y θ es la política que ya se tiene).

$$J(\theta') - J(\theta) = J(\theta') - E_{s_0 \sim p(s_0)}[V^{\pi_\theta}(s_0)] \quad (7.2)$$

$$= J(\theta') - E_{\tau \sim p(\tau)}[V^{\pi_\theta}(s_0)] \text{ ya que las trayectorias empiezan en } s_0 \quad (7.3)$$

$$= J(\theta') - E_{\tau \sim p_{\theta'}(\tau)} \left[\sum_{t=0}^{\infty} \gamma^t V^{\pi_\theta}(s_t) - \sum_{t=1}^{\infty} \gamma^t V^{\pi_\theta}(s_t) \right] \quad (7.4)$$

$$= J(\theta') + E_{\tau \sim p_{\theta'}(\tau)} \left[\sum_{t=0}^{\infty} \gamma^t (\gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t)) \right] \quad (7.5)$$

$$= E_{\tau \sim p_{\theta'}(\tau)} \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \right] + E_{\tau \sim p_{\theta'}(\tau)} \left[\sum_{t=0}^{\infty} \gamma^t (\gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t)) \right] \quad (7.6)$$

$$= E_{\tau \sim p_{\theta'}(\tau)} \left[\sum_{t=0}^{\infty} \gamma^t (r(s_t, a_t) + \gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t)) \right] \quad (7.7)$$

$$= E_{\tau \sim p_{\theta'}(\tau)} \left[\sum_{t=0}^{\infty} \gamma^t A^{\pi_\theta}(s_t, a_t) \right] \quad (7.8)$$

Se intenta optimizar esto más, ya que la esperanza está con respecto a $\pi_{\theta'}$. Es muy difícil calcular los gradientes de esta forma desde nuestras muestras, porque la esperanza es con respecto a $\pi_{\theta'}$ y el *advantage* se calcula con π_θ . Se puede hacer lo siguiente:

$$E_{\tau \sim p_{\theta'}(\tau)} \left[\sum_t \gamma^t A^{\pi_\theta}(s_t, a_t) \right] = \sum_t E_{s_t \sim p_{\theta'}(s_t)} [E_{a_t \sim \pi_{\theta'}(a_t | s_t)} [\gamma^t A^{\pi_\theta}(s_t, a_t)]] \quad (7.9)$$

Si se tiene una esperanza de una suma se puede sacar la suma fuera, quedando dentro la suma se la esperanza con respecto al *state-marginal* $p_{\theta'}$ de la acción con respecto a $\pi_{\theta'}$ del *advantage*. A continuación se usa el mismo truco usado en el tema 3 para llevar a cabo Importance Sampling:

$$\begin{aligned} E_{x \sim p(x)}[f(x)] &= \int p(x)f(x)dx \\ &= \int \frac{q(x)}{q(x)}p(x)f(x)dx \\ &= \int q(x)\frac{p(x)}{q(x)}f(x)dx \\ &= E_{x \sim q(x)} \left[\frac{p(x)}{q(x)}f(x) \right] \end{aligned}$$

Por lo que la ecuación anterior queda como:

$$E_{\tau \sim p_{\theta'}(\tau)} \left[\sum_t \gamma^t A^{\pi_\theta}(s_t, a_t) \right] = \sum_t E_{s_t \sim p_{\theta'}(s_t)} \left[E_{a_t \sim \pi_\theta(a_t | s_t)} \left[\frac{\pi_{\theta'}(a_t | s_t)}{\pi_\theta(a_t | s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t) \right] \right] \quad (7.10)$$

Todavía se tiene un problema que impide optimizar el problema: se tiene θ' en la esperanza exterior. Lo que nos impide usar π_θ para crear las muestras. ¿Se podría cambiar $p_{\theta'}$ por p_θ sin que pasase nada (que fuesen lo suficientemente similares)? Si esto fuera cierto, se cumpliría que:

$$J(\theta') - J(\theta) \approx \bar{A}(\theta') \implies \theta' \leftarrow \arg \max_{\theta'} \bar{A}(\theta) \quad (7.11)$$

Donde:

$$\bar{A}(\theta') = \sum_t E_{s_t \sim p_{\theta'}(s_t)} \left[E_{a_t \sim \pi_\theta(a_t|s_t)} \left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t) \right] \right] \quad (7.12)$$

Se pueden calcular los gradientes de $\bar{A}(\theta')$ sin generar ninguna muestra más. Este paso sería el equivalente al paso 2 de *policy iteration*.

¿Esto es verdad? ¿Y si es así, cuándo? Se va a intentar demostrar la afirmación de que $p_\theta(s_t)$ se acerca a $p_{\theta'}(s_t)$ cuando π_θ se está cerca de $\pi_{\theta'}$.

En el caso sencillo, se asume que π_θ es una política determinista $a_t = \pi_\theta(s_t)$. Se dice que una política está cerca de otra si la probabilidad de escoger una acción diferente de la otra es menor que un número pequeño ϵ :

$$\pi_{\theta'}(a_t \neq \pi_\theta(s_t)|s_t) \leq \epsilon \quad (7.13)$$

Esta definición se usó al principio del curso en Imitation Learning. Donde se usó la expresión:

$$p_{\theta'}(s_t) = (1 - \epsilon)^t p_\theta(s_t) + (1 - (1 - \epsilon)^t) p_{\text{mistake}}(s_t) \quad (7.14)$$

Se puede acotar la divergencia de la variación total (*total variation divergence*) entre $\pi_{\theta'}$ y π_θ como:

$$|p_{\theta'}(s_t) - p_\theta(s_t)| = (1 - (1 - \epsilon)^t) |p_{\text{mistake}}(s_t) - p_\theta(s_t)| \quad (7.15)$$

La divergencia de la variación total se define de la siguiente forma, y se puede pensar en ella como la distancia entre dos distribuciones de probabilidad:

$$|p_{\theta'}(s_t) - p_\theta(s_t)| = \sum_{s_t} |p_{\theta'}(s_t) - p_\theta(s_t)| \quad (7.16)$$

La ecuación 7.17 está acotada de la siguiente manera:

$$|p_{\theta'}(s_t) - p_\theta(s_t)| = (1 - (1 - \epsilon)^t) |p_{\text{mistake}}(s_t) - p_\theta(s_t)| \leq 2(1 - (1 - \epsilon)^t) \quad (7.17)$$

Ya que el máximo valor que puede tomar la diferencia absoluta entre dos distribuciones es 2 (porque las distribuciones van de 0 a 1). Para simplificar, se usa la identidad: $(1 - \epsilon)^t \geq 1 - \epsilon t \forall \epsilon \in [0, 1]$. De esta manera se acota por $\leq 2\epsilon t$. No es una gran acotación, ya que crece linealmente, pero sirve.

Esto es para el caso de usar una política determinista. Se quiere acotar para el caso en que se use una política estocástica (lo más común). Para ello, hay que definir de otra forma la noción de cercanía entre dos políticas. Para ello se vuelve a usar la divergencia de variación total.

$$\pi_{\theta'} \text{ es próxima a } \pi_\theta \text{ si } |\pi_{\theta'}(a_t|s_t) - \pi_\theta(a_t|s_t)| \leq \epsilon \text{ para todo } s_t \quad (7.18)$$

Esta definición es la misma que la anterior en el caso de políticas deterministas, por lo que esta es la generalización.

Para poder realizar la derivación a partir de aquí como se hizo para el caso determinista, se usa el siguiente lema:

$$\text{si } |p_X(x) - p_Y(x)| = \epsilon, \text{ existe } p(x, y) \text{ tal que } p(x) = p_X(x) \text{ y } p(y) = p_Y(y) \text{ y } p(x = y) = 1 - \epsilon \quad (7.19)$$

Al crear una probabilidad conjunta a partir de dos distribuciones hay infinitas maneras de hacerlo, pero existe una de ellas en la que se cumple que $x = y$ con probabilidad ϵ . De este lema se concluye que:

- $p_X(x)$ 'coincide' con $p_Y(y)$ con probabilidad ϵ .
- $\pi_{\theta'}(a_t|s_t)$ toma una acción diferente de $\pi_\theta(a_t|s_t)$ con probabilidad ϵ como máximo (si se usan los mismos números aleatorios).

Para un análisis más detallado mirar la publicación *Trust Region Policy Optimization* de Schulman et.al.

Con esto se puede aplicar al caso estocástico el mismo análisis que en el caso determinista, por lo que se obtiene el mismo resultado que en la ecuación 7.17 y queda acotada por $2\epsilon t$ de nuevo. Por lo que los *state-marginals* también serán parecidos.

Sabiendo esto, se intenta demostrar la ecuación 7.11. Para ello se intenta derivar una expresión general de lo anterior para meterla en la demostración.

$$\sum_{s_t} p_{\theta'}(s_t) f(s_t) = \sum_{s_t} (p_{\theta'}(s+t) + p + \theta(s_t) - p_\theta(s_t)) f(s_t) \quad (7.20)$$

$$= \sum_{s_t} p_\theta(s_t) f(s_t) + (p_{\theta'}(s_t) - p_\theta(s_t)) f(s_t) \quad (7.21)$$

$$\geq \sum_{s_t} p_\theta(s_t) f(s_t) + |p_{\theta'}(s_t) - p_\theta(s_t)| f(s_t) \quad (7.22)$$

$$= \sum_{s_t} p_\theta(s_t) f(s_t) + |p_{\theta'}(s_t) - p_\theta(s_t)| \max_{s_t} f(s_t) \quad (7.23)$$

$$\geq E_{p_\theta(s_t)} [f(s_t)] - 2\epsilon t \max_{s_t} f(s_t) \quad (7.24)$$

Si ϵ es un valor muy pequeño, el segundo término puede ser insignificante.

Haciendo derivado esto, se mete en la ecuación 7.17 y se obtiene:

$$\sum_t E_{s_t \sim p_{\theta'}(s_t)} \left[E_{a_t \sim \pi_\theta(a_t|s_t)} \left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t) \right] \right] \geq \quad (7.25)$$

$$\sum_t E_{s_t \sim p_\theta(s_t)} \left[E_{a_t \sim \pi_\theta(a_t|s_t)} \left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t) \right] \right] - \sum_t 2\epsilon t C \quad (7.26)$$

La constante C viene del término $\max_{s_t} f(s_t)$. Este valor se corresponde con el mayor *advantage* que se puede obtener, que es la mayor recompensa que se puede recibir del entorno multiplicada por el número de pasos hasta acabar el episodio. Por lo que la constante está en el orden de $O(Tr_{\max})$. En el caso de un problema de horizonte infinito, este valor es:

$$\sum_t r_{\max} \gamma^t = r_{\max} \sum_t \gamma^t = \frac{r_{\max}}{1 - \gamma} \quad (7.27)$$

Por lo que estará en el orden de $O(\frac{r_{\max}}{1 - \gamma})$.

Se puede ver que la constante C es grande, pero si el valor de ϵ es pequeño, el término sigue siendo gestionable.

Se ha obtenido una cota. Esta cota significa que si se optimiza la esperanza sobre π_θ y se mejora el objetivo por más del término constante, entonces se ha mejorado nuestro objetivo original que es la diferencia de $J(\theta') - J(\theta)$. En resumen, si se mejora por un valor mayor que el término constante, se ha obtenido una mejor política.

En resumen, con esta demostración, se ha comprobado que se puede optimizar $\pi_{\theta'}$ de manera que:

$$\theta' \leftarrow \arg \max_{\theta'} \sum_f E_{s_t \sim p_\theta(s_t)} \left[E_{a_t \sim \pi_\theta(a_t|s_t)} \left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t) \right] \right] \quad (7.28)$$

mientras se cumpla que las dos políticas sean lo suficientemente parecidas:

$$|\pi_{\theta'}(a_t|s_t) - \pi_\theta(a_t|s_t)| \leq \epsilon \quad (7.29)$$

Por lo que se tiene que optimizar con esa restricción. Si se elige un ϵ lo suficientemente pequeño, se garantiza que se mejora $J(\theta') - J(\theta)$.

7.3. Policy Gradient como una optimización con restricciones

La divergencia de variación total es inconveniente para programar algoritmos y para el caso de acciones continuas. Por lo que se busca una restricción más conveniente.

$$|\pi_{\theta'}(a_t|s_t) - \pi_\theta(a_t|s_t)| \leq \sqrt{\frac{1}{2} D_{KL}(\pi_{\theta'}(a_t|s_t) \| \pi_\theta(a_t|s_t))} \quad (7.30)$$

Donde D_{KL} se define como:

$$D_{KL}(p_1(x) \| p_2(x)) = E_{x \sim p_1(x)} \left[\log \frac{p_1(x)}{p_2(x)} \right] \quad (7.31)$$

Se usa esta aproximación ya que D_{KL} tiene ciertas propiedades que hace que sea mucho más fácil de aproximar. Ahora la restricción pasa a ser:

$$D_{KL}(\pi_{\theta'}(a_t|s_t) \| \pi_\theta(a_t|s_t)) \leq \epsilon \quad (7.32)$$

Una forma de plantearlo (no se usa mucho) es con el Lagrangiano:

$$L(\theta', \lambda) = \sum_t E_{s_t \sim p_\theta(s_t)} \left[E_{a_t \sim \pi_\theta(a_t|s_t)} \left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t) \right] \right] - \lambda (D_{KL}(\pi_{\theta'}(a_t|s_t) \| \pi_\theta(a_t|s_t)) - \epsilon) \quad (7.33)$$

Donde se siguen los pasos:

1. Maximizar $L(\theta', \lambda)$ con respecto a θ' .
2. $\lambda \leftarrow \lambda + \alpha(D_{KL}(\pi_{\theta'}(a_t|s_t) \| \pi_\theta(a_t|s_t)) - \epsilon)$

Este algoritmo se llama descenso por gradiente dual, y la intuición detrás de esto es que si la restricción es demasiado violada (D_{KL} mayor que ϵ), se incrementa λ . En caso contrario, disminuir λ .

Otra forma de plantear el problema es hacer una expansión de Taylor en el área que se quiere optimizar ($\pm \epsilon$) y optimizarla, lo que dará una aproximación de la optimización real. Lo más conveniente es usar una aproximación de Taylor de primer orden (linearización). Es conveniente calcular el gradiente en θ en vez de θ' porque en θ el ratio de Importance Sampling vale 1.

Por lo que el problema ahora queda como:

$$\theta' \leftarrow \arg \max_{\theta'} \nabla_\theta \bar{A}(\theta)^T (\theta' - \theta) \quad (7.34)$$

De manera que $D_{KL}(\pi_{\theta'}(a_t|s_t) \| \pi_\theta(a_t|s_t)) \leq \epsilon$.

Derivando el gradiente de la expresión anterior, queda de la siguiente forma:

$$\nabla_{\theta'} \bar{A}(\theta') = \sum_t E_{s_t \sim p_\theta(s_t)} \left[E_{a_t \sim \pi_\theta(a_t|s_t)} \left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \gamma^t \nabla_{\theta'} \log \pi_{\theta'}(a_t|s_t) A^{\pi_\theta}(s_t, a_t) \right] \right] \quad (7.35)$$

Si se evalúa el gradiente en θ , la expresión que queda es más conveniente:

$$\nabla_{\theta} \bar{A}(\theta') = \sum_t E_{s_t \sim p_{\theta}(s_t)} [E_{a_t \sim \pi_{\theta}(a_t|s_t)} [\gamma^t \nabla_{\theta} \log \pi_{\theta'}(a_t|s_t) A^{\pi_{\theta}}(s_t, a_t)]] \quad (7.36)$$

Se puede ver que esto no es exactamente ascenso por gradiente, ya que ascenso por gradiente lo que hace exactamente es:

$$\theta' \leftarrow \arg \max_{\theta'} \nabla_{\theta} J(\theta)^T (\theta' - \theta) \quad (7.37)$$

en el que $\|\theta - \theta'\|^2 \leq \epsilon$ (tamaño del salto).

En el caso lineal (Taylor de primer orden), la actualización de los pesos queda como:

$$\theta' = \theta + \sqrt{\frac{\epsilon}{\|\nabla_{\theta} J(\theta)\|^2}} \nabla_{\theta} J(\theta) \quad (7.38)$$

En la implementación, la raíz cuadrada se corresponde con el *learning rate*. Para nuestro problema, no se quiere esta restricción, se quiere la restricción de D_{KL} . Por lo que Policy Gradients funciona para ϵ pequeños, pero se puede hacer el trabajo de una forma más óptima.

Los algoritmos de optimización con *learning rates* dinámicos como Adam divide el gradiente por su norma. Por lo que esos algoritmos se parecen más al descenso por gradiente real.

Para arreglar esta disimilitud, se va a hacer otra expansión de Taylor, esta vez de segundo orden, en la restricción.

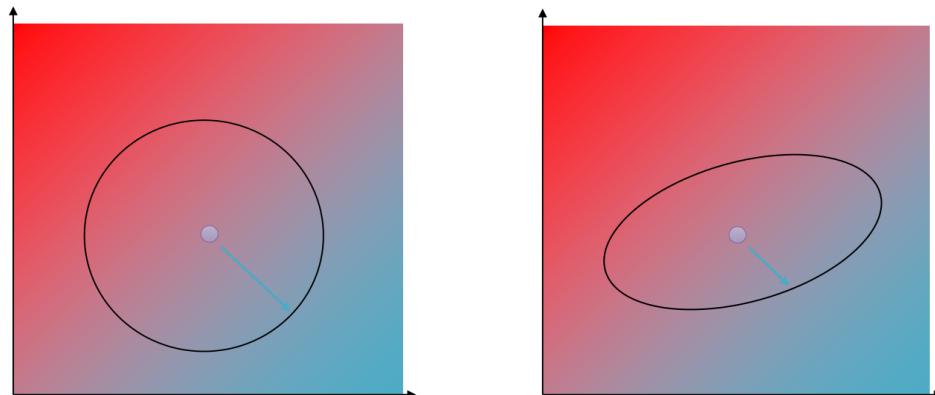
$$D_{KL}(\pi_{\theta'} \| \pi_{\theta}) \approx \frac{1}{2} (\theta' - \theta)^T F (\theta' - \theta) \quad (7.39)$$

Por lo que si la política actual está muy cercana a la política previa (D_{KL} pequeño), entonces se puede aproximar de esta forma. F es la matriz de información de Fisher (Enlace a Wikipedia), y se define como:

$$F = E_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(a|s) \nabla_{\theta} \log \pi_{\theta}(a|s)^T] \quad (7.40)$$

Para calcularla, se pueden usar las mismas muestras que se usaron para estimar el objetivo.

Se reemplaza la restricción D_{KL} por esta restricción cuadrática.



(a) Restricción en el caso de descenso por gradiente. (b) Restricción en el caso de la restricción cuadrática. Se tiene una elipse

Figura 7.1: En estas imágenes, se muestra el salto realizado por ascenso por gradiente en la dirección ascendente. El color azul indica valores altos y el rojo valores bajos. Se puede ver que se está optimizando sobre un plano.

En el caso de la elipse, es difícil calcular exactamente donde va a caer el salto al contorno de la ellipse. Por lo que se puede multiplicar al objetivo por F^{-1} . Por lo que ahora se puede optimizar de forma equivalente al nuevo objetivo con el círculo, lo que es mucho más fácil de calcular.

Por lo que la actualización de los parámetros queda como:

$$\theta' = \theta + \alpha F^{-1} \nabla_{\theta} J(\theta) \quad (7.41)$$

El valor óptimo de α es:

$$\alpha = \sqrt{\frac{2\epsilon}{\nabla_{\theta} J(\theta)^T F \nabla_{\theta} J(\theta)}} \quad (7.42)$$

Al gradiente de la ecuación 7.41 se le llama **gradiente natural**. Se llama así porque este gradiente cambiará la distribución por la misma cantidad sea cual sea la forma en la que esté representada esa distribución (da igual como esté parametrizada).

Como θ se corresponde a los parámetros del aproximador, en el caso de usar una red neuronal es posible que θ sea un vector con millones de parámetros, por lo que para conseguir una matriz *full-rank* habría que tener el mismo número de muestras. Una solución a este problema se planteará más adelante.

7.3.1. ¿Es todo esto necesario?

Para comprobarlo, se experimenta con un entorno de prueba.

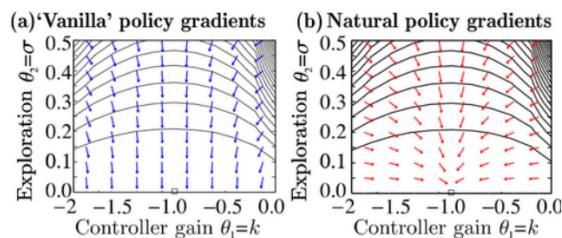


$$r(\mathbf{s}_t, \mathbf{a}_t) = -\mathbf{s}_t^2 - \mathbf{a}_t^2$$

$$\log \pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t) = -\frac{1}{2\sigma^2}(k\mathbf{s}_t - \mathbf{a}_t)^2 + \text{const} \quad \theta = (k, \sigma)$$

Los estados están representados por la recta real negra. El objetivo está en el 0 (representado por la estrella). El agente es el círculo azul y como acciones puede elegir desplazarse a la izquierda o a la derecha con valores continuos. La máxima recompensa se consigue cuando el agente está en el centro y no toma ninguna acción. La política es una distribución normal, con los parámetros σ y k .

Como se tienen sólo dos parámetros, se pueden visualizar los gradientes.



Se puede ver que para Policy Gradients normal, los gradientes no llevan al punto óptimo. En realidad los gradientes de los laterales al punto óptimo están ligeramente inclinados, pero es apenas

perceptible. Esencialmente es el mismo problema encontrado en el descenso por gradiente al intentar optimizar matrices pobres condicionadas (elipses muy 'estiradas'), en la que los parámetros van hacia zig-zag mientras convergen muy lentamente.

Usando los gradientes naturales, los gradientes llevan directamente al punto óptimo.

7.4. Resumen

- Gradiente de la política natural
 - Normalmente es una buena opción para estabilizar PG.
 - Más información en: *Reinforcement Learning of motor skills with policy gradients* por Peter, Schaaal.
 - Para hacer una implementación práctica requiere los productos de vectores de Fisher. No son triviales de conseguir sin calcular la matriz completa (para ello, ver *Trust Region Policy Optimization* por Schulman et. al.)
- TRPO: básicamente es lo mismo pero se escoge α según la expresión 7.42. Se puede pensar que es como una adaptación de Adam especializado en gradientes naturales.
- También se puede usar los gradientes duales explicados anteriormente (Lagrangiano). Mirar: Proximal Policy Optimization.

Tema 8

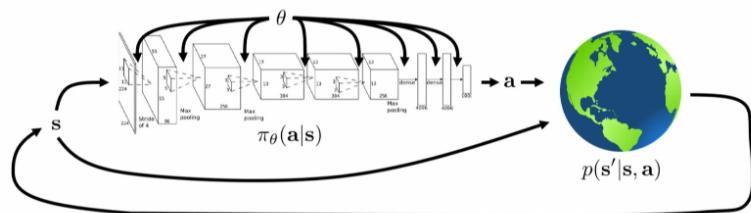
Control Óptimo y Planificación

Clase 10: Optimal Control and Planning

2020-06-18

8.1. Introducción a RL basado en modelo

Hasta ahora en el curso, se usaba objetivo:



$$p_{\theta}(\mathbf{s}_1, \mathbf{a}_1, \dots, \mathbf{s}_T, \mathbf{a}_T) = \underbrace{p(\mathbf{s}_1)}_{\pi_{\theta}(\tau)} \prod_{t=1}^T \pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t) p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)$$

$$\theta^* = \arg \max_{\theta} E_{\tau \sim p_{\theta}(\tau)} \left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right]$$

En el cual no se conocía $p(s_{t+1}|s_t, a_t)$. En RL basado en modelo se intenta aprender este término con el objetivo de tomar mejores acciones.

Normalmente, se conocen las dinámicas del entorno:

1. Videojuegos (Atari, Ajedrez, Go, ...)
2. Sistemas fácilmente modelables (navegación de un coche)
3. Entornos simulados (robots, videojuegos)

También, normalmente se pueden aprender las dinámicas:

1. Identificación del sistema: ajustar los parámetros desconocidos de un modelo. Por ejemplo se usa para encontrar los parámetros desconocidos de un robot.
2. Aprendizaje: ajustar un modelo de propósito general (como una red neuronal) a los datos que se tienen de las transiciones.

8.2. Si conocemos las dinámicas, ¿cómo se toman las decisiones?

El objetivo es minimizar el coste:

$$\min_{a_1, \dots, a_T} \sum_{t=1}^T c(s_t, a_t) \text{ s.t. } s_t = f(s_{t-1}, a_{t-1}) \quad (8.1)$$

En el caso determinista la dinámica del entorno está modelada por una función. Por lo que:

$$a_1, \dots, a_T = \arg \max_{a_1, \dots, a_T} \sum_{t=1}^T r(s_t, a_t) \text{ s.t. } s_{t+1} = f(s_t, a_t) \quad (8.2)$$

En el caso estocástico de bucle abierto, las dinámicas están gobernadas por una distribución de probabilidad.

$$p_\theta(s_1, \dots, s_T | a_1, \dots, a_T) = p(s_1) \prod_{t=1}^T p(s_{t+1} | s_t, a_t) \quad (8.3)$$

$$a_1, \dots, a_T = \arg \max_{a_1, \dots, a_T} E \left[\sum_t r(s_t, a_t) | a_1, \dots, a_T \right] \quad (8.4)$$

El bucle abierto es muy ineficiente, ya que se tienen que coger todas las acciones para terminar el episodio sólo con el estado inicial.

En el bucle cerrado, por otra parte, el agente recibe información del mundo después de haber realizado una acción. En este caso, en vez de una secuencia de acciones lo que se tiene es una política que se va siguiendo, como se ha visto hasta este tema.

$$p(s_1, a_1, \dots, s_T, a_T) = p(s_1) \prod_{t=1}^T \pi(a_t | s_t) p(s_{t+1} | s_t, a_t) \quad (8.5)$$

$$\pi = \arg \max_{\pi} E_{\tau \sim p(\tau)} \left[\sum_t r(s_t, a_t) \right] \quad (8.6)$$

En este tema se explicarán varias formas de realizar la **planificación en bucle abierto** por el momento.

8.3. Métodos estocásticos de optimización

Son optimizadores *black-box* (da igual que sea planificación o control). Se abstrae la planificación/control óptimo.

$$a_1, \dots, a_T = \arg \max_{a_1, \dots, a_T} J(a_1, \dots, a_T) \quad (8.7)$$

Realmente no importan las acciones tomadas, por lo que se escribe de la siguiente manera (siendo J el objetivo):

$$A = \arg \max_A J(A) \quad (8.8)$$

No suele ser una buena idea optimizar esto usando gradientes, por lo que se suelen usar optimizadores libres de derivadas (*Derivative-free optimizers*) como MCTS o CEM. Esto es porque suele ser difícil calcular derivadas de los entornos de simulación, por ejemplo en MuJoCo es mucho más fácil crear episodios que calcular diferenciales.

El método más sencillo para hacer esto es simplemente 'suponer y probar':

1. Escoger A_1, \dots, A_N de una distribución (por ejemplo uniforme).
2. Escoger A_i basándose en $\arg \max_i J(A)$.

Otra manera sencilla y popular de hacerlo mejor es el método de la entropía cruzada (*Cross-entropy method, CEM*). Consiste en hacer lo mismo varias veces, pero actualizando la distribución de la cual se escogen las acciones. Funciona así, iterativamente:

1. Se muestrea A_1, \dots, A_N de $p(A)$.
2. Se evalúa $J(A_1, \dots, J(A_N))$
3. Se escogen los *élites* A_{i_1}, \dots, A_{i_M} con el mayor valor, donde $M < N$.
4. Se ajusta $p(A)$ para los élitess A_{i_1}, \dots, A_{i_M} . Se puede usar cualquier distribución para ajustar, pero normalmente una distribución normal multivariante funciona bien.

(Nótese que A_i es una trayectoria de a_1, \dots, a_K).

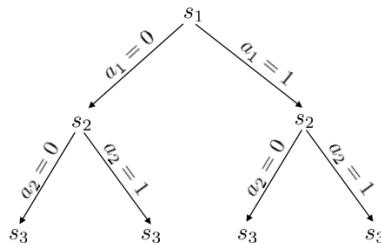
CEM no es un optimizador muy bueno pero es muy sencillo y paralelizable. Como desventajas tiene un límite de dimensionalidad muy fuerte y sólo hace planificación en bucle abierto.

Hay algoritmos similares pero un poco más complejos como CMA-ES, el cual se podría pensar que es CEM con momento.

8.4. Monte Carlo Tree Search

Es otro método que gestiona el caso estocástico **discreto** de una forma más elegante. También funciona en el caso continuo, pero es más complicado. Es muy popular para hacer planificación en videojuegos.

La planificación crece exponencialmente con el número de pasos en el tiempo que se quiere explorar.



Las s no se corresponden a estados sino a pasos en el tiempo. En vez de expandir el árbol hasta el final del episodio, se puede expandir hasta un cierto punto y a partir de ahí tener una estimación a partir de una política para ver si es un buen estado.

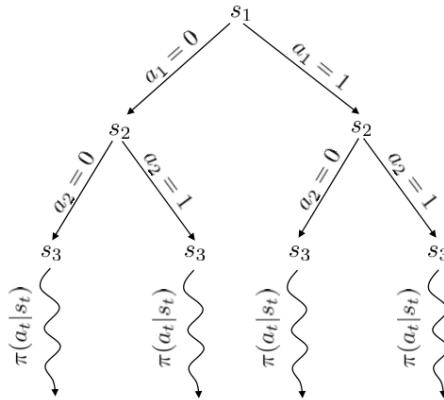


Figura 8.1: La política puede ser una política aleatoria por ejemplo.

La intuición es que a partir de 'desenrollar' los posibles estados inmediatos a los que se puede llegar y después añadir una política 'estúpida' que recoja una aproximación al valor de ese camino, se puede estimar cuál de las acciones es la más adecuada tomar.

Una de las mejoras más inmediatas a MCTS es entrenar una política en vez de usar una aleatoria.

Además, al no poder buscar por todos los caminos es interesante expandir los caminos que inicialmente tengan más recompensa. Esto no garantiza que se encuentre un mejor camino al final. También es posible preferir expandir nodos que no se visiten mucho, ya que quedan por explorar y se podría encontrar en esos un camino mejor.

Un boceto general del algoritmo MTCS puede ser el siguiente (iterativamente):

1. Encontrar una hoja s_l usando un $\text{TreePolicy}(s_1)$. TreePolicy no es una política, sino una estrategia mediante la que se elige qué hoja expandir.
2. Evaluar la hoja usando $\text{DefaultPolicy}(s_l)$. DefaultPolicy sí que es una política y se evalúa 'jugando' hasta el final del juego.
3. Se actualizan todos los valores del árbol entre s_1 y s_l .

Al final, se elige la mejor acción desde s_1 , repitiendo todo el proceso para el siguiente estado.

En cada nodo del árbol se guarda el valor de todas las recompensas, contando las obtenidas por la política 'tonta' (Q) y también se guarda el número de veces que se ha visitado ese nodo (N).

TreePolicy se puede formular de la siguiente forma:

- Si s_t no ha sido totalmente expandido, escoger esa nueva acción a_t .
- En caso contrario, escoger al hijo con la mayor puntuación.

La puntuación es algo que se puede elegir arbitrariamente, y pretende balancear la relación de puntuación Q y el número de veces que se ha visitado un nodo N (para valores menores de N , que sea más probable visitarlo). Una relación que se suele usar es:

$$\text{Score}(s_t) = \frac{Q(s_t)}{N(s_t)} + 2C \sqrt{\frac{2 \ln N(s_{t-1})}{N(s_t)}} \quad (8.9)$$

Donde C es un hiperparámetro.

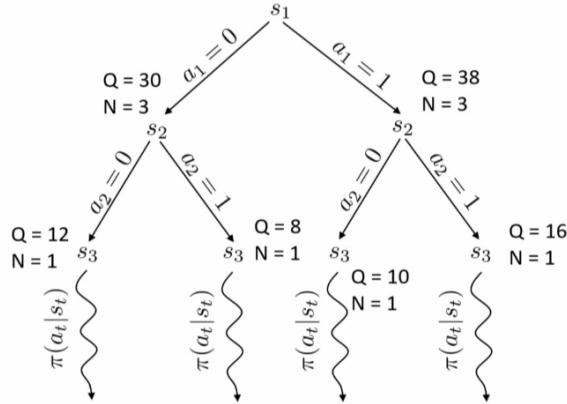


Figura 8.2: Ejemplo de un árbol expandido.

8.5. Optimización de trayectorias

Estos métodos se usan en los casos en los que se puedan obtener derivadas del entorno (por ejemplo modelos dinámicos de un robot). En control óptimo, los estados se representan como x_t y las acciones (control) como u_t . El objetivo en vez de maximizar una recompensa es el de minimizar un coste:

$$\min_{u_1, \dots, u_T} \sum_{t=1}^T c(x_t, u_t) \text{ s.t. } x_t = f(x_{t-1}, u_{t-1}) \quad (8.10)$$

Por ahora, se supone que los problemas son deterministas. La ecuación de arriba representa un problema de optimización con restricciones, pero se pueden quitar las restricciones expresándolo recursivamente como:

$$\min_{u_1, \dots, u_T} c(x_1, u_1) + c(f(x_1, u_1), u_2) + \dots + c(f(f(\dots)), u_T) \quad (8.11)$$

Se tiene una desigualdad como restricción, es muy fácil de sustituir.

Para optimizar esto usando técnicas de descenso por gradiente se necesita $\frac{df}{dx_t}, \frac{df}{du_t}, \frac{dc}{dx_t}, \frac{dc}{du_t}$. En muchos casos de control óptimo, no es una buena idea intentar optimizarlo mediante un optimizador de primer orden y se suelen usar optimizadores de segundo orden.

La expresión 8.11 se llama *Shooting Method*, se llama así porque se está optimizando sobre acciones y los estados son consecuencia de esas acciones. Estos sistemas tienen la propiedad de que cambios pequeños en las acciones del principio llevan a cambios grandes en los estados más avanzados. Numéricamente esto son malas noticias, ya que significa que podemos ser extremadamente sensibles a ciertos parámetros y totalmente insensibles a otros.

Como alternativa existen los llamados *Collocation method*, donde se están optimizando los estados en vez de las acciones, imponiendo una restricción:

$$\min_{u_1, \dots, u_T, x_1, \dots, x_T} \sum_{t=1}^T c(x_t, u_t) \text{ s.t. } x_t = f(x_{t-1}, u_{t-1}) \quad (8.12)$$

Estos métodos se suelen resolver con programación cuadrática.

En este tema se explicarán los *Shooting methods* donde en vez de usar descenso por gradiente se usa LQR, que es un algoritmo de optimización de segundo orden parecido al método de Newton pero sin la necesidad de calcular la matriz hessiana.

Para derivar el algoritmo, se va a hacer con un problema mucho más sencillo, llamado *Linear Quadratic Regulator*. Es un problema de control donde se intenta minimizar el coste como antes (expresión 8.11) pero $f(x_t, u_t)$ tiene una estructura especial (se asume que es lineal).

$$f(x_t, u_t) = F_t \begin{bmatrix} x_t \\ u_t \end{bmatrix} + f_t \quad (8.13)$$

En general puede ser que se tenga una F_t y f_t distintas en cada paso.

También se asume que el coste es cuadrático:

$$c(x_t, u_t) = \frac{1}{2} \begin{bmatrix} x_t \\ u_t \end{bmatrix}^T C_t \begin{bmatrix} x_t \\ u_t \end{bmatrix} + \begin{bmatrix} x_t \\ u_t \end{bmatrix}^T c_t \quad (8.14)$$

También se puede expresar con una constante sumada pero esto no cambia las acciones tomadas por lo que no hace falta tenerlo en cuenta.

Las expresiones 8.13 y 8.14 van respectivamente en 8.11.

Para empezar, se va a resolver para u_T , ya que al estar al final, no afecta a ningún estado futuro porque no hay. El único término que depende de u_T es $c(f(f(\dots)), u_T)$, donde $x_T = f(f(\dots))$ y es desconocido.

El coste para este término es:

$$Q(x_T, u_T) = \text{const} + \frac{1}{2} \begin{bmatrix} x_T \\ u_T \end{bmatrix}^T C_T \begin{bmatrix} x_T \\ u_T \end{bmatrix} + \begin{bmatrix} x_T \\ u_T \end{bmatrix}^T c_T \quad (8.15)$$

El valor de u_T que minimiza esta cantidad en términos de x_T . Como es una función cuadrática, es convexa y su único mínimo es el global. Por lo que:

$$\nabla_{u_T} Q(x_T, u_T) = C_{u_T, x_T} x_T + C_{u_T, u_T} u_T + c_{u_T}^T = 0 \quad (8.16)$$

$$u_T = -C_{u_T, u_T}^{-1} (C_{u_T, x_T} x_T + c_{u_T}) \quad (8.17)$$

Donde:

$$C_T = \begin{bmatrix} C_{x_T, x_T} & C_{x_T, u_T} \\ C_{u_T, x_T} & C_{u_T, u_T} \end{bmatrix} \quad (8.18)$$

$$c_T = \begin{bmatrix} c_{x_T} \\ c_{u_T} \end{bmatrix} \quad (8.19)$$

Para simplificar, se expresa 8.17 como $u_T = K_T x_T + k_T$, donde $K_T = -C_{u_T, u_T}^{-1} C_{u_T, x_T}$ y $k_T = -C_{u_T, u_T}^{-1} c_{u_T}$. Se ha expresado la mejor acción final como una función lineal del estado final.

Ahora se procede de forma recursiva hacia atrás en el tiempo. Como u_T está completamente determinado por x_T , se puede eliminar u_T mediante sustitución.

$$V(x_T) = \text{const} + \frac{1}{2} \begin{bmatrix} x_T \\ K_T x_T + k_T \end{bmatrix}^T C_T \begin{bmatrix} x_T \\ K_T x_T + k_T \end{bmatrix} + \begin{bmatrix} x_T \\ K_T x_T + k_T \end{bmatrix}^T c_T \quad (8.20)$$

Si se expande, se obtiene:

$$V(x_T) = \frac{1}{2} x_T^T C_{x_T, x_T} x_T + \frac{1}{2} x_T^T C_{x_T, u_T} K_T x_T + \frac{1}{2} x_T^T K_T^T C_{u_T, x_T} x_T + \frac{1}{2} x_T^T K_T^T C_{u_T, u_T} K_T x_T + x_T^T K_T^T C_{u_T, u_T} k_T + \frac{1}{2} x_T^T C_{x_T, u_T} k_T + x_T^T c_{x_T} + x_T^T K_T^T c_{u_T} + \text{const} \quad (8.21)$$

Para simplificar la notación, esto se expresa como:

$$V(x_T) = \text{const} + \frac{1}{2} x_T^T V_T x_T + x_T^T v_T \quad (8.22)$$

Donde:

$$\begin{aligned} V_T &= C_{x_T, x_T} + C_{x_T, u_T} K_T + K_T^T C_{u_T, x_T} + K_T^T C_{u_T, u_T} K_T \\ v_T &= c_{x_T} + C_{x_T, u_T} k_T + K_T^T C_{u_T} + K_T^T C_{u_T, u_T} k_T \end{aligned} \quad (8.23)$$

Ahora se resuelve u_{T-1} en términos de x_{T-1} . Pero se tiene que tener en cuenta que x_T también está afectado por u_{T-1} de la siguiente manera:

$$f(x_{T-1}, u_{T-1}) = x_T = F_{T-1} \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix} + f_{T-1} \quad (8.24)$$

$$Q(x_{T-1}, u_{T-1}) = \text{const} + \frac{1}{2} \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T C_{T-1} \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix} + \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T c_{T-1} + V(f(x_{T-1}, u_{T-1})) \quad (8.25)$$

Donde el último término se corresponde con 8.20. Si se expande, queda:

$$V(x_T) = \text{const} + \frac{1}{2} \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T F_{T-1}^T V_T F_{T-1} \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix} + \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T F_{T-1}^T V_T f_{T-1} + \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T F_{T-1}^T v_T \quad (8.26)$$

Que si se simplifica, queda como:

$$Q(x_{T-1}, u_{T-1}) = \text{const} + \frac{1}{2} \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T Q_{T-1} \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix} + \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T q_{T-1} \quad (8.27)$$

Que se puede ver perfectamente que vuelve a tener un término cuadrático y otro término lineal. Para hacer esta simplificación, se tiene en cuenta que:

$$\begin{aligned} Q_{T-1} &= C_{T-1} + F_{T-1}^T V_T F_{T-1} \\ q_{T-1} &= c_{T-1} + F_{T-1}^T V_T f_{T-1} + F_{T-1}^T v_T \end{aligned} \quad (8.28)$$

Como en el primer paso, se vuelve a calcular la derivada de esta expresión y se iguala a 0 para obtener el mínimo. Al hacer esto queda de solución:

$$u_{T-1} = K_{T-1} x_{T-1} + k_{T-1} \quad (8.29)$$

Donde:

$$K_{T-1} = -Q_{u_{T-1}, u_{T-1}}^{-1} Q_{u_{T-1}, x_{T-1}} \quad k_{T-1} = -Q_{u_{T-1}, u_{T-1}}^{-1} q_{u_{T-1}} \quad (8.30)$$

Esto se repite hasta llegar al paso inicial.

Por lo que el algoritmo queda como:

Cuando se finaliza, se llega al estado inicial x_1 el cual es conocido. Ahora lo que se hace es avanzar recursivamente hacia adelante para calcular todas las u_t .

El algoritmo descrito es de bucle abierto, pero se puede modificar para que se vuelva de bucle cerrado.

Este algoritmo es muy eficiente ya que las inversiones de las matrices que hay que hacer son de dimensionalidades bajas (porque es la dimensionalidad del espacio de acciones).

En este algoritmo, Q_t y v_t son literalmente sus funciones valor correspondientes en la formulación de RL (salvo que ahora el v en vez de ser el máximo de sus Q es el mínimo, porque se está minimizando el coste).

En el caso de que se tengan **dinámicas estocásticas** y sean gaussianas (mientras la covarianza sea constante), la solución es exactamente la misma.

Algoritmo 14: LQR

para $t=T$ hasta 1 **hacer**

$$\begin{aligned} Q_t &= C_t + F_t^T V_{t+1} F_t \\ q_t &= c_t + F_t^T V_{t+1} f_t + F_t^T v_{t+1} \\ Q(x_t, u_t) &= \text{const} + \frac{1}{2} \begin{bmatrix} x_t \\ u_t \end{bmatrix}^T Q_t \begin{bmatrix} x_t \\ u_t \end{bmatrix} + \begin{bmatrix} x_t \\ u_t \end{bmatrix}^T q_t \\ u_t &\leftarrow \arg \min_{u_t} Q(x_t, u_t) = K_t x_t + k_t \\ K_t &= -Q_{u_t, u_t}^{-1} Q_{u_t, x_t} \\ k_t &= -Q_{u_t, u_t}^{-1} q_{u_t} \\ V_t &= Q_{x_t, x_t} + Q_{x_t, u_t} K_t + K_t^T Q_{u_t, x_t} + K_t^T Q_{u_t, u_t} K_t \\ v_t &= q_{x_t} + Q_{x_t, u_t} k_t + K_t^T Q_{u_t} + K_t^T Q_{u_t, u_t} k_t \\ V(x_t) &= \text{const} + \frac{1}{2} x_t^T V_t x_t + x_t^T v_t \end{aligned}$$

fin

Algoritmo 15: LQR: recursión hacia adelante

para $t=1$ hasta T **hacer**

$$\begin{aligned} u_t &= K_t x_t + k_t \\ x_{t+1} &= f(x_t, u_t) \end{aligned}$$

fin

Este algoritmo sirve para resolver sistemas lineales. Para **sistemas no lineales** se puede extender el algoritmo a otro llamado LQR Iterativo o Programación Dinámica Diferencial (DDP en inglés).

Su funcionamiento se basa en la aproximación de los sistemas no lineales mediante series de Taylor en el entorno local. Concretamente, una aproximación lineal de la dinámica y una aproximación cuadrática del coste.

$$f(x_t, u_t) \approx f(\hat{x}_t, \hat{u}_t) + \nabla_{x_t, u_t} f(\hat{x}_t, \hat{u}_t) \begin{bmatrix} x_t - \hat{x}_t \\ u_t - \hat{u}_t \end{bmatrix} \quad (8.31)$$

$$c(x_t, u_t) \approx c(\hat{x}_t, \hat{u}_t) + \nabla_{x_t, u_t} c(\hat{x}_t, \hat{u}_t) \begin{bmatrix} x_t - \hat{x}_t \\ u_t - \hat{u}_t \end{bmatrix} + \frac{1}{2} \begin{bmatrix} x_t - \hat{x}_t \\ u_t - \hat{u}_t \end{bmatrix}^T \nabla_{x_t, u_t}^2 c(\hat{x}_t, \hat{u}_t) \begin{bmatrix} x_t - \hat{x}_t \\ u_t - \hat{u}_t \end{bmatrix} \quad (8.32)$$

Se define $\delta x_t = x_t - \hat{x}_t$ y $\delta u = u_t - \hat{u}_t$ (\hat{x} y \hat{u} son las trayectorias que se tenían en la iteración anterior).

$$\begin{aligned} \bar{f}(\delta x_t, \delta u_t) &= F_t \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix} \\ F_t &= \nabla_{x_t, u_t} f(\hat{x}_t, \hat{u}_t) \end{aligned} \quad (8.33)$$

$$\begin{aligned} \bar{c}(\delta x_t, \delta u_t) &= \frac{1}{2} \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix}^T C_t \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix} + \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix}^T c_t \\ C_t &= \nabla_{x_t, u_t}^2 c(\hat{x}_t, \hat{u}_t) \quad c_t = \nabla_{x_t, u_t} c(\hat{x}_t, \hat{u}_t) \end{aligned} \quad (8.34)$$

Con esto, se ejecuta el algoritmo LQR que se tenía antes.

Este algoritmo se parece bastante al método de Newton. Básicamente es el método de Newton aplicado a la optimización de trayectorias. La única diferencia es que el método de Newton usaría una aproximación de segundo orden para aproximar el estado.

Publicación relacionada: *Synthesis and Stabilization of Complex Behaviors through Online Trajectory Optimization*. Proporciona una guía práctica para implementar LQR iterativo.

Algoritmo 16: LQR Iterativo

mientras no haya convergido hacer

$$F_t = \nabla_{x_t, u_t} f(\hat{x}_t, \hat{u}_t)$$

$$c_t = \nabla_{x_t, u_t} c(\hat{x}_t, \hat{u}_t)$$

$$C_t = \nabla_{x_t, u_t}^2 c(\hat{x}_t, \hat{u}_t)$$

Ejecutar el paso hacia atrás de LQR en el estado δx_t y acción δu_t .

Ejecutar el paso hacia adelante con la dinámica no lineal real y $u_t = K_t(x_t - \hat{x}_t) + k_t + \hat{u}_t$

Actualizar \hat{x}_t u \hat{u}_t basándose en los estados y las acciones del paso hacia adelante.

fin

Tema 9

RL basado en modelo

Clase 11: Model-Based Reinforcement Learning

2020-06-20

9.1. Lo básico de RL basado en modelo.

El objetivo es aprender $f(s_t, a_t) = s_{t+1}$ (o $p(s_{t+1}|s_t, a_t)$) ya que conociéndolo se pueden utilizar los algoritmos vistos en el tema anterior.

Una vista general sencilla (que en la práctica no funciona) es la siguiente:

1. Ejecutar la política base $\pi_0(a_t|s_t)$ (por ejemplo aleatoria) para recoger $D = \{(s, a, s')_i\}$
2. Aprender la dinámica $f(s, a)$ de forma que minimice $\sum_i \|f(s_i, a_i) - s'_i\|^2$
3. Planificar con $f(s, a)$ para escoger las acciones.

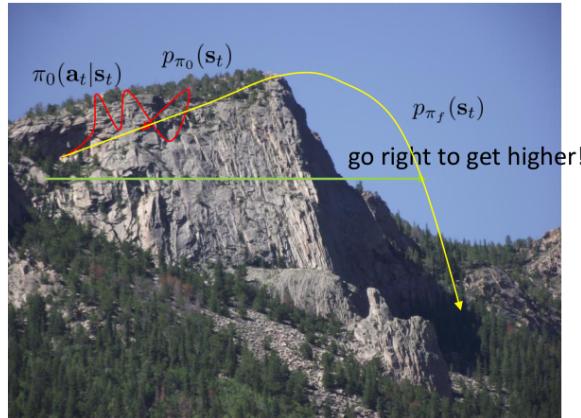
En el paso 2, en el caso de que se tengan estados discretos la función de pérdida podría pasar a ser la entropía cruzada, y en el caso de un modelos estocástico cambiaría completamente.

Esto funciona en casos sencillos, como por ejemplo la identificación de sistemas en robótica. En la que con una política hecha a mano se puede usar para calcular la masa de los enlaces por ejemplo.

En los casos de dinámica más complejas, para modelarlas se tiene que tener una política que explore lo suficiente, ya que de lo contrario se pueden quedar estados sin representar. En estos casos el algoritmo anterior no funciona.

9.1.1. El por qué una aproximación sencilla no funciona

Si se tiene una política $\pi_0(a_t|s_t)$, se generarán muestras del modelo bajo la distribución de esa política: $p_{\pi_0}(s_t)$. Si la política es pobre, es decir, no hace que se explore todo lo posible el espacio de estados, a la hora de dejar actuar a la política final es muy probable que tome acciones pensando que vaya a estados que no son los esperados. Por lo que la distribución de estados visitados bajo esta nueva política ($p_{\pi_f}(s_t)$) no es la misma y el modelo no ha sido entrenado bajo esta distribución.



Este problema se le conoce *distribution mismatch*, el cual se vuelve más exacerbado cuanto más expresivo es el modelo que representa el mundo.

El problema se resolvería si $p_{\pi_0}(s_t) = p_{\pi_f}(s_t)$. Para conseguirlo, se modifica el algoritmo anterior de la siguiente manera (es básicamente DAgger pero para modelos):

1. Ejecutar la política base $\pi_0(a_t|s_t)$ (por ejemplo aleatoria) para recoger $D = \{(s, a, s')_i\}$
2. Aprender la dinámica $f(s, a)$ de forma que minimice $\sum_i \|f(s_i, a_i) - s'_i\|^2$
3. Planificar con $f(s, a)$ para escoger las acciones.
4. Ejecutar esas acciones y añadir las nuevas muestras $\{(s, a, s')_j\}$ a D .

Los pasos 2 a 4 forman un bucle.

Hay una modificación que puede hacerlo funcionar mejor. Consiste en tomar solo una acción y replanificar para el siguiente estado:

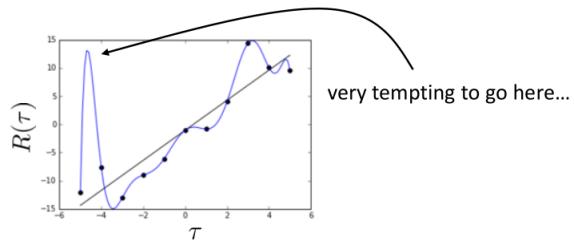
1. Ejecutar la política base $\pi_0(a_t|s_t)$ (por ejemplo aleatoria) para recoger $D = \{(s, a, s')_i\}$
2. Aprender la dinámica $f(s, a)$ de forma que minimice $\sum_i \|f(s_i, a_i) - s'_i\|^2$
3. Planificar con $f(s, a)$ para escoger las acciones.
4. Ejecutar la primera acción y observar el estado resultante s' (MPC).
5. Añadir las nuevas muestras $\{(s, a, s')_j\}$ a D .

Los pasos 3 a 5 están en un bucle. Existe otro bucle externo entre los pasos 2 y 5, que sirve para actualizar el modelo del mundo.

MPC significa *Model Predictive Controller*, que quiere decir que se realiza la planificación en cada paso.

Una planificación que se ejecute en cada paso compensa muy bien las imperfecciones de las dinámicas del mundo. Además hace que no sea necesario usar horizontes muy distantes.

Normalmente, si se usa una red neuronal para codificar la dinámica del entorno, al comienzo los modelos basados en modelo aprenderán mejor que los que no están basados en modelo. Pero con el tiempo se quedan atrapados en un 'máximo local' y son superados por los no basados en modelo. Esto se debe principalmente a que al comienzo la red neuronal se sobreentrena con el *dataset* D , que contiene poca experiencia. Lo que hace que el planificador tienda a escoger falsas altas recompensas.



Este problema se ve agravado conforme las dimensiones van aumentando. Llega un momento en el que es la norma sacar valores erróneos en vez de la excepción.

9.2. Incertidumbre en RL basado en modelo

Para contrarrestar este problema, en vez de usar un modelo determinista para aproximar el entorno, se puede usar un modelo probabilístico, en donde se indica un nivel de desconocimiento en el modelado del mismo.

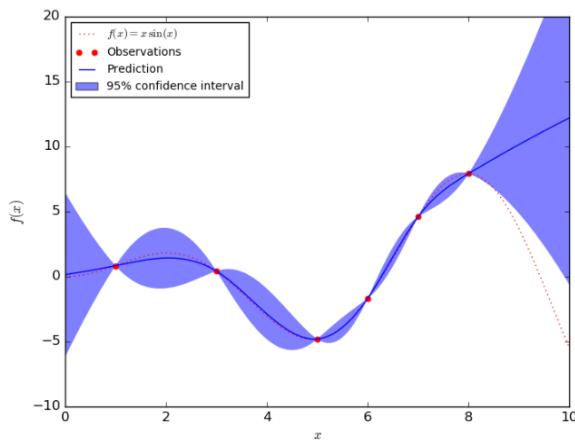


Figura 9.1: Proceso de gaussianas para modelar la dinámica. No es un método escalable pero suficiente para ilustrar.

Al tomar la esperanza, se evita que el agente explote las zonas del aproximador que no son correctas. Además, conforme se vayan recogiendo más muestras, la incertidumbre se irá reduciendo y las predicciones mejorando.

A la hora de usar este modelo hay que tener lo siguiente en cuenta:

- No se tiene un mecanismo para explorar
- El valor esperado no es lo mismo que el valor pesimista. Aunque se podría usar para tareas críticas. No explora mucho.
- El valor esperado no es lo mismo que el valor optimista. Suele explorar más.
- La esperanza es normalmente un buen comienzo.

En el caso de redes neuronales, se puede hacer que sean 'conscientes' de su incertidumbre de diferentes formas:

- Usar la entropía de la salida. Esto no es lo suficientemente bueno porque el modelo puede estar seguro (baja entropía) pero tener una salida errónea. La raíz del problema está en que hay dos tipos de incertidumbre:

- Incertidumbre estadística o aleatoria: básicamente es incertidumbre sobre los datos.
- Incertidumbre del modelo o epistemológica: el modelo predice bien los datos, pero con ese modelo no se obtienen resultados coherentes.

Por lo que en el caso de que se sobreeentrene la red, la entropía parecerá ser baja pero el modelo no será bueno.

- Estimar la incertidumbre del modelo. Cuando se entrena red neuronal, típicamente se suele hacer con *Maximum Likelihood*, donde se estima

$$\arg \max_{\theta} \log p(\theta|D) = \arg \max_{\theta} \log p(D|\theta)$$

¿Se puede aproximar $p(\theta|D)$? Esto nos da la entropía del modelo. Ya que si se tiene que todas las θ son igual de probables dado D , no se tiene ni idea de cuál es el modelo. En caso contrario, si se sabe que una única θ explica D , se conoce completamente el modelo. La entropía de $p(\theta|D)$ nos da la incertidumbre del modelo. Entonces si se quiere predecir, se hace:

$$\int p(s_{t+1}|s_t, a_t, \theta) p(\theta|D) d\theta$$

En la práctica es casi imposible hacerlo así.

Se va a intentar seguir con la segunda aproximación mediante **redes neuronales bayesianas**.

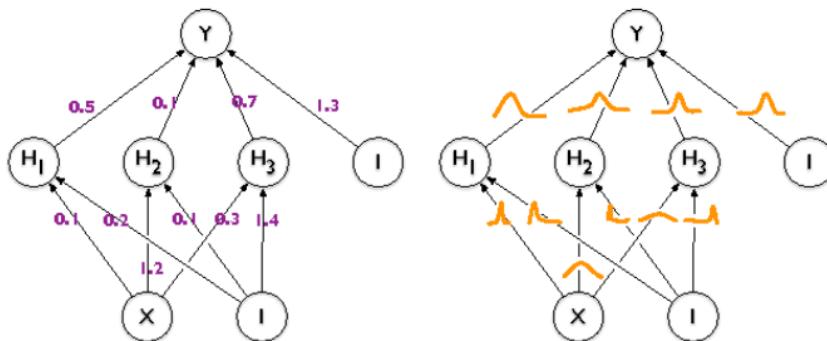


Figura 9.2: Izqda: red neuronal. Dcha: red neuronal bayesiana

En vez de tener un número asociado a cada conexión, se tiene una distribución (sobre los pesos) asociada a cada conexión. Para hacer este problema realizable en la práctica, se realiza una aproximación. Una de las simplificaciones más grandes que se puede hacer es estimar la distribución sobre los parámetros como un producto de marginales independientes:

$$p(\theta|D) = \prod_i p(\theta_i|D) \quad (9.1)$$

$$p(\theta_i|D) = N(\mu_i, \sigma_i) \quad (9.2)$$

Donde μ_i es la esperanza del parámetro y σ_i la incertidumbre sobre ese parámetro.

Esta simplificación se llama *Mean Field Approximation* y suele usarse en física.

Existen varias formas de entrenar redes neuronales bayesianas, siendo una de las más sencillas **bootstrap ensembles**. Formalmente:

$$p(\theta|D) \approx \frac{1}{N} \sum_i \delta(\theta_i) \quad (9.3)$$

Este método consiste en entrenar varios modelos y comprobar si se ponen de acuerdo.

$$\int p(s_{t+1}|s_t, a_t, \theta) p(\theta|D) d\theta \approx \frac{1}{N} \sum_i p(s_{t+1}|s_t, a_t, \theta_i) \quad (9.4)$$

Para entrenar estos modelos, se generan datasets independientes para obtener modelos 'independientes'. Una de las formas de conseguir esto es coger el dataset original D y hacer un remuestreo con reemplazos.

En resumen:

- Este método funciona.
- Es una aproximación bastante simplista, ya que normalmente el número de modelos es pequeño (<10).
- Remuestreo con reemplazo suele ser innecesario. Porque SGD y la inicialización aleatoria normalmente hace que los modelos sean lo suficientemente independientes.

9.2.1. Planificación con incertidumbre

En el tema anterior, se vio que se necesitaba una estimación de la calidad de una serie de acciones.

$$J(a_1, \dots, a_H) = \sum_{t=1}^H r(s_t, a_t), \text{ where } s_{t+1} = f(s_t, a_t) \quad (9.5)$$

Ahora como se tienen N redes neuronales, se saca la media de los modelos:

$$J(a_1, \dots, a_H) = \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^H r(s_{t,i}, a_t), \text{ where } s_{t+1,i} = f_i(s_{t,i}, a_t) \quad (9.6)$$

Lo anterior es para modelos deterministas.

Más generalmente, si se tiene una serie de acciones candidatas a_1, \dots, a_H :

1. Coger una muestra $\theta \sim p(\theta|D)$
2. En cada paso t , coger la muestra $s_{t+1} \sim p(s_{t+1}|s_t, a_t, \theta)$.
3. Calcular $R = \sum_t r(s_t, a_t)$
4. Repetir los pasos de 1 a 3 y acumular la recompensa media.

Como resultado se tiene la esperanza del valor de la recompensa para la serie de acciones candidatas en modelos estocásticos y distribuciones sobre los parámetros.

Existen otras opciones de hacer esto: *moment matching*, estimaciones más complejas con BNN, ...

Publicaciones relacionadas:

- Deisenroth et al. PILCO: A Model-Based and Data-Efficient Approach to Policy Search.
- Nagabandi et al. Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning.
- Chua et al. Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models.
- Feinberg et al. Model-Based Value Expansion for Efficient Model-Free Reinforcement Learning.
- Buckman et al. Sample-Efficient Reinforcement Learning with Stochastic Ensemble Value Expansion.

9.3. RL basado en modelo con observaciones complejas

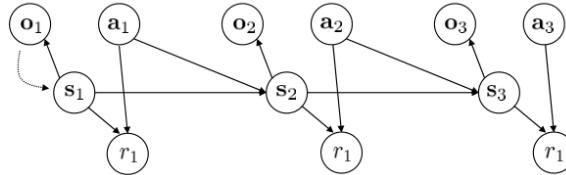
En el caso de que se quieran controlar robots con cámaras, o jugar a Atari, las observaciones se vuelven muy complejas. Si se quiere aprender $f(s_t, a_t) = s_{t+1}$, siendo s imágenes por ejemplo, la tarea se complica por la alta dimensionalidad, la redundancia y la observabilidad parcial (en Atari por ejemplo con una imagen no se puede saber la velocidad de la bola).

9.3.1. State space (latent space) models

Se podría aprender de forma separada $p(o_t|s_t)$ y $p(s_{t+1}|s_t, a_t)$. Esto puede ser bueno porque el primer término tiene una alta dimensionalidad pero no es dinámico, mientras que el segundo puede ser más complejo por la dinámica pero tiene una dimensionalidad mucho más reducida. Para verlo, se puede pensar que en el estado queda codificada la posición de la pala del juego de Atari Breakout mientras que la observación son los píxeles en sí.

Aunque puede ser que esta no sea la aproximación que se tome siempre (ver final del tema).

A la hora de implementar este método, también se necesita aprender la probabilidad de la recompensa dado el estado actual y la acción: $p(r_t|s_t, a_t)$.



¿Cómo se entrena?

Anteriormente, cuando se tenía un modelo totalmente observable, se entrenaba bajo ML:

$$\max_{\phi} \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \log p_{\phi}(\mathbf{s}_{t+1,i} | \mathbf{s}_{t,i}, \mathbf{a}_{t,i}) \quad (9.7)$$

Ahora se tiene que introducir algo más complejo, *expected log-likelihood*:

$$\max_{\phi} \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T E_{(s_t, s_{t+1}) \sim p(s_t, s_{t+1} | o_{1:T}, a_{1:T})} [\log p_{\phi}(\mathbf{s}_{t+1,i} | \mathbf{s}_{t,i}, \mathbf{a}_{t,i}) + \log p_{\phi}(\mathbf{o}_{t,i} | \mathbf{s}_{t,i})] \quad (9.8)$$

La esperanza con respecto a $(s_t, s_{t+1}) \sim p(s_t, s_{t+1} | o_{1:T}, a_{1:T})$ hace que el entrenamiento sea muy complicado.

Esto se suele hacer aprendiendo un posterior aproximado $q_{\phi}(s_t | o_{1:t}, a_{1:t})$ (a veces llamado *encoder*). Existen distintas opciones para aproximar el posterior:

- $q(s_t, s_{t+1} | o_{1:T}, a_{1:T})$: *full smoothing posterior*. Es más preciso pero más complicado de implementar.
- $q_{\phi}(s_t | o_t)$: encoder de un paso. Es el más simple, pero menos preciso.

Estos *encoders* también se tienen que entrenar, ello se discutirá en el siguiente tema.

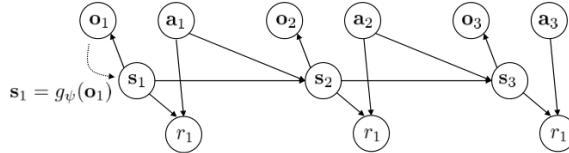
A continuación se muestra como entrenar el encoder de un paso. Por lo que la esperanza se obtiene a partir de $\mathbf{s}_t \sim q_{\psi}(\mathbf{s}_t | \mathbf{o}_t), \mathbf{s}_{t+1} \sim q_{\psi}(\mathbf{s}_{t+1} | \mathbf{o}_{t+1})$. Se considera que en este caso $q(s_t | o_t)$ es determinista.

$$q_{\psi}(\mathbf{s}_t | \mathbf{o}_t) = \delta(\mathbf{s}_t = g_{\psi}(\mathbf{o}_t)) \implies s_t = g_{\psi}(o_t) \quad (9.9)$$

Por lo que queda:

$$\max_{\phi, \psi} \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \log p_\phi(g_\psi(o_{t+1,i}) | g_\psi(o_{t,i}), a_{t,i}) + \log p_\phi(o_{t,i} | g_\psi(o_{t,i})) \quad (9.10)$$

Todo es diferenciable y se puede entrenar con retropropagación.



$$\max_{\phi, \psi} \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \log p_\phi(g_\psi(o_{t+1,i}) | g_\psi(o_{t,i}), a_{t,i}) + \log p_\phi(o_{t,i} | g_\psi(o_{t,i})) \quad (9.11)$$

El primer término se corresponde con el espacio latente de la dinámica, el segundo con la reconstrucción de la imagen y el tercero con el modelo de la recompensa.

Algoritmo 17: RL basado en modelo con espacio latente de modelos

Ejecutar la política base $\pi_0(a_t|o_t)$ (por ejemplo aleatorio) para recoger $D = \{(o, a, o')_i\}$
mientras se siga entrenando **hacer**

Aprender $p_\phi(s_{t+1}|s_t, a_t), p_\phi(r_t|s_t), p(o_t|s_t), g_\psi(o_t)$

mientras no se haya repetido N veces **hacer**

Planificar según el modelo para escoger las acciones

Ejecutar la primera acción de la planificación, observar la o' resultante (MPC)

Añadir (o, a, o') al dataset D .

fin

fin

9.3.2. Aprender directamente en el espacio de observaciones

En este caso, se aprende directamente $p(o_{t+1}|o_t, a_t)$. Esto puede funcionar razonablemente bien.

Por ejemplo se puede entrenar una red recurrente gigante para que haga predicciones de video.

Tema 10

Inferencia variacional y modelos generativos

Clase 12: La clase es la del 2018, la de 2019 no estaba subida

2020-06-30

10.1. Modelos probabilísticos de variables latentes

El modelo probabilístico más sencillo es aquel que estima $p(x)$ (modelo que mejor explica los datos). Otro un poco más complejo es el condicionado $p(y|x)$ (regresión lineal).

Una clase de modelo probabilístico son los modelos de variables latentes.

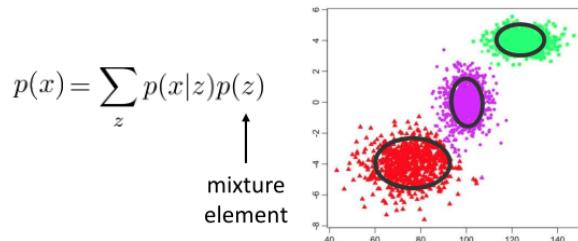


Figura 10.1: En el caso de que se tenga una distribución que por ejemplo siga una mezcla de gaussianas, se puede añadir una variable 'virtual' z (mixture element) que indica a que clase pertenece cada punto.

También se puede dar el caso en probabilidades condicionales:

$$p(y|x) = \sum_z p(y|x, z)p(z) \quad (10.1)$$

Donde $p(z)$ también podría ser $p(z|x)$.

Estos modelos se vieron en Imitation Learning (el ejemplo del árbol, en el que se tiene que elegir uno de los caminos en vez del de en medio). Normalmente una MDN como se vio en 1.11.2 funciona lo suficientemente bien.

Pero puede ser que tener una variable latente que toma valores discretos no es lo suficientemente bueno.

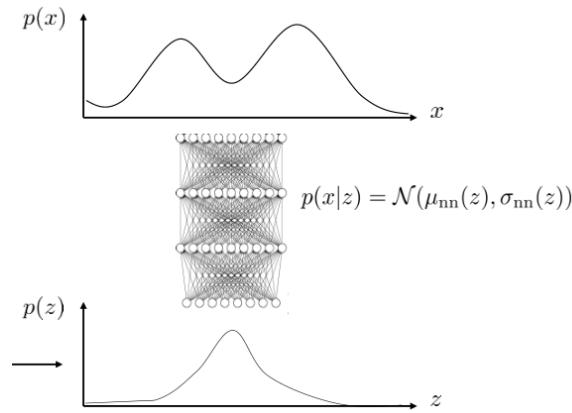
En el caso de que se desee aproximar distribuciones de probabilidad muy complejas, con una mezcla de gaussianas la tarea se puede complicar. Se puede pensar que una distribución compleja viene

de una transformación no lineal de otra más sencilla, por lo que puede ser de interés coger una distribución simple con la que se trabaje fácil como la gaussiana, aplicarle una transformación no lineal con una red neuronal (que son aproximadores de funciones universales) y conseguir así la distribución deseada.

La desventaja es que ahora la expresión anterior se convierte en:

$$p(x) = \int p(x|z)p(z)dz \quad (10.2)$$

Donde ambos términos $p(z)$ y $p(x|z)$ son sencillos (gaussiana y mezcla de gaussianas por ejemplo), pero al ser una integral continua de distribuciones no lineales es muy difícil de calcular.



10.1.1. Modelos con variables latentes en RL

Estos modelos se usan mucho en RL.

- Multi-modal Imitation Learning (el ejemplo del árbol con MDN).
- RL basado en modelo con imágenes. Donde el modelo de variable latente es $p(o_t|x_t)$. Donde se modela $p(x_{t+1}|x_t)$ y $p(x_t)$.
- Al usar RL o técnicas de control combinadas con inferencia variacional para modelar el comportamiento humano.
- En modelos generativos, donde se usa para la exploración.

10.2. Inferencia variacional

Por ahora se trabajará con el modelo más simple: $p_\theta(x)$. Para entrenarlo se usan los datos $D = \{x_1, x_2, x_3, \dots, x_N\}$. Para entrenar se usa *Maximum Likelihood*:

$$\theta \leftarrow \arg \max_{\theta} \frac{1}{N} \sum_i \log p_\theta(x_i) \quad (10.3)$$

Al tener la integral por usar variables latentes continuas, al sustituir se tiene:

$$\theta \leftarrow \arg \max_{\theta} \frac{1}{N} \sum_i \log \left(\int p_\theta(x_i|z)p(z)dz \right) \quad (10.4)$$

Lo cual no es tratable computacionalmente.

Se podría pensar que para entrenar se puede usar *expected log-likelihood* (*Expectation Maximization*). Donde se intenta estimar para cada x a que z pertenece, suponer que es la correcta y actualizar según eso:

$$\theta \leftarrow \arg \max_{\theta} \frac{1}{N} \sum_i E_{z \sim p(z|x_i)} [\log p_{\theta}(x_i, z)] \quad (10.5)$$

Pero se tiene que calcular $p(z|x_i)$. Cuando se tienen z discretas es una tarea sencilla, pero al ser continua y estar estimando $p(x|z)$ mediante una red neuronal, la tarea es muy compleja.

Por lo que se va a hacer una aproximación, en vez de calcular $p(z|x)$, se va a aproximar mediante una gaussiana (o otra distribución simple) $q_i = N(\mu_i, \sigma_i)$. Nótese que q_i es específica para el punto i del dataset. Si se usa el objetivo de la forma correcta se puede usar q_i para delimitar la cantidad que se quiere, que es $\log p(x_i)$. Es una técnica muy sencilla, primero se escribe el *log-likelihood* que se quiere, en este caso con la integral intratable, y se multiplica por 1, permitiendo escribir la esperanza con respecto a $q_i(z)$:

$$\log p(x_i) = \log \int_z p(x_i|z)p(z) \quad (10.6)$$

$$= \log \int_z p(x_i|z)p(z) \frac{q_i(z)}{q_i(z)} \quad (10.7)$$

$$= \log E_{z \sim q_i(z)} \left[\frac{p(x_i|z)p(z)}{q_i(z)} \right] \quad (10.8)$$

Todavía se tiene la integral intratable. A continuación se va a convertir esta igualdad en una cota usando la desigualdad de Jensen: $\log E[y] \geq E[\log y]$ (tener cuidado con los signos).

$$\geq E_{z \sim q_i(z)} \left[\log \frac{p(x_i|z)p(z)}{q_i(z)} \right] \quad (10.9)$$

$$= E_{z \sim q_i(z)} [\log p(x_i|z) + \log p(z)] - E_{z \sim q_i(z)} [\log q_i(z)] \quad (10.10)$$

El último término es simplemente la entropía de q_i : $H(q_i)$. Como este resultado es una cota inferior a $\log p(x_i)$, maximizar esta cota también maximiza $\log p(x_i)$. Al maximizar el primer término de la expresión, lo que se está haciendo es buscar una distribución con poca entropía en el máximo de la distribución que se está逼近ando, y al maximizar el tercer término se está maximizando explícitamente la entropía, por lo que la aproximación se expande por ese entorno.

La divergencia KL se define como:

$$D_{KL}(q||p) = E_{x \sim q(x)} [\log \frac{q(x)}{p(x)}] = E_{x \sim q(x)} [\log q(x)] - E_{x \sim q(x)} [\log p(x)] = -E_{x \sim q(x)} [\log p(x)] - H(q) \quad (10.11)$$

sirve para saber como de diferentes son dos distribuciones. Otra forma de verlo es cuán pequeña es la *expected log probability* de una distribución sobre la otra, menos la entropía.

Se prueba a usar la divergencia KL para逼近ar $p(z|x)$. Por simplicidad, se va a llamar $L_i(p, q_i)$ a $E_{x \sim q(x)} [\log \frac{q(x)}{p(x)}] = E_{x \sim q(x)} [\log q(x)] - E_{x \sim q(x)} [\log p(x)] = -E_{x \sim q(x)} [\log p(x)] - H(q)$. Partiendo de la intuición de que $q_i(z)$ debería逼近ar $p(z|x_i)$, se compara en términos de la divergencia KL:

$$D_{KL}(q_i(x_i) || p(z|x_i)) = E_{z \sim q_i(z)} \left[\log \frac{q_i(z)}{p(z|x_i)} \right] = E_{z \sim q_i(z)} \left[\log \frac{q_i(z)p(x_i)}{p(x_i, z)} \right] \quad (10.12)$$

$$= -E_{z \sim q_i(z)} [\log p(x_i|z) + \log p(z)] + E_{z \sim q_i(z)} [\log q_i(z)] + E_{z \sim q_i(z)} [\log p(x_i)] \quad (10.13)$$

$$= -E_{z \sim q_i(z)} [\log p(x_i|z) + \log p(z)] - H(q_i) + \log p(x_i) \quad (10.14)$$

$$= -L_i(p, q_i) + \log p(x_i) \quad (10.15)$$

Por lo que, moviendo los términos:

$$\log p(x_i) = D_{KL}(q_i(x_i)\|p(z|x_i)) + L_i(p, q_i) \quad (10.16)$$

$$\log p(x_i) \geq L_i(p, q_i) \quad (10.17)$$

Cuanto menor sea la divergencia KL, mejor será la cota inferior, por lo que interesa que tienda a 0. Resulta que minimizar la divergencia KL con respecto a q es equivalente a maximizar la cota inferior de la evidencia con respecto a q . Esto se puede ver porque en las expresiones 10.12 y 10.14 se ve que para minimizar la divergencia hay que reducir sólo los términos que son dependientes de q (todos menos el último).

En resumen, maximizar $L_i(p, q_i)$ con respecto a q_i minimiza la divergencia KL.

Para hacerlo, en vez de tener ML como objetivo se tiene:

$$\theta \leftarrow \arg \max_{\theta} \frac{1}{N} \sum_i L_i(p, q_i) \quad (10.18)$$

Se quiere usar SGD con minibatches ya que es lo que se usa en DL. Se explica para un minibatch de tamaño 1 pero para tamaños más grandes simplemente se calcula la media del gradiente.

Algoritmo 18: Inferencia variacional

```

para cada  $x_i$  (o mini-batch) hacer
    Calcular  $\nabla_{\theta} L_i(p, q_i)$ :
        Muestrear  $z \sim q_i(x_i)$ 
         $\nabla_{\theta} L_i(p, q_i) \approx \nabla_{\theta} L_i(p, q_i)$ 
         $\theta \leftarrow \theta + \alpha \nabla_{\theta} L_i(p, q_i)$ 
    Actualizar  $q_i$  para maximizar  $L_i(p, q_i)$ 
fin

```

¿Cómo se actualiza q_i en el último paso? Supongamos que $q_i(z) = N(\mu_i, \sigma_i)$. Entonces se podría calcular $\nabla_{\mu} L_i(p, q_i)$ y $\nabla_{\sigma} L_i(p, q_i)$ y escalar el gradiente. Esto funciona, pero hay un problema con este procedimiento: si se usa DL, se tiene un gran número de datapoints. Con este procedimiento el número de parámetros es $|\theta| + (|\mu_i||\sigma_i|) \times N$, por lo que aumenta linealmente con cada nuevo datapoint, lo cual es un problema si se tienen millones de datapoints.

10.3. Inferencia variacional amortizada

Como todos los problemas en DL, este se resuelve con otra red neuronal. En este caso, se intenta aprender una red $q_i(z) = q(z|x_i) \approx p(z|x_i)$, en vez de aprender μ_i y σ_i .

Por lo que ahora hay una red que va de $z \mapsto p_{\theta}(x|z)$ y otra que va de $x \mapsto q_{\phi}(z|x) = N(\mu_{\phi}(x), \sigma_{\phi}(x))$.

De esta forma se consigue que el número de parámetros sea independiente al número de datapoints. Este método se llama Inferencia variacional amortizada (*Amortized variational inference*).

El objetivo sigue siendo, como antes, acotar:

$$\log p(x_i) \geq E_{z \sim q_{\phi}(z|x_i)} [\log p_{\theta}(x_i|z) + \log p(z)] + H(q_{\phi}(z|x_i)) \quad (10.19)$$

donde ahora se tiene q_{ϕ} .

Para saber calcular $\nabla_{\phi} L$, vamos a escribir L_i y ver que términos dependen de ϕ :

$$L_i = E_{z \sim q_{\phi}(z|x_i)} [\log p_{\theta}(x_i|z) + \log p(z)] + H(q_{\phi}(z|x_i)) \quad (10.20)$$

Algoritmo 19: Inferencia variacional amortizada

```

para cada  $x_i$  (o mini-batch) hacer
    Calcular  $\nabla_{\theta} L(p_{\theta}(x_i|z), q_{\phi}(z|x_i))$ :
        Muestrear  $z \sim q_{\phi}(z|x_i)$ 
         $\nabla_{\theta} L \approx \nabla_{\theta} \log p_{\theta}(x_i|z)$ 
         $\theta \leftarrow \theta + \alpha \nabla_{\theta} L$ 
         $\phi \leftarrow \phi + \alpha \nabla_{\phi} L$ 
fin

```

Se ve que aparece en la distribución de la esperanza y en la entropía. La entropía es sencilla, en el caso de que $q_{\phi}(z|x_i)$ sea una gaussiana, se mira la expresión de la entropía para una gaussiana en internet y ya.

En cuanto a la esperanza, se puede ver que los parámetros no influyen en el interior sino en la distribución. Se aprecia que es lo mismo que pasaba en el objetivo de Policy Gradients, donde:

$$J(\phi) = E_{z \sim q_{\phi}(z|x_i)}[r(x_i, z)] \quad (10.21)$$

Por lo que se puede usar el mismo truco que se usaba en ese tema para entrenar la red neuronal:

$$J(\phi) \approx \frac{1}{M} \sum_j \nabla_{\phi} \log q_{\phi}(z_j|x_i) r(x_i, z_j) \quad (10.22)$$

Lo malo es que los mismos problemas de alta varianza que se veían en el tema de Policy Gradients también ocurren aquí. Se pueden aplicar los trucos vistos en el tema 3, pero además se puede aplicar otro más. A diferencia de Policy Gradients, aquí no hay una noción de dinámicas desconocidas, ya que se conoce todo: los parámetros de q_{ϕ} y sus derivadas. Por lo que se puede usar lo que se conoce como el truco de la reparametrización (*reparametrization trick*). Consiste en calcular $z = \mu_{\phi}(x) + \epsilon \sigma_{\phi}(x)$, donde $\epsilon \sim N(0, 1)$, por lo que:

$$J(\phi) = E_{z \sim q_{\phi}(z|x_i)}[r(x_i, z)] \quad (10.23)$$

$$= E_{\epsilon \sim N(0, 1)}[r(x_i, \mu_{\phi}(x_i) + \epsilon \sigma_{\phi}(x_i))] \quad (10.24)$$

Y la distribución de la esperanza pasa a ser independiente de ϕ . Ahora para calcular el gradiente:

- Se muestran $\epsilon_1, \dots, \epsilon_M$ de $N(0, 1)$.
- Se calcula el gradiente: $\nabla_{\phi} J(\phi) \approx \frac{1}{M} \sum_j \nabla_{\phi} r(x_i, \mu_{\phi}(x_i) + \epsilon_j \sigma_{\phi}(x_i))$

Este estimador tiene tan poca varianza que normalmente es suficiente con coger solamente una ϵ .

Otra forma de ver el truco de la reparametrización. En la práctica es muy útil expresarlo mediante la divergencia KL ya que facilita la implementación.

$$L_i = E_{z \sim q_{\phi}(z|x_i)}[\log p_{\theta}(x_i|z) + \log p(z)] + H(q_{\phi}(z|x_i)) \quad (10.25)$$

$$= E_{z \sim q_{\phi}(z|x_i)}[\log p_{\theta}(x_i|z)] + E_{z \sim q_{\phi}(z|x_i)}[\log p(z)] + H(q_{\phi}(z|x_i)) \quad (10.26)$$

$$E_{z \sim q_{\phi}(z|x_i)}[\log p(z)] + H(q_{\phi}(z|x_i)) = -D_{KL}(q_{\phi}(z|x_i) \| p(z)) \quad (10.27)$$

$$= E_{z \sim q_{\phi}(z|x_i)}[\log p_{\theta}(x_i|z)] - D_{KL}(q_{\phi}(z|x_i) \| p(z)) \quad (10.28)$$

$$= E_{\epsilon \sim N(0, 1)}[\log p_{\theta}(x_i|\mu_{\phi}(x_i) + \epsilon \sigma_{\phi}(x_i))] - D_{KL}(q_{\phi}(z|x_i) \| p(z)) \quad (10.29)$$

$$\approx \log p_{\theta}(x_i|\mu_{\phi}(x_i) + \epsilon \sigma_{\phi}(x_i)) - D_{KL}(q_{\phi}(z|x_i) \| p(z)) \quad (10.30)$$

Por lo que el proceso queda:

1. Se mete x_i en la red parametrizada por ϕ y se obtiene $\mu_{\phi}(x_i)$ y $\sigma_{\phi}(x_i)$

2. Se genera $\epsilon \in N(0, 1)$.
3. Se calcula $\mu_\phi(x_i) + \epsilon\sigma_\phi(x_i) = z$
4. Se mete z como la entrada a la otra red parametrizada por θ , lo que devuelve $p_\theta(x_i|z)$.
5. Se recogen todos los términos necesarios (intermedios y el final) para calcular L y calcular los gradientes y se aplica descenso por gradiente.

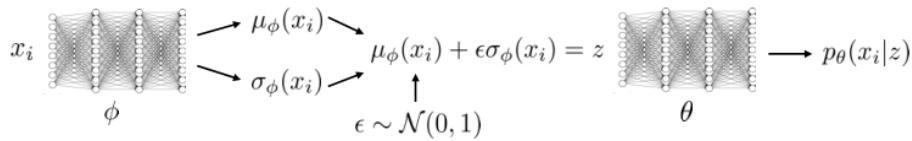


Figura 10.2: Implementación del pipeline de inferencia variacional amortizada con el truco de la reparametrización

10.3.1. Policy Gradient vs el truco de la reparametrización

Policy Gradients:

- +: Puede manejar tanto variables latentes continuas como discretas.
- -: Alta varianza, lo que requiere muchas muestras y *learning rates* pequeños.

Truco de la reparametrization:

- -: Sólo se puede usar para variables latentes continuas.
- +: Muy sencillo de implementar.
- +: Varianza pequeña.

10.4. Modelos generativos: autoencoder variacional

Es uno de los modelos más sencillos que pueden usar el truco anterior. Es un modelo de DL. Está formada por un encoder y un decoder.

Para entrenarlo:

$$\max_{\theta, \phi} \frac{1}{N} \sum_j \log p_\theta(x_i | \mu_\phi(x_i) + \epsilon\sigma_\phi(x_i)) - D_{KL}(q_\phi(z|x_i) \| p(z)) \quad (10.31)$$

Ejemplo de implementación: <https://github.com/noctrog/conv-vae>

10.4.1. Modelos condicionales

Al principio del tema se comentaba que también están los modelos basados en $p(y|x)$. Los pasos para calcular la cota inferior es el mismo pero todo ahora depende de x_i :

$$L_i = E_{z \sim q_\phi(z|x_i, y_i)} [\log p_\theta(y_i|x_i, z) + \log p(z|x_i)] + H(q_\phi(z|x_i, y_i)) \quad (10.32)$$

Los modelos condicionados se usan por ejemplo en el caso de 1.11.2, donde se inyecta ruido para que el modelo escoja una acción.

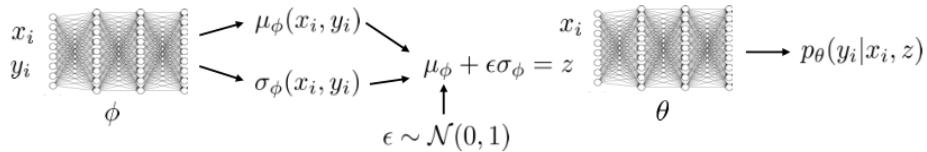


Figura 10.3: Pipeline con la distribución condicionada

10.5. Ejemplos

Publicación: Embed to Control: A Locally Linear Latent Dynamics Model for Control from Raw Images. El modelo de esta publicación es un tipo de autoencoder variacional que tiene dinámica en el espacio latente (no es gaussiano, tiene trayectorias). En la publicación se visualiza la topología de z .

Publicación: SOLAR: Deep Structured Latent Representations for Model-Based Reinforcement Learning. El modelo es un poco más complejo pero también funciona con autoencoders para controlar un robot.

También se puede usar para realizar exploración y modelar el comportamiento humano.

Tema 11

Aprendizaje de política basado en modelo

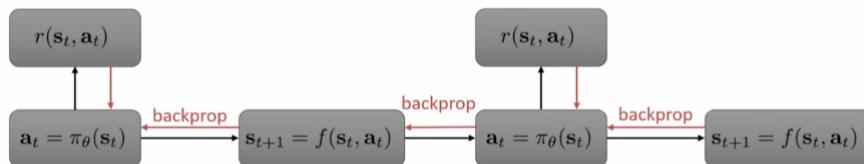
Clase 13: Model Based Policy Learning

2020-07-02

En el tema 9 se habló de algoritmos basados en modelo en los que se planificaban las acciones y por tanto no había política. En este tema se usan políticas. Se tratarán tanto políticas globales (las normales de siempre) y las políticas locales (políticas que solo aproximan valores correctos en un subconjunto de estados).

En el tema 9 al último algoritmo al que se llegó fue el MPC (página 57).

Si se quiere entrenar un modelo de la dinámica a la vez que una política, ambas representadas con una red neuronal, se podría pensar que entrenar de esta manera: fuera una buena idea. Pero tiene



el problema que los episodios al poder tener cientos de pasos aparece el problema de los *vanishing gradient* o *exploding gradients*. Además, las acciones que se tomen al principio afectarán mucho más a los estados finales (efecto mariposa) lo que lleva a la inestabilidad numérica (como los *shooting methods*). Por lo que no se pueden utilizar métodos como LQR. Los problemas son parecidos a los del entrenamiento de las redes neuronales recurrentes con *backpropagation*, pero no se puede jugar con los valores propios de las jacobianas ya que se tienen que aprender las dinámicas del mundo y no se pueden modificar los datos.

Como posibles soluciones:

- Se pueden usar algoritmos RL sin derivadas (sin modelo), donde el modelo es usado para generar muestras sintéticas. Puede parecer una idea absurda pero en la práctica funciona bastante bien. Se puede pensar en ello como una 'aceleración basada en modelo' para algoritmos RL sin modelo.
- Usar políticas más sencillas que redes neuronales. Por ejemplo: LQR con modelos aprendidos (LQR-FLM). Donde se entran políticas locales para resolver tareas simples. También se puede usar aprendizaje supervisado para combinarlas y conseguir una política global.

11.1. Optimización sin modelo con un modelo

Mirando a la fórmula de Policy Gradient:

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_{i,t}|s_{i,t}) \hat{Q}_{i,t}^{\pi} \quad (11.1)$$

Es peculiar ver que no hace falta hacer ninguna regla de la cadena pero aún así Policy Gradients optimiza a través del tiempo. La clave está en que es la derivada de una esperanza. El gradiente de la retropropagación del camino se expresa como:

$$\nabla_{\theta} J(\theta) = \sum_{t=1}^T \frac{dr_t}{ds_t} \prod_{t'=2}^t \frac{ds_{t'}}{da_{t'-1}} \frac{da_{t'-1}}{ds_{t'-1}} \quad (11.2)$$

Esto es un poco inútil. Pero con el truco de la reparametrización del tema anterior aplicada a esta ecuación en entornos estocásticos, calcula el mismo gradiente que Policy Gradient, pero de una manera diferente, por lo que hay ventajas y desventajas para cada uno. PG tiene una alta varianza y BT no tiene ese problema pero para episodios largos es más probable que la aproximación numérica no sea buena.

Si se tuviera un modelo del entorno, se podrían generar millones de muestras y reducir así la varianza de PG. Por lo que PG se vuelve más estable ya que no hay que multiplicar ninguna serie jacobiana.

En la publicación: *Parmas et al. '18: PIPP: Flexible Model-Based Policy Search Robust to the Curse of Chaos*, dicen que aunque se pueda usar BT, suele ser preferible usar PG.

Este método (usando PG) también tiene sus problemas cuando los episodios son largos. Estos vienen de que el modelo del entorno no es perfecto y va introduciendo errores que se van acumulando y hacen que los datos producidos al final no valgan. Lo que hace que la estimación de PG no sea buena.

Aquí es donde entran los métodos 'Dyna'. Consiste en un algoritmo Online Q-Learning que hace RL sin modelo con un modelo.

Algoritmo 20: Dyna

Dado el estado s , escoger una acción a usando la política exploratoria

Observar s' y r , obtener la transición (s, a, s', r) .

Actualizar el modelo $\hat{p}(s'|s, a)$ y $\hat{r}(s, a)$ usando (s, a, s')

Actualización de Q: $Q(s, a) \leftarrow Q(s, a) + \alpha E_{s', r}[r + \max_{a'} Q(s', a') - Q(s, a)]$

repetir

Muestrear $(s, a) \sim B$ del buffer de acciones y estados pasados

Actualización de Q: $Q(s, a) \leftarrow Q(s, a) + \alpha E_{s', r}[r + \max_{a'} Q(s', a') - Q(s, a)]$

hasta que K veces;

El modelo se usa para calcular las esperanzas. Este es el algoritmo propuesto en el paper original por R. Sutton. Hoy en día la generalización del algoritmo que se suele usar es la siguiente:

Las ventajas de este método son:

- Los *rollouts* no tienen por qué ser muy grandes. Normalmente con $N = 1$ está bien.
- Se exploran estados diversos.

Algoritmo 21: Dyna Generalizado

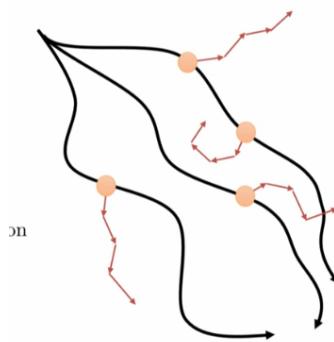
Recolectar algunos datos, consistiendo en las transiciones (s, a, s', r)

Aprender el modelo $\hat{p}(s'|s, a)$ (y opcionalmente $\hat{r}(s, a)$).

repetir

- Muestrear $s \sim B$ del buffer
- Elegir la acción a (de B , a partir de π o aleatoriamente)
- Simular $s' \sim \hat{p}(s'|s, a)$ (y $r = \hat{r}(s, a)$)
- Entrenar sobre (s, a, s', r) el algoritmo RL sin modelo
- (Opcionalmente) tomar N pasos más con el modelo.

hasta que K veces;



Como desventajas de estos métodos, es difícil saber cuando usarlos es mejor que usar algoritmos sin modelo. Además si el modelo es malo se tienen actualizaciones malas.

11.2. Modelos locales

Cuando se habló de control óptimo, se vió que se puede reescribir el problema de control como una optimización sin restricciones.

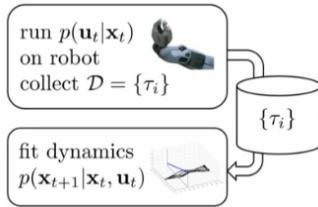
$$\min_{u_1, \dots, u_T} \sum_{t=1}^T c(x_t, u_t) \text{ s.t. } x_t = f(x_{t-1}, u_{t-1}) \quad (11.3)$$

$$\min_{u_1, \dots, u_T} c(x_1, u_1) + c(f(x_1, u_1), u_2) + \dots + c(f(f(\dots)), u_T) \quad (11.4)$$

Donde se necesita $\frac{df}{dx_t}$, $\frac{df}{du_t}$, $\frac{dc}{dx_t}$, $\frac{dc}{du_t}$. En RL, no se tiene el conocimiento de las primeras dos de estas derivadas. Por lo que si se aprendiese un modelo para poder calcular las derivadas sobre ese modelo, se podría entonces utilizar algoritmos como LQR y obtener políticas locales.

La idea es sencilla: se tiene una política y con suerte las trayectorias que produce están cerca las unas de las otras, por lo que se pueden coger esas trayectorias y ajustar una dinámica lineal (regresión lineal) para cada paso del tiempo.

El procedimiento es ejecutar el controlador para conseguir unas trayectorias y ajustar las dinámicas:



Las dinámicas serán unas gaussianas condicionadas:

$$p(x_{t+1}|x_t, u_t) = N(f(x_t, u_t), \Sigma) \quad (11.5)$$

La media de esas gaussianas es una función lineal.

$$f(x_t, u_t) \approx A_t x_t + B_t u_t \quad (11.6)$$

Σ no hace falta aproximarla ya que no afecta a LQR. Con esto, se tiene:

$$A_t = \frac{df}{dx_t} \quad B_t = \frac{df}{du_t} \quad (11.7)$$

Se mejora el controlador y se repite el proceso.

Para formular un algoritmo práctico de este tipo se tiene que resolver la pregunta de qué controlador se tiene que ejecutar.

iLQR produce $\hat{x}_t, \hat{u}_t, K_t, k_t$, y:

$$u_t = K_t(x_t - \hat{x}_t) + k_t + \hat{u}_t \quad (11.8)$$

Se puede pensar que una acción de control razonable sería $p(u_t|x_t) = \delta(u_t = \hat{u}_t)$. Pero esto no corrige ni las desviaciones ni la deriva. Por lo que en su lugar se puede pensar en coger:

$$p(u_t|x_t) = \delta(u_t = K_t(x_t - \hat{x}_t) + k_t + \hat{u}_t) \quad (11.9)$$

Donde K_t sirve para compensar los pequeños errores del modelo. Pero la aproximación lineal puede ser un problema al ser demasiado mala en el caso de que todas las trayectorias sean muy parecidas o estén muy dispersas.

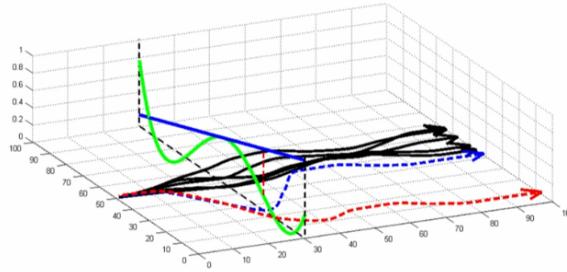
Una mejor elección sería construir un controlador estocástico tal que:

$$p(u_t|x_t) = N(K_t(x_t - \hat{x}_t) + k_t + \hat{u}_t, \Sigma_t) \quad (11.10)$$

Una buena elección de Σ_t es $\Sigma_t = Q_{u_t, u_t}^{-1}$. Una intuición de esto es que Q modela la curvatura local de la función Q . Por lo que en curvaturas agudas no se quiere una varianza alta y viceversa.

¿Cómo se aprenden las dinámicas (A y B)? Con regresión lineal se tiene el problema de que los datos requeridos crecen con la dimensionalidad del problema. En la práctica esto se puede simplificar usando **regresión lineal bayesiana** usando nuestro modelo global favorito (red neuronal, mezcla de gaussianas, ...) como el conocimiento a priori.

¿Qué pasa si nos vamos muy lejos? Como la aproximación es lineal, si nos alejamos de la aproximación el controlador tomará acciones malas y este error se acentuará en estados futuros.



Para que este método funcione bien, se tiene que conseguir que la nueva distribución de trayectorias $p(\tau)$ esté cerca de la distribución anterior $\bar{p}(\tau)$. Tal como se vio en TRPO donde se restringía la divergencia KL para mantener las distribuciones similares entre sí, aquí se va a hacer lo mismo. Ya que si las trayectorias son similares, las dinámicas también lo serán. Que las distribuciones estén cerca significa que $D_{KL}(p(\tau)||\bar{p}(\tau)) \leq \epsilon$. Resulta que esto es muy fácil de hacer si $\bar{p}(\tau)$ vino también de un controlador lineal.

Para más detalles, mirar el paper: *Learning Neural Network Policies with Guided Policy Search under Unknown Dynamics*

11.3. Combinar modelos locales en un modelo global

Se le llama búsqueda de política guiada. Consiste en generar varias trayectorias con algoritmos sencillos basados en modelo como por ejemplo iLQR y con todas las muestras obtenidas entrenar una red neuronal para que reproduzca el modelo. Esto funciona bien pero se puede mejorar si una vez se ha entrenado la red se modifican los controladores LQR para que intenten aproximarse a la política global de la red neuronal. Esto hace que los diversos controladores lleguen a un consenso y se converja a una política global buena.

Algoritmo 22: Búsqueda guiada de política

repetir

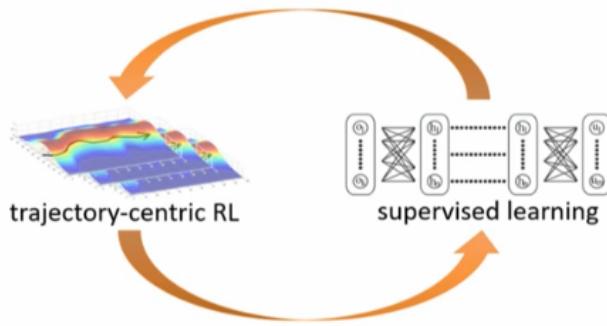
Optimizar cada política global $\pi_{LQR,i}(u_t|x_t)$ en el estado inicial $x_{0,i}$ con respecto a $\tilde{c}_{k,i}(x_t, u_t)$.

Usar las muestras del paso anterior para entrenar π_θ de forma que mimetice cada $\pi_{LQR,i}(u_t|x_t)$

Actualizar la función de coste $\tilde{c}_{k+1,i}(x_t, u_t) = c(x_t, u_t) - \lambda_{k+1,i} \log \pi_\theta(u_t|x_t)$

hasta que;

El coste $\tilde{c}_{k,i}$ será modificado levemente para mantener $\pi_{LQR,i}$ cerca de π_θ . $\lambda_{k+1,i}$ puede ser visto como un multiplicador de Lagrange para la restricción que se asegura que en la convergencia π_θ hace lo mismo que π_{LQR} .



Esta idea puede ser utilizada en otros ámbitos de RL, como por ejemplo RL sin modelo, o en ámbitos fuera de RL como por ejemplo la técnica de la destilación (*distillation*).

Esta técnica viene del mundo del aprendizaje supervisado para entrenar redes neuronales. La técnica se aplica para evitar los **ensemble models**: que consiste en entrenar varios modelos y calcular la media de sus predicciones, en vez de tener solamente un modelo con una única predicción.

La técnica de la destilación se basa en entrenar un modelo sobre las predicciones de los otros modelos como 'objetivos blandos' (*soft-targets*). La idea está en que los *labels* del aprendizaje supervisado no dan mucha información. Por ejemplo en MNIST pueden haber números que estén mal escritos y se parezcan a otros, por lo que los modelos no estarán 100 % seguros de su decisión aunque el *label* es 100 % cierto.

La extensión a RL se llama *Distillation for Multi-Task Transfer*. Era una publicación en la que se pretendía tener un agente que jugase a todos los juegos de Atari. Se entrena un agente por cada juego y luego se crea otro agente que los engloba a todos usando aprendizaje supervisado y destilación.

Otra técnica diferente a la destilación es la de 'divide y vencerás', en la cual en vez de tener algoritmos LQR para las políticas locales se tienen más redes neuronales. Un sketch del algoritmo sería

Algoritmo 23: Divide and conquer RL

repetir

- Optimizar cada política local $\pi_{\theta_i}(a_t|s_t)$ sobre los estados iniciales $s_{0,i}$ con respecto a $\tilde{r}_{k,i}(s_t, a_t)$.
- Usar las muestras del paso anterior para entrenar $\pi_\theta(a_t|x_t)$ para mimetizar cada $\pi_{\theta_i}(a_t|s_t)$.
- Actualizar la función de recompensa $\tilde{r}_{k+1,i}(x_t, u_t) = r(x_t, u_t) + \lambda_{k+1,i} \log \pi_\theta(u_t|x_t)$

hasta que;

Publicaciones relacionadas:

- L. *, Finn *, et al. End-to-End Training of Deep Visuomotor Policies. 2015.
- Rusu et al. Policy Distillation. 2015.
- Parisotto et al. Actor-Mimic: Deep Multitask and Transfer Reinforcement Learning. 2015.
- Ghosh et al. Divide-and-Conquer Reinforcement Learning. 2017.
- Teh et al. Distral: Robust Multitask Reinforcement Learning. 2017.

Tema 12

Reformulación del Control como un problema de inferencia

Clase 14: Reframing Control as an Inference Problem

2020-07-04

12.1. ¿El control óptimo y RL son capaces de proveer un modelado razonable del comportamiento humano?

Este tema será un poco más teórico y el siguiente más práctico.

12.1.1. Control Óptimo como un modelo del comportamiento humano

Desde hace mucho tiempo (Muybridge, 1870) se ha intentado explicar el comportamiento humano como una búsqueda de la optimalidad de una tarea, como por ejemplo correr minimizando la energía utilizada. Hoy en día se puede modelar bastante bien (clase de hoy) y también hay ejemplos del modelado probabilístico de conductores de taxi.

La idea es la siguiente: si se tiene una función de recompensa y una forma de modelar el mundo o interactuar con él se puede extraer una planificación óptima resolviendo un problema de optimización.

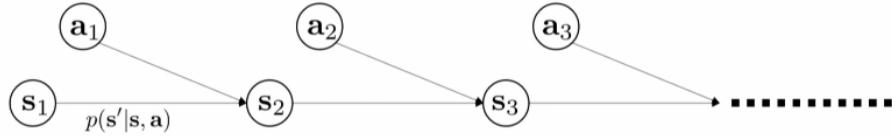
Pero el comportamiento de las personas y animales es más complejo. Por ejemplo si se tiene a un mono y se le pide que lleve un punto en la pantalla hasta otro que es el destino mediante un joystick, no hará una trayectoria en línea recta perfecta, que sería lo más óptimo. En su lugar haría una trayectoria parecida pero curva, ya que es lo más fácil de hacer llegando aún así al destino. Incluso las trayectorias se verán influenciadas por el día y su estado de ánimo. En resumen, su comportamiento es estocástico, pero las buenas acciones son las más probables.

Teniendo esto en cuenta, hay que tirar a la basura los objetivos que se tenían antes:

$$a_1, \dots, a_T = \arg \max_{a_1, \dots, a_T} \sum_{t=1}^T r(s_t, a_t) \quad (12.1)$$

$$s_{t+1} = f(s_t, a_t) \quad (12.2)$$

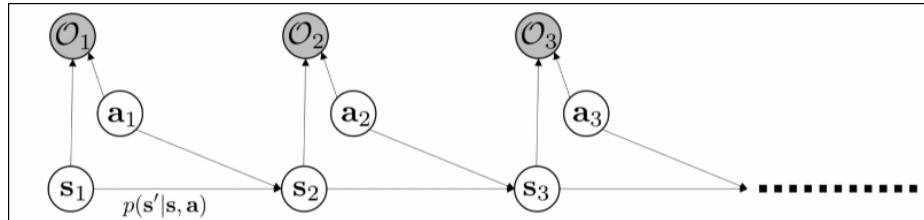
ya que únicamente dan como óptimo la línea recta. Para explicar las acciones aleatorias del mono, se necesita un modelo gráfico:



Se puede escribir la probabilidad de ver una secuencia de estados con otra secuencia de acciones:

$$p(s_{1:T}, a_{1:T}) = ?? \text{ no existe ninguna asunción de comportamiento óptimo} \quad (12.3)$$

El modelo que se tiene no dice que los pares de acciones-estados sean óptimos, simplemente si son físicamente posibles. Por lo que se le añade al grafo los nodos O (por optimalidad) para cada paso en el tiempo. Lo que representan estas variables es si se está siendo óptimo o no, por lo que es una variable binaria. También se puede pensar en esta variable como la intencionalidad.



Para modelar el comportamiento del mono, se introduce lo siguiente:

$$p(\tau|O_{1:T}) \quad p(O_t|s_t, a_t) = \exp(r(s_t, a_t)) \quad (12.4)$$

El término de la izquierda significa cuál es la probabilidad de la trayectoria τ si nuestro mono siempre se comporta de forma óptima. Para el de la derecha, se elige arbitrariamente la probabilidad del comportamiento óptimo para ese paso en el tiempo como la exponencial de la recompensa si se toma la acción a_t . Si se quiere normalizar se puede restar por la exponencial de la recompensa más grande. Esta elección arbitraria tendrá sentido más adelante. Con lo que se tiene en 12.4, se deriva:

$$p(\tau|O_{1:T}) = \frac{p(\tau, O_{1:T})}{p(O_{1:T})} \quad (12.5)$$

$$\propto p(\tau) \prod_t \exp(r(s_t, a_t)) = p(\tau) \exp\left(\sum_t r(s_t, a_t)\right) \quad (12.6)$$

Esto es interesante porque puede modelar comportamientos subóptimos, lo cual es importante para RL inverso. También se puede aplicar para resolver problemas de control y planificación. Además da una explicación de porque un comportamiento estocástico puede ser más preferible que uno determinista (útil para la exploración y *transfer learning*).

12.1.2. Inferencia en este modelo

El modelo gráfico anterior tiene dos tipos de distribuciones condicionales:

- Dinámicas: $p(s_{t+1}|s_t, a_t)$
- Exponencial de la recompensa: $p(O_t|s_t, a_t) \propto \exp(r(s_t, a_t))$

Además para el primer estado se tiene $p(s_1)$.

Para realizar la inferencia, se hacen tres preguntas:

1. ¿Cómo se calcula un mensaje hacia atrás?

Un mensaje hacia atrás se define como $\beta_t(s_t, a_t) = p(O_{t:T}|s_t, a_t)$ y es la probabilidad de que las variables de optimalidad O desde el tiempo actual t hasta el final T sean verdaderas dado (s_t, a_t) . Parece una pregunta un tanto arbitraria pero con esto se puede recuperar una política casi óptima en este modelo.

2. ¿Cómo se calcula la política $p(a_t|s_t, O_{1:T})$? Esta distribución condicional es equivalente a la política óptima bajo este modelo gráfico. Resuelve la pregunta de: dado el estado actual y dado a que todas futuras acciones sean tomadas óptimamente, cuál es la probabilidad de tomar la acción a_t .

Resolver la pregunta 1 ayuda a resolver la pregunta 2.

3. ¿Cómo se calcula un mensaje hacia adelante $\alpha_t(s_t) = p(s_t|O_{1:t-1})$? Un mensaje hacia adelante te da la probabilidad de acabar en el estado s_t sabiendo que se ha tenido un comportamiento óptimo en todos los estados anteriores. No se necesita para calcular la política. Pero es muy útil para saber en qué estados la política óptima cae. Son necesarios para resolver RL inverso.

Mensajes hacia atrás

De los tres anteriores son los más útiles.

Como se dijo, se definen como:

$$\beta_t(s_t, a_t) = p(O_{t:T}|s_t, a_t) \quad (12.7)$$

Se va a expresar β en función de la probabilidad de transición y la probabilidad de la optimalidad, y como esto es un proceso iterativo, también en β_{t+1} .

$$\beta_t(s_t, a_t) = \int p(O_{t:T}, s_{t+1}|s_t, a_t) ds_{t+1} \quad (12.8)$$

$$= \int p(O_{t+1:T}|s_{t+1}) p(s_{t+1}|s_t, a_t) p(O_t|s_t, a_t) ds_{t+1} \quad (12.9)$$

El paso de 12.8 a 12.9 se puede hacer aplicando la regla de la cadena de la probabilidad porque O_t es independiente de O_{t+1} dado s_{t+1} . De 12.9, se conoce $p(O_t|s_t, a_t)$ (la recompensa), $p(s_{t+1}|s_t, a_t)$ también se conoce por lo que nos quedamos solo con $p(O_{t+1:T}|s_{t+1})$. A este término se le 'inserta la acción':

$$p(O_{t+1:T}|s_{t+1}) = \int p(O_{t+1:T}|s_{t+1}, a_{t+1}) p(a_{t+1}|s_{t+1}) da_{t+1} \quad (12.10)$$

El primer término del producto es igual a $\beta_t s_{t+1}, a_{t+1}$. El segundo, $p(a_{t+1}|s_{t+1})$ nos dice que probabilidad tenemos de poder tomar cualquier acción sobre el estado s_{t+1} . Como prácticamente se pueden tomar todas las acciones que sean físicamente posibles, se puede simplificar como una distribución uniforme. Esta asunción no reduce la generalidad de esta formulación.

Por ahora se está asumiendo que se conocen todas las probabilidades (modelo, ...), en la práctica se tendrán que tomar muestras.

Algoritmo 24: Paso hacia atrás

```

para t = T - 1 hasta 1 hacer
    |   β_t(s_t, a_t) = p(O_t|s_t, a_t) E_{s_{t+1} ~ p(s_{t+1}|s_t, a_t)} [β_{t+1}(s_{t+1})]
    |   β_t(s_t) = E_{a_t ~ p(a_t|s_t)} [β_t(s_t, a_t)]
fin

```

Se puede pensar que el segundo paso tiene un aspecto familiar. Se introducen las siguientes variables arbitrariamente.

$$\begin{aligned} V_t(s_t) &= \log \beta_t(s_t) \\ Q_t(s_t, a_t) &= \log \beta_t(s_t, a_t) \end{aligned} \quad (12.11)$$

Que son básicamente los mensajes hacia atrás en espacio logarítmico. Expandiendo con la definición, se tiene que:

$$V_t(s_t) = \log \int \exp(Q_t(s_t, a_t)) da_t \quad (12.12)$$

Si se tienen valores de Q muy grandes para una cierta acción, este silenciará a los valores más pequeños ya que las diferencias debido a la exponencial serán muy grandes. Este efecto se ve aún más acentuado después de aplicar el logaritmo. Por lo que según $Q_t(s_t, a_t)$ se va haciendo más grande, $V_t(s_t) \rightarrow \max_{a_t} Q_t(s_t, a_t)$. Esto será de utilidad más adelante.

Para el primer paso, al expresarlo en el espacio logarítmico:

$$Q_t(s_t, a_t) = r(s_t, a_t) + \log E[\exp(V_{t+1}(s_{t+1}))] \quad (12.13)$$

Como se ha comentado antes, V_{t+1} tenderá a ser $\max_{a_t} Q_{t+1}(s_t, a_t)$ según Q_{t+1} vaya creciendo. Junto a esto, si se quitan el logaritmo y la exponencial, el paso es el mismo que el primer paso que en el algoritmo de iteración del valor, donde:

$$Q(s, a) \leftarrow r(s, a) + \gamma E[V(s')] \quad (12.14)$$

Con dinámicas deterministas, el logaritmo y la exponencial se cancelan y efectivamente coincide con este paso.

Para el caso estocástico, esta formulación es un problema porque describe un comportamiento que no se quiere ver en control óptimo. Esto es porque la transición se convierte en optimista, lo cual no es una buena idea.

Antes se comentó que como simplificación se puede tener que $p(a_{t+1}|s_{t+1})$ fuera uniforme. En el caso de que no se quiera que sea uniforme:

$$V(s_t) = \log \int \exp(Q(s_t, a_t) + \log p(a_t|s_t)) a_t \quad (12.15)$$

$$Q(s_t, a_t) = r(s_t, a_t) + \log E[\exp(V(s_{t+1}))] \quad (12.16)$$

$$\tilde{Q}(s_t, a_t) = r(s_t, a_t) + \log p(a_t|s_t) + \log E[\exp(V(s_{t+1}))] \quad (12.17)$$

$$V(s_t) = \log \int \exp(\tilde{Q}(s_t, a_t)) a_t \Leftrightarrow V(s_t) = \log \int \exp(Q(s_t, a_t) + \log p(a_t|s_t)) a_t \quad (12.18)$$

Por lo que no cambia mucho la formulación.

Calcular la política

Se corresponde con la pregunta 'de verdad', ¿cómo calcular la política óptima a partir de este modelo gráfico $p(a_t|s_t, O_{1:T})$?

Lo que se quiere buscar esencialmente es la política:

$$p(a_t|s_t, O_{1:T}) = \pi(a_t|s_t) \quad (12.19)$$

$$= p(a_t|s_t, O_{t:T}) \quad (12.20)$$

$$= \frac{p(a_t, s_t | O_{t:T})}{p(s_t | O_{t:T})} \quad (12.21)$$

$$= \frac{p(O_{t:T} | a_t, s_t) p(a_t, s_t) / p(O_{t:T})}{p(O_{t:T} | s_t) p(s_t) / p(O_{t:T})} \quad (12.22)$$

$$= \frac{p(O_{t:T} | a_t, s_t)}{p(O_{t:T} | s_t)} \frac{p(a_t, s_t)}{p(s_t)} = \frac{p(O_{t:T} | a_t, s_t)}{p(O_{t:T} | s_t)} p(a_t | s_t) \quad (12.23)$$

$$= \frac{\beta_t(s_t, a_t)}{\beta_t(s_t)} p(a_t | s_t) \quad (12.24)$$

Como $p(a_t|s_t)$ se considera como uniforme, se puede expresar:

$$\pi(a_t|s_t) = \frac{\beta_t(s_t, a_t)}{\beta_t(s_t)} \quad (12.25)$$

Sustituyendo β por las variables V y Q se obtiene:

$$\pi(a_t|s_t) = \exp(Q_t(s_t, a_t) - V_t(s_t)) = \exp(A_t(s_t, a_t)) \quad (12.26)$$

Por lo que la política es simplemente la exponencial del *advantage*. También se puede expresar con temperatura de la siguiente forma:

$$\pi(a_t|s_t) = \exp\left(\frac{1}{\alpha}Q_t(s_t, a_t) - \frac{1}{\alpha}V_t(s_t)\right) = \exp\left(\frac{1}{\alpha}A_t(s_t, a_t)\right) \quad (12.27)$$

Lo que permite convertir la política en estocástica con $\alpha = 1$ o determinista $\alpha = 0$.

Como resumen:

- La interpretación natural es que las acciones mejores son las más probables.
- Si dos acciones son igual de buenas tendrán la misma probabilidad.
- Análogo a la exploración de Boltzmann.
- Se aproxima a la política voraz según va decreciendo la temperatura.

Mensajes hacia adelante

Los mensajes hacia adelante es la probabilidad de acabar en un estado dado que se hayan realizado acciones óptimas hasta ese momento, se va a realizar un procedimiento de expansión parecido al anterior:

$$\alpha_t(s_t) = p(s_t|O_{1:t-1}) \quad (12.28)$$

$$= \int p(s_t, s_{t-1}, a_{t-1}|O_{1:t-1}) ds_{t-1} da_{t-1} = \int p(s_t|s_{t-1}, a_{t-1}, O_{1:t-1}) p(a_{t-1}|s_{t-1}, O_{1:t-1}) p(s_{t-1}|O_{1:t-1}) ds_{t-1} da_{t-1} \quad (12.29)$$

$$= \int p(s_t|s_{t-1}, a_{t-1}) p(a_{t-1}|s_{t-1}, O_{t-1}) p(s_{t-1}|O_{1:t-1}) ds_{t-1} da_{t-1} \quad (12.30)$$

De los tres términos de 12.30, se conoce el primero que son las dinámicas pero los otros dos tenemos que trabajarlos:

$$p(a_{t-1}|s_{t-1}, O_{t-1}) p(s_{t-1}|O_{1:t-1}) = \frac{p(O_{t-1}|s_{t-1}, a_{t-1}) p(a_{t-1}|s_{t-1})}{p(O_{t-1}|s_{t-1})} \frac{p(O_{t-1}|s_{t-1}) p(s_{t-1}|O_{1:t-2})}{p(O_{t-1}|O_{1:t-2})} \quad (12.31)$$

Se tiene en cuenta que $p(s_{t-1}|O_{1:t-2}) = \alpha_{t-1}(s_{t-1})$, y que por lo tanto $\alpha_1(s_1) = p(s_1)$, el cual normalmente se conoce.

Ya se han calculado los mensajes hacia adelante. Una vez se tienen estos y los mensajes hacia atrás, se puede calcular $p(s_t|O_{1:T})$.

$$p(s_t|O_{1:T}) = \frac{p(s_t, O_{1:T})}{p(O_{1:T})} = \frac{p(O_{1:T}) p(s_t, O_{1:T-1})}{p(O_{1:T})} \propto \beta_t(s_t) p(s_t|O_{1:T-1}) p(O_{1:T-1}) \propto \beta_t(s_t) \alpha_t(s_t) \quad (12.32)$$

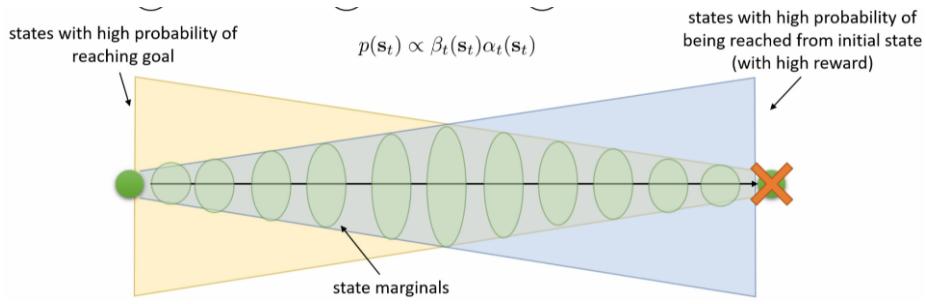


Figura 12.1: Los *state marginals* es el producto de los mensajes hacia adelante y hacia atrás. Normalmente se espera que sean mayores a mitad de la trayectoria. Esto es consistente con experimentos de comportamiento sobre humanos.

12.1.3. Resumen

- Se ha introducido un modelo gráfico para hacer control óptimo.
- Se ha planteado el control como inferencia (como en HMM, filtros de Kalman, ...).
- Muy similar a la programación dinámica, iteración de valor, ... (pero suave).

12.2. ¿Hay una explicación mejor?

12.2.1. El problema del optimismo

Antes se comentó que el logaritmo de 12.13 es demasiado optimista. Esto es porque se está haciendo la pregunta incorrecta. El problema de inferencia es $p(s_{1:T}, a_{1:T}|O_{1:T})$, marginalizando y condicionando, se obtiene $p(a_t|s_t, O_{1:T})$ (que es la política). Esto es el equivalente a preguntar: dado que has obtenido una alta recompensa, ¿cuál fue la probabilidad de tu acción? Parece una pregunta razonable.

Pero también se puede marginalizar y condicionar la dinámica, $p(s_{t+1}|s_t, a_t, O_{1:T}) \neq p(s_{t+1}|s_t, a_t)$. Esto no es igual a la dinámica real del entorno, porque es lo mismo que preguntar: dado que has obtenido una recompensa alta, ¿cuál fue tu probabilidad de transición? Desde el punto de vista de la inferencia es correcto, pero desde el punto de vista del control es curioso, por ejemplo, comprar un ticket de lotería porque no se puede planificar ganarla.

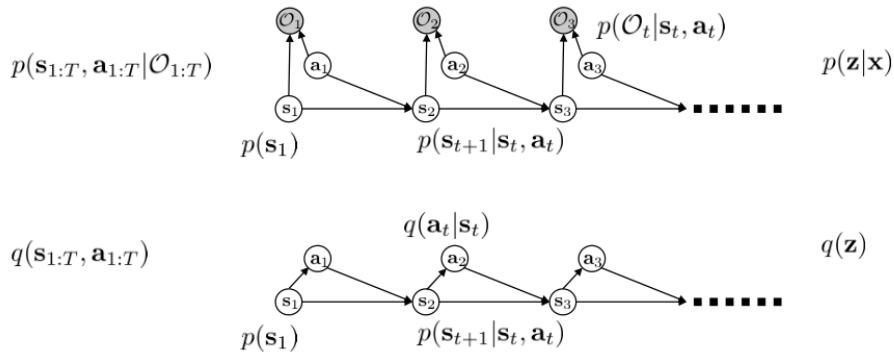
Para arreglar el problema, se quiere resolver la primera pregunta y no la segunda. Esto se consigue preguntando: dado que has obtenido una recompensa alta, ¿cuál fue la probabilidad de la acción tomada, dado que la probabilidad de transición no cambió? Esta es una pregunta difícil de realizar en este problema de inferencia.

Se tiene que encontrar una distribución $q(s_{1:T}, a_{1:T})$ que sea próxima a $p(s_{1:T}, a_{1:T}|O_{1:T})$ pero que tiene la misma dinámica $p(s_{t+1}|s_t, a_t)$. Este problema de encontrar una distribución similar a otra para resolver problemas de inferencia en el tema 10.

Sea $x = O_{1:T}$ y $z = (s_{1:T}, a_{1:T})$. Se busca $q(z)$ que aproxime $p(z|x)$. Se va a hacer con inferencia variacional. Sea:

$$q(s_{1:T}, a_{1:T}) = p(s_1) \prod_t p(s_{t+1}|s_t, a_t) q(a_t|s_t) \quad (12.33)$$

Donde se ve que el estado inicial y la dinámica es la misma que en p (haciendo alusión a la última parte de la pregunta). $q(a_t|s_t)$ es lo que se va a convertir en la política.



Se escribe la cota variacional inferior:

$$\log p(x) \geq E_{z \sim q(z)}[\log p(x, z) - \log q(z)] \quad (12.34)$$

El término de la derecha es $H(q)$ (entropía). Sustituyendo en esta ecuación la p y q que se han elegido para este problema de control se tiene:

$$\begin{aligned} \log p(O_{1:T}) &\geq E_{(s_{1:T}, a_{1:T}) \sim q}[\log p(s_1) + \sum_{t=1}^T \log p(s_{t+1}|s_t, a_t) + \sum_{t=1}^T \log p(O_t|s_t, a_t) \\ &\quad - \log p(s_1) - \sum_{t=1}^T \log p(s_{t+1}|s_t, a_t) - \sum_{t=1}^T \log q(a_t|s_t)] \end{aligned} \quad (12.35)$$

$$= E_{(s_{1:T}, a_{1:T}) \sim q}[\sum_t r(s_t, a_t) - \log q(a_t|s_t)] \quad (12.36)$$

El resultado se empieza a parecer mucho al objetivo de RL. Se puede sacar el sumatorio fuera de la esperanza:

$$= \sum_t E_{(s_t, a_t) \sim q}[r(s_t, a_t) + H(q(a_t|s_t))] \quad (12.37)$$

El objetivo es el mismo que el de RL lo único que se añade la entropía de q . Por lo que al maximizar se está maximizando tanto la recompensa como la entropía.

Algoritmo 25: Paso hacia atrás - variacional

```

para t = T - 1 hasta 1 hacer
    | Q_t(s_t, a_t) = r(s_t, a_t) + E[(V_{t+1}(s_{t+1})]
    | V_t(s_t) = log ∫ exp(Q_t(s_t, a_t))da_t
fin

```

12.3. Q-Learning con soft-optimality

En el algoritmo estándar de Q-Learning, la actualización de los parámetros se hace como: $\phi \leftarrow \phi + \alpha \nabla_\phi Q_\phi(s, a)(r(s, a) + \gamma V(s') - Q_\phi(s, a))$ y el objetivo del valor es: $V(s') = \max_{a'} Q_\phi(s', a')$.

El único cambio es cambiar el máx por softmax:

- Actualización: $\phi \leftarrow \phi + \alpha \nabla_\phi Q_\phi(s, a)(r(s, a) + \gamma V(s') - Q_\phi(s, a))$
- Objetivo del valor: $V(s') = \text{soft max}_{a'} Q_\phi(s', a') = \log \int \exp(Q_\phi(s', a'))da'$

Después de hacer esto, en vez de tener una política voraz se debe de tener una política basada en el exponente del *advantage*.

Algoritmo 26: Q-Learning with soft-optimality

repetir

Tomar una acción a_i y observar (s_i, a_i, s'_i, r_i) , añadirlo a R .
 Sacar un mini-batch $\{s_j, a_j, s'_j, r_j\}$ de R uniformemente.
 Calcular $y_j = r_j + \gamma \text{soft max}_{a'_j} Q_{\phi'}(s'_j, a'_j)$ usando la red objetivo $Q_{\phi'}$
 $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_{\phi}}{d\phi}(s_j, a_j)(Q_{\phi}(s_j, a_j) - y_j)$
 Actualizar ϕ' : copiar ϕ cada N pasos o Polyak
hasta que mientras se siga entrenando;

12.4. Policy Gradient con soft-optimality

$$\pi(a|s) = \exp(Q_{\phi}(s, a) - V(s)) \text{ optimiza } \sum_t E_{\pi(s_t, a_t)}[r(s_t, a_t)] + E_{\pi(s_t)}[H(\pi(a_t|s_t))] \quad (12.38)$$

Intuición:

$$\pi(a|s) \propto \exp(Q_{\phi}(s, a)) \text{ cuando } \pi \text{ minimiza } D_{KL}(\pi(a|s) \parallel \frac{1}{Z} \exp(Q(s, a))) \quad (12.39)$$

$$D_{KL}(\pi(a|s) \parallel \frac{1}{Z} \exp(Q(s, a))) = E_{\pi(a|s)}[Q(s, a)] - H(\pi) \quad (12.40)$$

Normalmente se le llama Entropy Regularized Policy Gradient. Combate contra el colapso de la entropía de Policy Gradient normal, lo que lleva a que la varianza se reduzca mucho al principio y reduzca la exploración.

Esta modificación se utilizaba de forma heurística antes de averiguar el porque.

Está muy relacionado con soft Q-Learning (Haarnoja et al. 2017 y Schulman et al. 2017).

12.5. Beneficios de la optimalidad blanda

- Mejora la exploración y previene los colapsos de entropía
- Es más fácil ajustar políticas más finamente a tareas más específicas
- Rompe los empates de acciones con igual probabilidad
- Más robusto debido a que se ven más estados (porque se maximiza la entropía).
- Se puede reducir a optimalidad dura según la magnitud de la recompensa vaya incrementándose.

Tema 13

Aprendizaje por refuerzo inverso

Clase 15: Inverse Reinforcement Learning

2020-07-06

Hasta ahora todos los algoritmos de RL que se han visto requieren que se genere a mano una función de recompensa para definir la tarea que se quiere realizar.

La idea del RL inverso es aprender una función de recompensa tras observar a un experto, para después poder aplicar aprendizaje por refuerzo sobre esa función de recompensa aprendida.

13.1. ¿Por qué deberíamos preocuparnos en aprender las recompensas?

13.1.1. La perspectiva del aprendizaje por imitación

En el aprendizaje por imitación explicado al comienzo del curso, se le enseña a los agentes pares de estado-acción de los cuales aprenden. Lo que conlleva a que los algoritmos no razonan, simplemente intentan copiar exactamente las acciones del maestro. Por lo que en entornos nuevos fracasan.

Por otro lado el aprendizaje que realizan los humanos se puede realizar simplemente mirando a otros humanos, pudiendo incluso realizar nuevas acciones para realizar la misma tarea de forma más eficiente. Se tiene un conocimiento de lo que se está haciendo.

13.1.2. La perspectiva del aprendizaje por refuerzo

En tareas como los juegos de Atari, la puntuación muchas veces está en la propia pantalla y eso se puede usar como recompensa. Por otro lado, en tareas como la conducción autónoma, la función de recompensa no está clara: se tiene que llegar al destino, respetar las normas, ser educado con los otros coches, no conducir bruscamente para que las personas de dentro no sufran, ...

13.2. Aprendizaje por refuerzo inverso

El problema del aprendizaje por refuerzo inverso consiste en inferir funciones de recompensa a partir de demostraciones.

Por sí mismo, este problema no está especificado ya que hay múltiples funciones de recompensa que puedan explicar un comportamiento. Por lo que hay que introducir otros criterios.

13.2.1. Un poco más formalmente

Aprendizaje por refuerzo normal	Aprendizaje por refuerzo inverso
Dado: estados $s \in S$, acciones $a \in A$ (a veces) transiciones $p(s' s, a)$ función de recompensa $r(s, a)$	Dado: estados $s \in S$, acciones $a \in A$ (a veces) transiciones $p(s' s, a)$ muestras $\{\tau_i\}$ muestreadas de $\pi^*(\tau)$
Aprender $\pi^*(a s)$	Aprender $r_\psi(s, a)$... y después usarla para aprender $\pi^*(a s)$

Para representar a la función r_ψ se pueden utilizar varios métodos, uno de los más usados desde el principio es la función de recompensa lineal:

$$r_\psi(s, a) = \sum_i \psi_i f_i(s, a) = \psi^T f(s, a) \quad (13.1)$$

Pero también se puede representar mediante una red neuronal de entrada (s, a) .

13.2.2. Aprendizaje de la variable de optimalidad

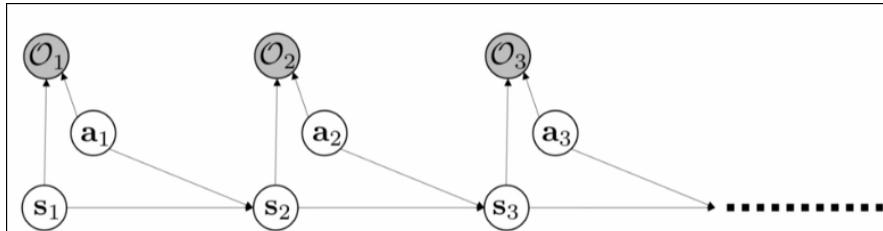
En el tema pasado se introdujo la variable de optimalidad O que indicaba la probabilidad de escoger una trayectoria que sea óptima para resolver el problema. En el tema pasado se vio la inferencia de ese modelo gráfico, en este se va a realizar aprendizaje de r_ψ .

$$p(O_t|s_t, a_t) = \exp(r_\psi(s_t, a_t)) \quad (13.2)$$

$$p(\tau|O_{1:T}, \psi) \propto p(\tau) \exp\left(\sum_t r_\psi(s_t, a_t)\right) \quad (13.3)$$

Se va a intentar maximizar 13.3 con respecto a ψ mediante ML. Se dan las trayectorias $\{\tau_i\}$ muestreadas de $\pi^*(\tau)$.

$$\max_\psi \frac{1}{N} \sum_{i=1}^N \log p(\tau_i|O_{1:T}, \psi) \quad (13.4)$$



Como se está maximizando con respecto a ψ , el término de la dinámica $p(\tau)$ se puede eliminar de 13.3. Por lo que 13.4 se puede expresar como:

$$\max_\psi \frac{1}{N} \sum_{i=1}^N r_\psi(\tau_i) - \log Z \quad (13.5)$$

Donde $\log Z$ es un término que hace que todo sume a 1, ya que las probabilidades con respecto a algo tienen que sumar hasta 1 (en este caso es respecto a τ). Lo que significa que Z es la integral de todas las posibles trayectorias τ de $r_\psi(\tau)$. $\log Z$ es la función de partición (*partition function*). Esto es difícil de calcular porque hay muchas trayectorias posibles (tiempo exponencial en sumarlas todas).

13.2.3. La función de partición de RL inverso

Z se define como:

$$Z = \int p(\tau) \exp(r_\psi(\tau)) d\tau \quad (13.6)$$

La función de partición es necesaria ya que con maximizar la recompensa no basta, hay que maximizarla para que tenga valores más altos que las recompensas de otras trayectorias. Usando esta definición en 13.5 y derivando se obtiene:

$$\nabla_\psi L = \frac{1}{N} \sum_{i=1}^N \nabla_\psi r_\psi(\tau_i) - \frac{1}{Z} \int p(\tau) \exp(r_\psi(\tau)) \nabla_\psi r_\psi(\tau) d\tau \quad (13.7)$$

Sabiendo que:

$$p(\tau|O_{1:T}, \psi) = \frac{1}{Z} \int p(\tau) \exp(r_\psi(\tau)) \quad (13.8)$$

Se puede escribir lo anterior en forma de esperanza:

$$\nabla_\psi L = E_{\tau \sim \pi^*(\tau)} [\nabla_\psi r_\psi(\tau)] - E_{\tau \sim p(\tau|O_{1:T}, \psi)} [\nabla_\psi r_\psi(\tau)] \quad (13.9)$$

Se puede ver que en esta expresión se toma como positivas las recompensas obtenidas de π^* y se restan las recompensas obtenidas con π_ψ . El gradiente será 0 cuando las distribuciones sean iguales.

El primer término se estima usando muestras. El segundo término es el complicado de estimar y el resto del tema tratará de varias técnicas para obtenerlo.

13.3. Estimar la esperanza

Se recuerda que:

$$E_{\tau \sim p(\tau|O_{1:T}, \psi)} [\nabla_\psi r_\psi(\tau)] = E_{\tau \sim p(\tau|O_{1:T}, \psi)} \left[\nabla_\psi \sum_{t=1}^T r_\psi(s_t, a_t) \right] \quad (13.10)$$

Si se saca la suma fuera de la esperanza, se puede expresar con respecto al estado marginal (s_t, a_t) :

$$E_{\tau \sim p(\tau|O_{1:T}, \psi)} \left[\nabla_\psi \sum_{t=1}^T r_\psi(s_t, a_t) \right] = \sum_{t=1}^T E_{(s_t, a_t) \sim p(s_t, a_t|O_{1:T}, \psi)} [\nabla_\psi r_\psi(s_t, a_t)] \quad (13.11)$$

Se usa la regla de la cadena para:

$$(s_t, a_t) \sim p(s_t, a_t|O_{1:T}, \psi) = p(a_t|s_t, O_{1:T}, \psi)p(s_t|O_{1:T}, \psi) \quad (13.12)$$

Estos términos se han visto antes en el tema anterior. El primero es la política óptima blanda (*soft-optimal policy*), la cual con los mensajes hacia atrás se pueden calcular esas probabilidades. El segundo término también se vio en el tema pasado al final de la tercera pregunta.

$$p(a_t|s_t, O_{1:T}, \psi) = \frac{\beta(s_t, a_t)}{\beta(s_t)} \quad (13.13)$$

$$\propto \alpha(s_t)\beta(s_t) \quad (13.14)$$

Por lo que:

$$p(a_t|s_t, O_{1:T}, \psi)p(s_t|O_{1:T}, \psi) \propto \beta(s_t, a_t)\alpha(s_t) \quad (13.15)$$

Todo esto se está haciendo haciendo muchas asunciones. La más importante de ellas es que se pueden enumerar todos los estados y acciones, calcular los mensajes α y β , multiplicarlos entre ellos, y lo peor, normalizarlos, para que sea igual a 1 y no proporcional. Todo esto requiere que el espacio de acciones y estados sea lo suficientemente pequeño como para hacer esto. Pero esta asunción ni se aproxima a las complicaciones de calcular todas las trayectorias posibles, ya que es exponencial con respecto al tamaño del espacio de estados, mientras que esto es lineal en el espacio de estados.

Se define μ_t :

$$\mu_t(s_t, a_t) \propto \beta(s_t, a_t)\alpha(s_t) \quad (13.16)$$

Por lo que:

$$E_{\tau \sim p(\tau | O_{1:T}, \psi)} [\nabla_\psi r_\psi(\tau)] = \sum_{t=1}^T \int \int \mu_t(s_t, a_t) \nabla_\psi r_\psi(s_t, a_t) ds_t da_t \quad (13.17)$$

$$= \sum_{t=1}^T \vec{\mu}_t^T \nabla_\psi \vec{r}_\psi \quad (13.18)$$

Con esto, ya se tiene un algoritmo para hacer aprendizaje por refuerzo inverso:

Algoritmo 27: Máximo Entropy Inverse Reinforcement Learning

repetir

Dado ψ , calcular los mensajes hacia atrás $\beta(s_t, a_t)$

Dado ψ , calcular los mensajes hacia adelante $\alpha(s_t)$

Calcular $\mu_t(s_t, a_t) \propto \beta(s_t, a_t)\alpha(s_t)$

Evaluando: $\nabla_\psi L = \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\psi r_\psi(s_{i,t}, a_{i,t}) - \sum_{t=1}^T \int \int \mu_t(s_t, a_t) \nabla_\psi r_\psi(s_t, a_t) ds_t da_t$

$\psi \leftarrow \psi + \nu \nabla_\psi L$

hasta que mientras se siga entrenando;

Para espacios de estados pequeños y discretos esto funciona bastante bien.

Este método se llama *Maximum Entropy* ya que se puede demostrar que en el caso en el que la función de recompensa fuera lineal, este algoritmo da la solución que maximiza la entropía de la política resultante de modo que la esperanza de las características de esta política es igual a la esperanza de las características de la política del experto.

$$\max_\psi H(\pi^{r_\psi}) \text{ demodoo que } E_{\pi^{r_\psi}}[f] = E_{\pi^*}[f] \quad (13.19)$$

Esto tiene sentido ya que se está haciendo que a parte de correlacionar las características de la función de recompensa, no se está asumiendo nada más del comportamiento.

Este algoritmo requiere:

- Resolver la política óptima blanda en el bucle interno. Se calculan los mensajes. Básicamente se está resolviendo el problema de RL por cada paso del descenso por gradiente.
- Enumerar todas las tuplas acción-estado para obtener la frecuencia de visita y el gradiente.

Para poder aplicar esto en problemas prácticos, se tienen que relajar las asunciones:

- Espacios de acciones y estados grandes y continuos.
- Tratar con estados obtenidos solamente de muestras.
- Tratar con dinámicas desconocidas.

13.3.1. Con dinámica desconocida y espacio de acciones/estados grandes

Se asume que no se conoce la dinámica pero que se pueden sacar muestras como con RL.

Se recuerda que el gradiente viene dado por 13.9, donde la primera parte se calcula fácilmente pero la segunda no.

Se podría pensar en un algoritmo bueno pero impráctico que sería aprender $p(a_t|s_t, O_{1:T}, \psi)$ usando cualquier algoritmo *Maximum Entropy RL algorithm* (*soft Q-Learning*, el algoritmo Policy Gradient con el término de la entropía, ...). El problema es que esos se tienen que ejecutar hasta la convergencia para cada paso tomado en ψ , aunque no requieren conocer las dinámicas ni enumerar los estados.

$$\nabla_\psi L \approx \frac{1}{N} \sum_{i=1}^N \nabla_\psi r_\psi(\tau_i) - \frac{1}{M} \sum_{j=1}^M \nabla_\psi r_\psi(\tau_j) \quad (13.20)$$

Se pretenden realizar actualizaciones de ψ que sean más eficientes con respecto a las muestras. Para ello, en vez de aprender $p(a_t|s_t, O_{1:T}, \psi)$ se va simplemente a mejorar ligeramente usando un algoritmo de máxima entropía. Una forma de mejorarla ligeramente por ejemplo es hacer un paso de policy gradient (esto funciona). Esto no nos va a dar las muestras de $p(\tau|O_{1:T}, \psi)$, pero nos dará las muestras de otra distribución en la que a medida que vayamos tomando más pasos en el gradiente las muestras se irán pareciendo más y más.

Para convertir muestras de una distribución a otra se puede usar Importance Sampling:

$$\nabla_\psi L \approx \frac{1}{N} \sum_{i=1}^N \nabla_\psi r_\psi(\tau_i) - \frac{1}{\sum_j w_j} \sum_{j=1}^M w_j \nabla_\psi r_\psi(\tau_j) \quad (13.21)$$

$$w_j = \frac{p(\tau) \exp(r_\psi(\tau_j))}{\pi(\tau_j)} \quad (13.22)$$

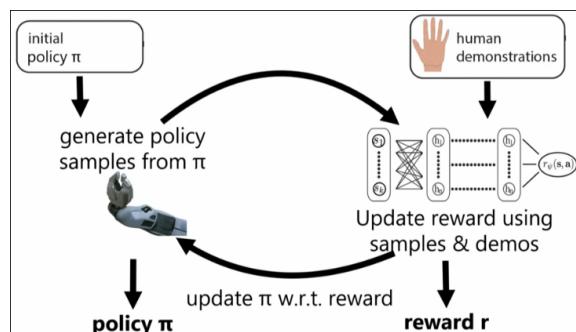
Expandiendo la expresión para los pesos:

$$w_j = \frac{p(\tau) \exp(r_\psi(\tau_j))}{\pi(\tau_j)} = \frac{p(s_1) \prod_t p(s_{t+1}|s_t, a_t) \exp(r_\psi(s_t, a_t))}{p(s_1) \prod_t p(s_{t+1}|s_t, a_t) \pi(a_t|s_t)} = \frac{\exp(\sum_t r_\psi(s_t, a_t))}{\prod_t \pi(a_t|s_t)} \quad (13.23)$$

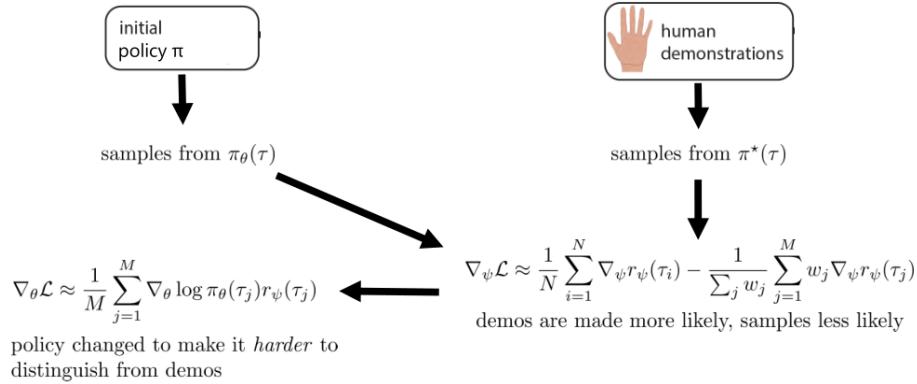
El denominador resultante es difícil de trabajar, pero es un paso en la buena dirección.

En general, Importance Sampling no es una buena idea usarlo si se trata de dos distribuciones muy distantes. Aquí las distribuciones pueden ser distantes al principio pero por cada actualización de r_ψ las trayectorias se irán pareciendo más y más, por lo que no es descabellado usarlo.

El primer algoritmo que usó esta estructura fue *Guided Cost Learning Algorithm*, Finn et al. ICML 2016.



Este algoritmo puede ser interpretado como un ascenso por gradiente con respecto a ψ . Pero también se puede interpretar que r_ψ se actualiza, aumenta la recompensa del experto y disminuye la del agente, y cuando se actualiza la política intenta que su recompensa sea máxima.



El objetivo final es hacer que la política π_θ sea indistinguible de la política del experto π^* . El único equilibrio es resolver el problema del aprendizaje por refuerzo inverso.

Esto se parece mucho a una GAN. La política tiene el rol del generador, y la función de recompensa el discriminador.

En el caso de aprendizaje por refuerzo inverso, el mejor discriminador es:

$$D^*(x) = \frac{p^*(x)}{p_\theta(x) + p^*(x)} \quad (13.24)$$

como la política óptima se acerca a $\pi_\theta(\tau) \propto p(\tau) \exp(r_\psi(\tau))$, se elige la siguiente parametrización para el discriminador:

$$D_\psi(\tau) = \frac{p(\tau) \frac{1}{Z} \exp(r(\tau))}{p_\theta(\tau) + p(\tau) \frac{1}{Z} \exp(r(\tau))} = \frac{p(\tau) \frac{1}{Z} \exp(r(\tau))}{p(\tau) \prod_t \pi_\theta(a_t | s_t) + p(\tau) \frac{1}{Z} \exp(r(\tau))} = \frac{\frac{1}{Z} \exp(r(\tau))}{\prod_t \pi_\theta(a_t | s_t) + \frac{1}{Z} \exp(r(\tau))} \quad (13.25)$$

Esta expresión no se puede escribir para GAN en otros ámbitos porque aquí conocemos π_θ .

Ahora se optimiza el discriminador con respecto a ψ . Si se escribe el objetivo del discriminador de antes y le metemos esta definición de discriminador se obtiene:

$$\psi \leftarrow \arg \max_\psi E_{\tau \sim p^*} [\log D_\psi(\tau)] + E_{\tau \sim \pi_\theta} [\log (1 - D_\psi(\tau))] \quad (13.26)$$

Se obtiene una expresión que coincide con el gradiente de RL inverso.

Sorprendentemente, también se puede optimizar Z con respecto a ψ en el objetivo y se obtiene el valor correcto de Z .

La explicación de esto está en la publicación: *Finn; Christiano et al. A Connection Between Generative Adversarial Networks, Inverse Reinforcement Learning, and Energy-Based Models.*

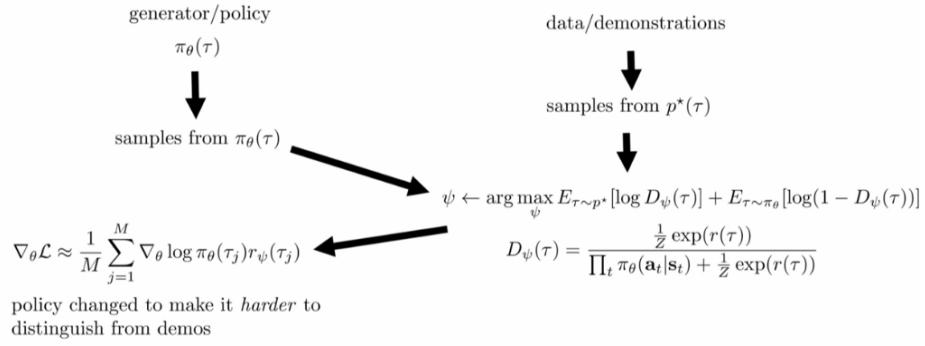
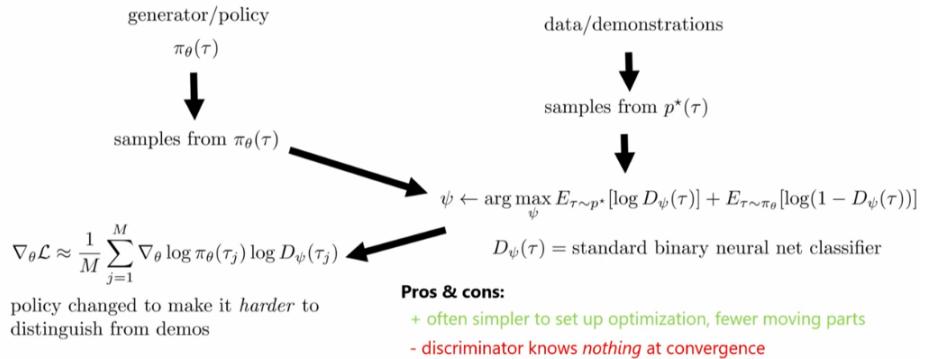


Figura 13.1: Algoritmo de IRL

Si solo nos preocupa la política obtenida y no la explicación de las acciones de los agentes, se puede usar un método más parecido a una GAN literalmente:



En resumen, en vez de minimizar y maximizar, lo que se hace es tener un discriminador binario. $D(\tau)$ es la probabilidad de que τ sea una demostración y no una trayectoria generada por el agente. Se usa $D(\tau)$ como recompensa:

Tema 14

Aprendizaje transferible y multi-tarea

Clase 16: Transfer and Multi-Task Learning

2020-07-08

14.1. ¿Cuál es el problema?

Para tareas 'simples' como por ejemplo Breakout, los algoritmos vistos hasta ahora funcionan razonablemente bien. Pero para entornos complejos como por ejemplo *Montezuma's Revenge* estos algoritmos fallan catastróficamente. Esto es porque este tipo de juegos requiere un entendimiento de qué significan los *sprites* que salen por pantalla. Por ejemplo:

- La llave: abre puertas
- Escaleras: se pueden subir
- Calavera: no se sabe lo que hace exactamente, pero se sobreentiende que nada bueno.

Un conocimiento previo de la estructura del problema puede ayudarnos a resolver tareas complejas rápidamente.

14.1.1. ¿Puede RL usar el mismo conocimiento previo que nosotros?

La idea está en que una vez se hayan resuelto otras tareas, es posible que se haya obtenido un conocimiento útil para resolver otra tarea nueva.

Hay varias formas de guardar el conocimiento:

- Función Q: nos dice que estados y acciones son buenos
- Política: nos dice que acciones son potencialmente útiles. Algunas acciones puede ser que nunca sean útiles.
- Modelos: nos dicen las leyes de la física que gobiernan el mundo.
- Características/estados ocultos: proveen de una buena representación. Por ejemplo, una CNN que detecte elementos de una escena y los codifique. No subestimar.

14.2. Terminología del aprendizaje transferible

- *Transfer Learning*: usar experiencia de una serie de tareas para un aprendizaje más rápido y mejor en otra tarea nueva. En aprendizaje por refuerzo, la tarea es el MDP.
- *Shot*: número de intentos en el entorno objetivo.

- *0-shot*: simplemente se corre la política entrenada en el dominio de la fuente. La política es muy generalizable. Normalmente se requieren ciertas asunciones.
- *1-shot*: el algoritmo sólo necesita aprender a partir de un ejemplo.
- *few-shot*: necesita realizar unos pocos episodios para aprender.

14.3. ¿Cómo se pueden plantear los problemas de aprendizaje por transferencia?

Hay varias formas de plantearlo, cada una de ellas forma algoritmos diferentes:

- *Forward transfer*: entrenar en una tarea, transferir a otra tarea.
- *Multi-task transfer*: entrenar en muchas tareas, transferir a otra tarea nueva.
- *Multi-task meta-learning*: aprender a aprender a partir de muchas tareas (próximo tema).

14.3.1. Forward Transfer

Se entrena en una tarea y se transfiere a otra tarea. Se puede hacer así burdamente pero lo mejor es entrenar en una tarea y refinar en otra tarea.

Si se tiene el poder de elegir las tareas fuente, se pueden aleatorizar para aumentar la generalización.

14.3.2. Finetuning

Finetuning viene del mundo del aprendizaje supervisado, y su aplicación a RL requiere de ciertos cambios.

Uno de los problemas más grandes es que al entrenar una política en entornos totalmente observables, la política óptima siempre será determinista. Los algoritmos que se han visto hasta ahora van reduciendo la exploración a medida que van convergiendo para aproximarse a esta política determinista. Por lo que al no haber exploración, pasar esa política a un nuevo dominio es problemático.

Por lo que normalmente se entranan políticas algo más estocásticas en los entornos fuente.



En el caso que se quiera transferir de la tarea de la izquierda a la de la derecha, es mejor que al comienzo se entrene una política que llegue al destino por los dos caminos posibles, aunque el camino largo de menos recompensa. Para esto, se puede utilizar lo que se vio en el replanteamiento de RL como un problema de inferencia, donde se maximiza tanto la recompensa como la entropía. Esto hace que en la nueva tarea el agente pueda llegar al destino a partir de la política original.

En la práctica, una vez que se tenga entrenada la política en el entorno objetivo es preferible dejar de maximizar la entropía. Para esto se pueden hacer varias cosas:

- Entrenar la política con un coeficiente que multiplique a la entropía y vaya disminuyendo con el paso del tiempo.
- Si se tiene una red neuronal que produce una mezcla de gaussianas en su salida, se puede reducir su varianza a 0. Suele funcionar bien pero no está garantizado que funcione, ya que el modelo no fue entrenado con varianza nula.

Publicaciones importantes sobre *finetuning*:

- Finetuning via MaxEnt RL: Haarnoja*, Tang*, et al. (2017). Reinforcement Learning with Deep
- Energy-Based Policies.
- Finetuning from transferred visual features (via VAE): Higgins et al. DARLA: improving zero-shot
- transfer in reinforcement learning. 2017.
- Andreas et al. Modular multitask reinforcement learning with policy sketches. 2017.
- Florensa et al. Stochastic neural networks for hierarchical reinforcement learning. 2017.

Manipular el dominio fuente

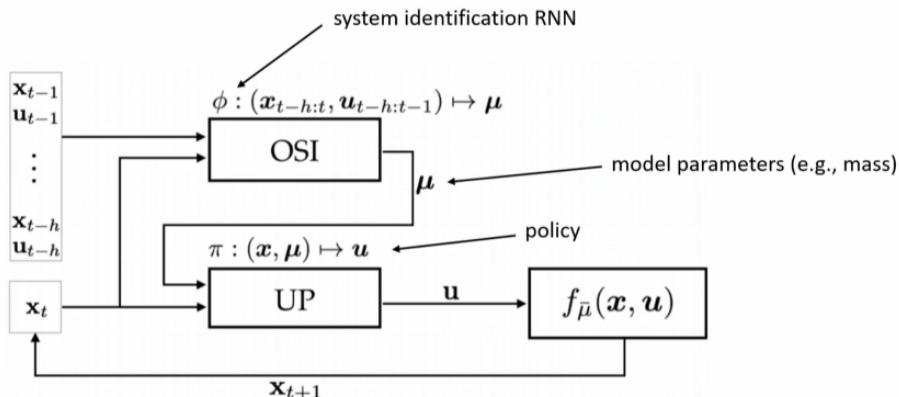
Pueden haber casos en los que los dominios fuente se puedan modificar. Esto ocurre cuando se diseña una simulación para luego ser aplicada en el mundo real, por ejemplo.

Lo que interesa es tener un entorno fuente en el que el agente aprenda a partir de muchas muestras muy diversas, para luego aplicarla a otra tarea más específicas con la esperanza de que el nuevo entorno más específico se parezca en parte al entorno original.

Por ejemplo, en la publicación *EPOpt: Learning robust neural network policies* se entrena un agente sobre un robot virtual con masas variantes en sus enlaces. El control robusto obtenido tiene un precio: para poder funcionar con cualquier tipo de masas, la política no será la óptima en ninguna de ellas. Esto se acentúa con políticas lineales o poco expresivas, pero con las redes neuronales este problema se disminuye considerablemente (incluso haciendo que la diferencia sea casi nula en algunos casos).

Se pueden recoger los parámetros de un robot real para pasarlo a simulación haciendo varias pruebas con el hardware real y luego entrenar sobre los parámetros físicos obtenidos. Pero puede resultar ser mejor idea crear un controlador robusto que se adapte a los cambios de esos parámetros físicos, ya que para los parámetros reales tiene la misma eficiencia que el controlador específico además de tener un mejor comportamiento con parámetros distintos.

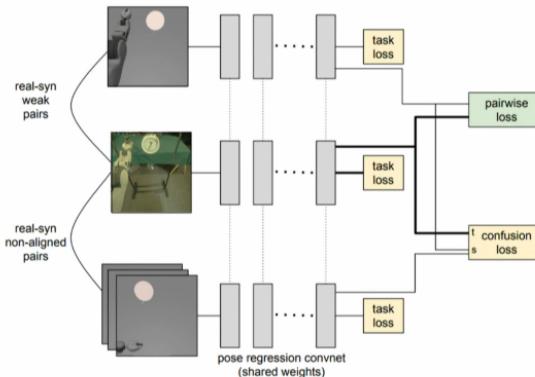
En otra publicación (*Preparing for the unknown: Learning a Universal Policy with Online System Identification*), se tiene un diseño más complejo en el que se tiene a una red recurrente que intenta predecir los parámetros físicos del simulador, para poder meterlos como entrada a la política, y de esta forma que sepa qué acciones tomar.



Estos métodos suponen que no se sabe nada del dominio del objetivo (0-shot), pero en problemas reales si que se conocen cosas de la tarea que se quiere realizar.

Una de las cosas que se puede hacer es *Adversarial Domain Adaptation*, también llamado *Domain Confusion*. En el cual se tienen por ejemplo imágenes de simulación con pocas características e

imágenes reales que se corresponden exactamente con las de la simulación. Se pretende obtener un modelo que saque las mismas características para dos imágenes correspondientes: una de la simulación y otra de la realidad.



Hay dos formas de conseguir esto:

- Si las dos figuras no son solo análogas sino que también se corresponden con el mismo estado exactamente, entonces se pueden regularizar las características para que tomen valores similares. Normalmente no se tiene ese conocimiento.
- Si no se corresponden pero vienen de la misma distribución (por ejemplo una política aleatoria sobre la simulación y sobre el mundo) se puede utilizar una función de pérdida *unpaired distributional alignment* sobre las características. Un ejemplo de esta función de pérdida viene prestada de las GANs. Se usa un discriminador sobre las características extraídas de las imágenes. Se pretende que la red neuronal genere características que hagan que el discriminador no sea capaz de diferenciar su procedencia (real o simulación). Esto se conoce como *Domain Adversarial Neural Networks*, *Domain Confusion* o *Adversarial Domain Adaptation*.

Resumen del *forward training*

- Preentrenamiento y finetuning:
 - El finetuning estándar aplicado a RL es difícil.
 - La formulación de la entropía máxima puede ayudar.
- Se pude modificar el entorno fuente:
 - La aleatorización puede ayudar mucho a crear políticas fuente más diversas.
- En el caso que se tengan pocos datos del dominio fuente:
 - *Domain adaptation*: hacer que la red neuronal no sea capaz de distinguir entre las observaciones de ambos dominios.
 - O modificar las observaciones en el dominio fuente para que se vean como en el dominio objetivo (por ejemplo con una GAN).

14.3.3. Multi-task transfer

Hasta ahora se usaba un dominio bastante general para poder preentrenar un modelo que posteriormente lo hiciera bien en una tarea específica. Pero si se quiere entrenar un modelo para que haga varias tareas, es posible que no haya falta diseñar un entorno fuente tan diverso.

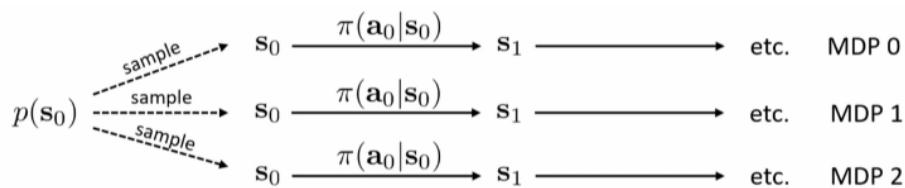
Esto es más cercano a lo que las personas hacemos, que es utilizar lo aprendido en tareas anteriores para realizar nuevas. Pero es sustancialmente más difícil ya que no todas las tareas pasadas

contribuyen a realizar cualquier tarea nueva, incluso puede ser que dificulte su aprendizaje.

RL basado en modelo

Para este caso la transferencia es fácil de entender. Se busca encontrar que tienen en común todas las tareas:

- Idea 1: las leyes de la física.
 - Versión simple: entrenar un modelo en tareas pasadas y usarlo para resolver tareas nuevas.
 - Versión más compleja: se adapta el modelo a la nueva tarea. Es más fácil de entrenar que el *finetuning* si las físicas son las mismas pero la tarea cambia mucho.
- Idea 2: Entrenar todas las políticas de forma separada (MDP diferentes) y después unirlas en una única política. Para combinarlos se inicializa las simulaciones con un estado inicial y un MDP.



Pero esto es una tarea difícil ya que conseguir que un optimizador consiga una política que se comporte bien en todos los MDP es una tarea complicada. Por lo que se pueden combinar las políticas mediante *Policy Distillation*.

¿Cómo sabe el modelo qué hacer?

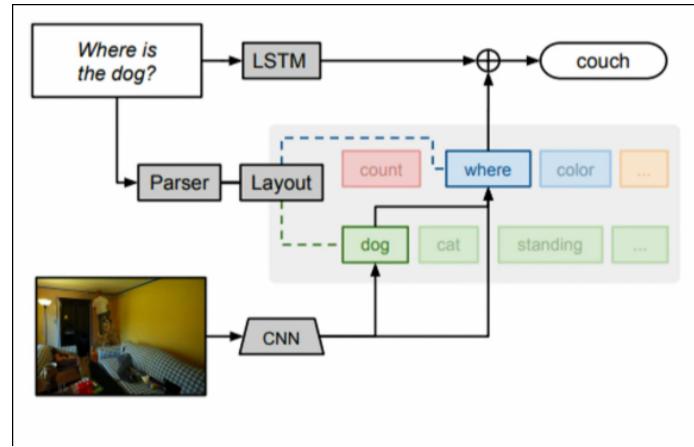
En el caso de juegos de Atari, es fácil ya que se puede ver en qué juego está simplemente mirando el estado. Pero hay otros casos como por ejemplo un robot que realice varias tareas en las que no está tan claro qué tarea debe realizar.

Se pueden enumerar las tareas a realizar con un número o representarlas con un vector *one-hot* y condicionar la política en ese número o vector. (*Contextual Policy*: $\pi_\theta(a|s, \omega)$). Esto es muy utilizado en robótica y en animación.

Arquitecturas para multi-task learning

Hasta ahora se ha creado una red neuronal para todas las tareas. Esto no tiene sentido porque por ejemplo se puede querer una política que controle un coche a partir de la información de un LiDAR y otro que lo haga a partir de la información de una cámara, o diez robots intentando hacer tareas distintas. Por lo que interesa crear arquitecturas con componentes reusables.

En la publicación *Neural Module Networks*, Andreas et al. se hace justo esto. Aunque trabaja sobre una tarea de aprendizaje supervisado.



Su aplicación a RL se explica en *Learning Modular Neural Network Policies*, Devin, Gupta, et al. En este paper se divide la política en dos partes: la específica del robot y la específica de la tarea. Por lo que se pueden coger todas las tareas y los robots y distribuirlos en una matriz. La idea está en que si se entrena las políticas sobre algunas tareas compuestas de las combinaciones de la tabla, el robot lo hará bien en combinaciones no vistas o necesitará muy poco entrenamiento.

En la práctica esto puede ir mal por dos motivos:

- Si se tienen pocos módulos puede producir sobreentrenamiento.
- Si la capacidad de la capa intermedia es lo suficientemente alta se puede sobreentrenar aunque se tengan muchas tareas. Por lo que habría que aplicar regularización.

Tema 15

RL Distribuido

Clase 17: Distributed RL

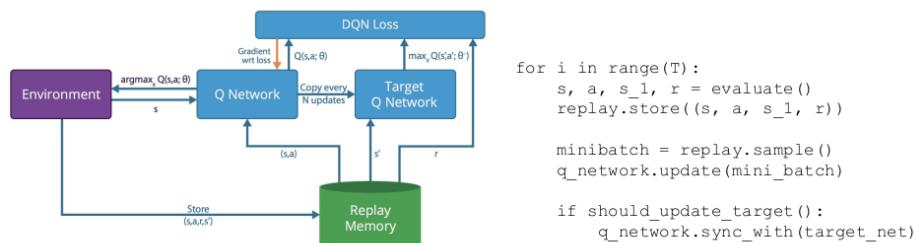
2020-07-13

15.1. Patrones computacionales comunes en RL

Se han visto algoritmos que actualizan la política a partir de un batch de observaciones. Esto se hace en contraposición de los algoritmos online ya que de esta forma se está aprovechando todo el poder computacional de las CPU de forma paralela.

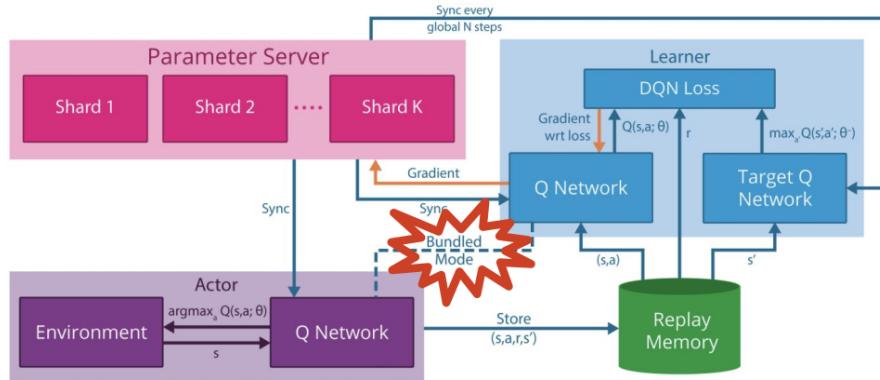


- DQN (2013-2015): no es distribuido ni paralelizado pero introduce ideas que serán útiles para ello.



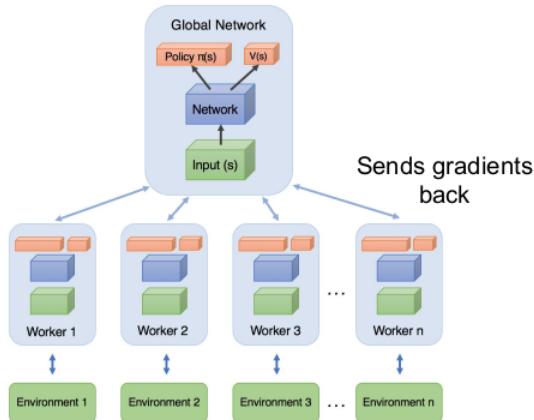
Se tiene un agente que evalúa un entorno y los datos recogidos se guardan en un buffer. De este buffer se recogen batches aleatorios que sirven para actualizar las redes neuronales.

- GORILA (2015):

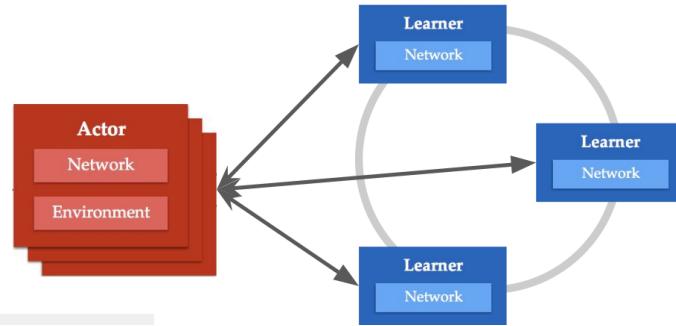


Con esta arquitectura se mejora el resultado de casi todos los entornos de Atari, lo que lleva a pensar que escalar RL escogiendo las abstracciones adecuadas puede suponer un gran paso hacia adelante.

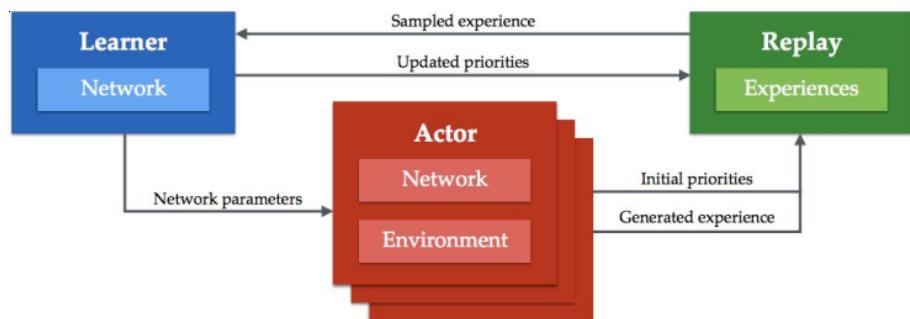
- A3C (2016): es una gran mejora sobre GORILA. Una de sus características más sobresalientes es que se puede entrenar en una sola máquina. Esto permite evitar los cuellos de botella generados en las comunicaciones cuando se tienen varios componentes. También permite no usar un replay buffer.



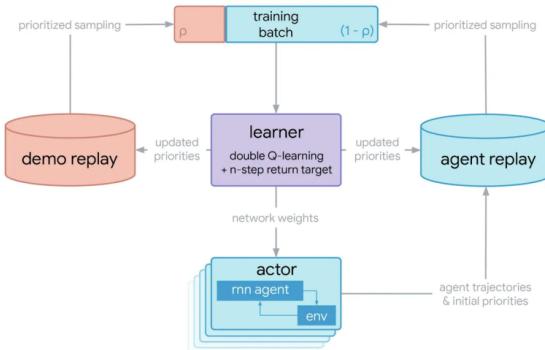
- Importance Weighted Actor-Learner Architectures (IMPALA, 2018): se tienen varias copias de Learners y Actors. Los Actors tienen su propia copia de la red neuronal y actúan continuamente en sus entornos, generando una inmensa cantidad de datos. Por otro lado los Learners se encargan de distribuir la computación de la optimización. Como las redes que actúan no son las mismas que las que se están optimizando se usa un algoritmo de *Importance Sampling* llamado *V-tracing*. En la publicación muestran que este algoritmo consigue estabilizar el proceso de aprendizaje.



- Ape-X/R2D2 (2018): da un paso atrás hacia GORILA y reaparece el replay buffer. Se tienen Actores corriendo asíncronamente y guardando las observaciones en el replay buffer. Otro proceso coge muestras del buffer y las usa para entrenar a la política. El buffer y el Learner son independientemente escalables. La novedad de esta arquitectura está en el buffer, que se ordena usando *Distributed Prioritization*. Comparado con A3C y DQN escala extremadamente bien.



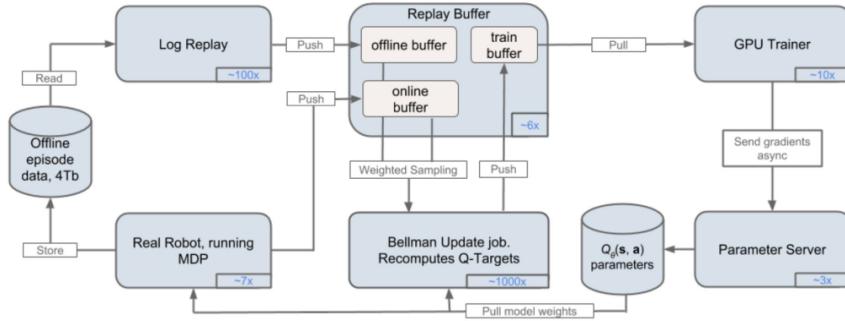
- R2D3 (Ape-X con demostraciones, 2019): una arquitectura similar que permite el uso de datos de entrenamiento generados por humanos o un profesor. Se tienen dos buffers, el *demo replay* y el *agent replay*, de los cuales se muestrea con probabilidad ρ y $1 - \rho$ respectivamente.



15.1.1. Otras arquitecturas distribuidas interesantes

QT-Opt

Es una arquitectura que se conecta con el mundo real mediante robots. Consigue hacer un bucle cerrado para entrenarlos en la tarea de la manipulación de objetos.



Abajo a la izquierda se tienen a los robots generando experiencia. Estos datos se meten en un almacenamiento y posteriormente en un buffer. El replay buffer recibe experiencias inmediatas y guardadas en el buffer.

Estrategias de evolución

Consisten en algoritmos de optimización altamente paralelizables. Cada agente tiene su copia del entorno y de la red neuronal y cada red es ligeramente perturbada por un ruido y produce una trayectoria.

Algoritmo 28: Estrategias de evolución paralelizables

Entrada: Learning rate α , desviación estándar del ruido σ , parámetros iniciales θ_0
 Inicializar n trabajadores con semillas aleatorias conocidas y parámetros iniciales θ_0
para $t = 1, 2, 3, \dots$ **hacer**

```

para cada trabajador  $i = 1, \dots, n$  hacer
  | Sacar una muestra  $\epsilon \sim N(0, I)$ 
  | Calcular las recompensas  $F_i = F(\theta_t + \sigma\epsilon_i)$ 
fin
Enviar todos los escalares  $F_i$  de cada trabajador a los demás trabajadores.
para cada trabajador  $i = 1, \dots, n$  hacer
  | Reconstruir todas las perturbaciones  $\epsilon_j$  para todo  $j = 1, \dots, n$  usando las semillas
    aleatorias conocidas.
  |  $\theta_{t+1} \leftarrow \theta_t + \frac{1}{n\sigma} \sum_{j=1}^n F_j \epsilon_j$ 
fin
fin

```

Más allá de RL: Entrenamiento basado en población

Es un método de búsqueda de hiperparámetros creado por DeepMind que comienza siendo como una búsqueda aleatoria. Pero posteriormente en cada iteración los modelos que no se estén comportando tan bien como el resto son modificados para mejorarlo.

Esto puede ayudar en mejorar los modelos existentes en un 5 % aproximadamente.

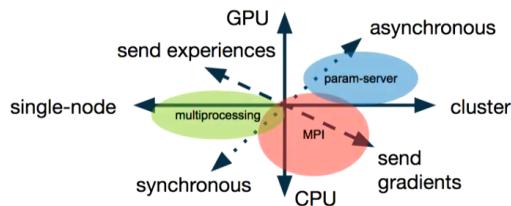
15.2. RLlib: Abstractions for Distributed Reinforcement Learning (ICML 2018)

Últimamente el hardware se está haciendo mucho más potente y barato, pero no todo el mundo tiene la infraestructura de software para hacer uso de todos estos beneficios. Para la creación de modelos existen múltiples librerías excelentes que ofrecen abstracciones como Tensorflow o PyTorch. Pero en el mundo de RL no existía una abstracción para la creación de algoritmos escalables.

Si se mira a los algoritmos implementados hasta ahora, la mayoría de ellos usa MPI, Redis, Multiprocessing, Distributed TF, En github existen aproximadamente 16000 implementaciones de algoritmos de RL y no comparten la misma base de código.

Los desafíos para la creación de una base de código reusable son:

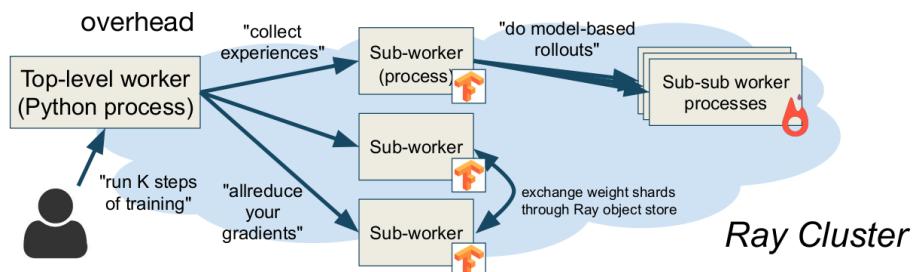
- Un amplio rango de estrategias de ejecución físicas para un algoritmo.



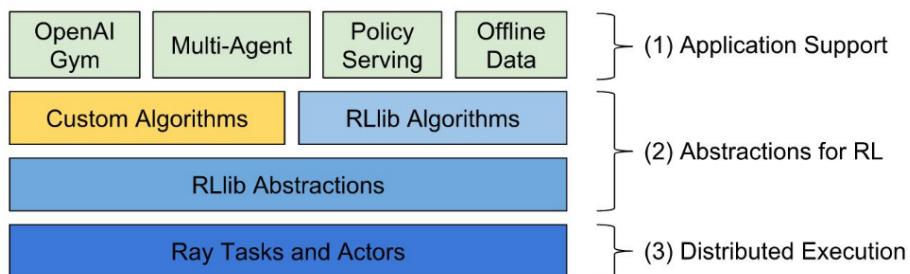
- Las implementaciones están fuertemente acopladas a la librería usada para crear los modelos (PyTorch, Tensorflow, ...).
- Gran variedad de algoritmos con diferentes estructuras.

Con Ray se puede crear una instancia de clase en el cluster (*statefull workers*) y después ir asignando pequeñas tareas a esos trabajadores. El desafío está en que mantenga un alto rendimiento: 1e6+ tareas/s, 200us por tarea.

Se pueden crear arquitecturas jerárquicas, en las que un trabajador tiene subtrabajadores y esos tienen subsubtrabajadores.



En resumen, RLLib está por encima de una simple colección de algoritmos. Las abstracciones permiten implementar y escalar fácilmente nuevos algoritmos.



Tema 16

Exploración

Clase 18: Exploration (Part 1)

2020-07-14

16.1. ¿Qué es la exploración? ¿Por qué es un problema?

Desde el punto de vista del algoritmo no se conocen las reglas del entorno en el que se tiene que desenvolver. Solo recibe las señales de recompensa y a partir de eso tiene que averiguar que es lo que está bien y que es lo que está mal, por lo que va descubriendo las reglas a partir de prueba y error. Descubrir las reglas es de notoria dificultad si las tareas se extienden en el tiempo como por ejemplo en Montezuma's Revenge.

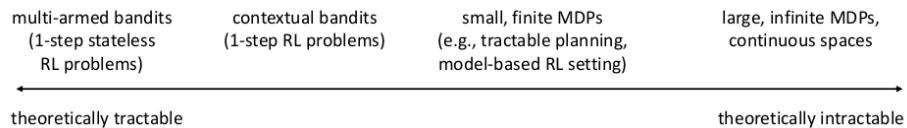
Dos posibles definiciones para el problema de la exploración son:

- ¿Cómo puede un agente descubrir estrategias que den una alta recompensa que requieran de una secuencia compleja de acciones extendida en el tiempo que, individualmente, no devuelven una recompensa?
- ¿Cómo puede un agente decidir si probar nuevos comportamientos (descubrir nuevos con mejores recompensas) o continuar haciendo la mejor acción conocida hasta el momento?

En realidad es el mismo problema, en el que se tiene que balancear entre:

- Explotación: hacer lo que se 'sabe' que va a dar la mayor recompensa.
- Exploración: hacer cosas que no se han probado antes con la intención de encontrar recompensas mayores.

La exploración es un problema complicado.



Se puede formalizar el proceso de exploración como la identificación de un POMDP. El aprendizaje de la política es trivial incluso en POMDP.

16.2. *Multi-armed bandits*

En un problema de múltiples bandidos (máquinas tragaperras), el espacio de acciones es $A = \{pull_1, pull_2, \dots, pull_n\}$. Se quiere encontrar qué acciones llevan a la máxima recompensa, ya que ciertas máquinas dan más dinero que otras. Se puede asumir que $r(a_n) \sim p(r|a_n)$.

Un bandido se define como $r(a_i) \sim p_{\theta_i}(r_i)$. Por ejemplo $p(r_i = 1) = \theta_i$ y $p(r_i = 0) = 1 - \theta_i$. Se asume que puede haber un prior, pero de otra forma son desconocidos.

Esto define un POMDP con $s = [\theta_1, \dots, \theta_n]$. Los estados son las probabilidades que se piensa que se tienen en todos los bandidos.

Se necesita una medida de como de bien se está haciendo la exploración, esta medida se le llama arrepentimiento (*regret*) y mide la diferencia con respecto a la política óptima en el paso T .

$$Reg(T) = TE[r(a^*)] - \sum_{t=1}^T r(a_t) \quad (16.1)$$

No es una medida que se pueda usar en un problema que se tenga que resolver, pero sirve para comparar entre varios algoritmos.

Hay varias estrategias simples para 'vencer' el problema de los bandidos. Típicamente lo que se quiere es demostrar una cota del arrepentimiento en la esperanza. Hay varios algoritmos que son igual de óptimos hasta un factor constante, y varían muy poco entre ellos dependiendo del problema.

La exploración basada en métodos más complicados como DRL carece de las garantías de convergencia de los métodos sencillos.

16.3. Exploración basada en optimismo

Es una de las más sencillas pero de las más efectivas. Se tiene en cuenta la media de las recompensas $\hat{\mu}_a$ para cada acción a . De modo que la explotación queda en elegir $a = \arg \max \hat{\mu}_a$. Esto no se quiere hacer siempre ya que no se explorará.

En vez de coger el máximo siempre, se va a bonificar a las acciones que se hayan cogido pocas veces:

$$a = \arg \max \hat{\mu}_a + C\sigma_a \quad (16.2)$$

Hay muchas formas de elegir σ . Una regla UCB bastante buena es (Auer et al.):

$$a = \arg \max \hat{\mu}_a + \sqrt{\frac{2 \log T}{N(a)}} \quad (16.3)$$

Con esto $Reg(T)$ es $O(\log T)$, que está demostrado que es lo mejor que se puede obtener.

16.4. Posterior matching exploration

Se asume que $r(a_i) \sim p_{\theta_i}(r_i)$. Se define un POMDP como $s = [\theta_1, \dots, \theta_n]$. El estado de creencias es $\hat{p}(\theta_1, \dots, \theta_n)$.

La idea es simple, se supone que nuestras creencias \hat{p} son correctas y se actúa con respecto a ellas. Naturalmente se tiene una fuerte exploración. Una vez se ha tomado la acción, se mira el resultado y se actualizan nuestras creencias de nuevo.

Es más difícil de analizar teóricamente pero empíricamente funciona muy bien.

16.5. Information-theoretic exploration

Está basado en la ganancia de información, por lo que necesita un poco más de explicación.

Diseño experimental Bayesiano: se quiere determinar una variable latente z (por ejemplo z puede ser la acción óptima o su valor). $H(\hat{p}(z))$ es la entropía de nuestra estimación de z . $H(\hat{p}(z)|y)$ es la

entropía de nuestra estimación de z después de observar z (por ejemplo y puede ser $r(a)$). Se define la ganancia de información como:

$$IG(z, y) = E_y[H(\hat{p}(z)) - H(\hat{p}(z)|y)] \quad (16.4)$$

Normalmente depende de las acciones por lo que se tiene $IG(z, y|a)$. Por lo que para estimar mejor z se tiene que coger a que de la mayor ganancia de información.

16.6. Métodos de exploración en DRL

Exploración optimista

La función de bonus de exploración pueden ser muchas, siempre que decrezcan con $N(a)$. Se puede usar esta idea con MDP, ahora se usa $N(s, a)$ o $N(s)$ para añadir el bonus a la exploración. Por lo que ahora $r^+(s, a) = r(s, a) + B(N(s))$. Ahora se usa r^+ en vez de r con cualquier algoritmo sin modelo.

Para casos en que los estados sean incontables (imágenes por ejemplo), es imposible e impráctico tener una cuenta de todas las veces que han aparecido los estados. Por lo que se puede entrenar un aproximador de funciones para que mantenga la cuenta entre estados similares.

Se va a entrenar un modelo de densidad $p_\theta(s)$ o $p_\theta(s, a)$ que nos de un número que sea pequeño para estados similares que no ha visto nunca y grande para estados similares que ya ha visitado.

Si se tiene un MDP pequeño tabular, la probabilidad está relacionada con la cuenta de los estados:

$$P(s) = \frac{N(s)}{n} \quad (16.5)$$

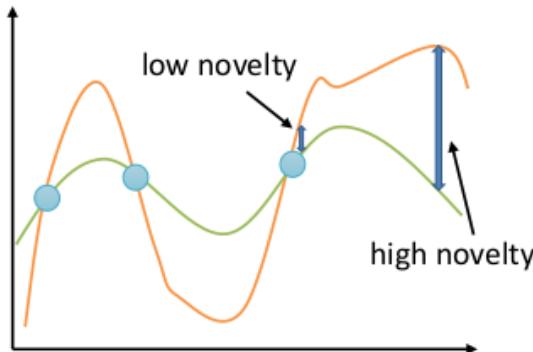
n es el número de estados vistos. Después de que se vea s , se actualiza como:

$$P'(s) = \frac{N(s) + 1}{n + 1} \quad (16.6)$$

Para hacer los pseudo-contadores se ajusta $p_\theta(s)$ a todos los estados D vistos hasta ahora. Se toma un pas i y se observa s_i . Se ajusta el nuevo modelo $p_{\theta'}(s)$ a $D \cup s_i$. Finalmente se usa $p_\theta(s_i)$ y $p_{\theta'}(s_i)$ para estimar $\hat{N}(s)$ y se pone $r_i^+ = r_i + B(\hat{N}(s))$.

Para conseguir $\hat{N}(s)$ se resuelve el sistema de ecuaciones formado por las dos ecuaciones anteriores.

Otra forma de estimar las cuentas es mediante los errores. Se tiene un modelo capaz de sobreentrenarse sobre los estados. Se entrena para que pase por los datos visitados, por lo que los datos que no estén visitados no pasarán por las predicciones del modelo y se puede medir el error.



Clase 19: Exploration (Part 2)

2020-07-15

Muestreo de Thompson

Se aprende una distribución sobre las funciones Q o las políticas y se muestrea y actúa de acuerdo con las muestras extraídas.

$$\theta_1, \dots, \theta_n \sim \hat{p}(\theta_1, \dots, \theta_n) \quad (16.7)$$

$$a = \arg \max_a E_{\theta_a}[r(a)] \quad (16.8)$$

En el problema de los *bandits* se mantenía una distribución del modelo (se muestrea lo que se piensa que va a dar el entorno). En un problema de DRL una opción es que representen la distribución de las funciones Q.

Lo teóricamente correcto si se quiere hacer un muestreo de Thompson en un problema de DRL es tener una distribución sobre el MDP. En la práctica esto es complicado por lo que se suele hacer la distribución sobre las funciones Q, que son un análogo del MDP.

Lo siguiente, repetido, describe un proceso de RL

1. Se muestrea la función Q a partir de $p(Q)$
2. Se actúa según Q por un episodio.
3. Actualizar $p(Q)$

Esto se puede hacer con Q-Learning porque es off-policy, si se quieren usar PG la cosa se complica más.

Ganancia de información

Se razona sobre la ganancia de información desde la visita a nuevos estados.

La intuición dice de usar IG con respecto a la recompensa, pero en la práctica se suelen tener entornos con una recompensa muy dispersa por lo que esto no funciona.

Una opción que funciona mejor es medir IG en función de la densidad de $p(s)$. Intuitivamente esto hace que se elijan acciones que cambien nuestras creencias sobre $p(s)$.

Otra opción es considerar IG sobre la dinámica $p(s'|s, a)$.

Generalmente esto no es tratable en la práctica, sea lo que sea lo que se esté estimando, por lo que se trabajará con aproximaciones. Se pueden elegir varias aproximaciones:

- Ganancia de predicción (*prediction gain*): se define como $\log p_{\theta}(s) - \log p_{\theta'}(s)$. Es muy parecido a IG matemáticamente. θ' son los parámetros actualizados después de visitar s . Intuitivamente, si la densidad cambia mucho es que el estado era nuevo.
- Inferencia variacional (Houthooft et al. "VIME"): IG se puede escribir de forma equivalente como $D_{KL}(p(z|y) || p(z))$. Se puede aprender las recompensas o las funciones Q pero de ejemplo se va a mostrar como aprender las transiciones $p_{\theta}(s_{t+1}|s_t, a_t) : z = \theta$. Lo que se observa es $y = (s_t, a_t, s_{t+1})$. Como se sabe que IG es equivalente a la divergencia KL, se sustituye por la definición de z y θ :

$$D_{KL}(p(\theta|h, s_t, a_t, s_{t+1}) || p(\theta|h)) \quad (16.9)$$

La intuición está en que una transición es más informative si hace que el conocimiento sobre θ cambie. Como se dijo antes, esto no es tratable en la práctica por lo que se va a hacer una aproximación $q(\theta|\phi) \approx p(\theta|h)$, como por ejemplo inferencia variacional. Por lo que ahora dada una transición nueva (s, a, s') , se actualiza ϕ para conseguir ϕ' . Específicamente, se está

optimizando la cota inferior variacional de $D_{KL}(q(\theta|\phi)||p(h|\theta)p(\theta))$.

Se representa $q(\theta|\phi)$ como el producto de distribuciones gaussianas independientes con media ϕ .

Como resumen final, IG tiene como ventaja su formalismo matemático pero por otra parte los modelos son más complejos y difíciles de usar de forma efectiva ya que se tienen muchos hiperparámetros.

Lecturas recomendadas

- Schmidhuber. (1992). A Possibility for Implementing Curiosity and Boredom in Model-Building Neural Controllers.
- Stadie, Levine, Abbeel (2015). Incentivizing Exploration in Reinforcement Learning with Deep Predictive Models.
- Osband, Blundell, Pritzel, Van Roy. (2016). Deep Exploration via Bootstrapped DQN.
- Houthooft, Chen, Duan, Schulman, De Turck, Abbeel. (2016). VIME: Variational Information Maximizing Exploration.
- Bellemare, Srinivasan, Ostrovski, Schaul, Saxton, Munos. (2016). Unifying Count-Based Exploration and Intrinsic Motivation.
- Tang, Houthooft, Foote, Stooke, Chen, Duan, Schulman, De Turck, Abbeel. (2016). Exploration: A Study of Count-Based Exploration for Deep Reinforcement Learning.
- Fu, Co-Reyes, Levine. (2017). EX2: Exploration with Exemplar Models for Deep Reinforcement Learning.

16.7. Aprendizaje por Imitación vs. Aprendizaje por refuerzo

Imitación	RL
<ul style="list-style-type: none"> - Requiere demostraciones <li style="margin-left: 20px;">- <i>Distributional shift</i> <li style="margin-left: 20px;">+ Simple, estable y supervisado <li style="margin-left: 20px;">- Solamente tan bueno como la demostración 	<ul style="list-style-type: none"> - Requiere función de recompensa <li style="margin-left: 20px;">- Exploración <li style="margin-left: 20px;">- Potencialmente no convergente <li style="margin-left: 20px;">+ Puede ser arbitrariamente bueno

Interesa conseguir lo mejor de los dos mundos. Por ejemplo, ¿qué pasaría si se tienen demostraciones y recompensas?

Una de las formas más sencillas de usar ambas es preentrenar con las imitaciones y ajustar el agente resultante con las recompensas. Esto puede funcionar muy bien porque las demostraciones evitan que se pierda tiempo en la exploración y RL puede mejorar la efectividad por encima de la del demostrador.

Algoritmo 29: Preentrenar y ajustar

Recoger datos de demostraciones (s_i, a_i)

Inicializar π_θ como $\max_\theta \sum_i \log \pi_\theta(a_i|s_i)$

repetir

- | Ejecutar π_θ para recoger experiencia
- | Mejorar π_θ con cualquier algoritmo RL

hasta que se converja;

El algoritmo anterior puede no funcionar ya que se puede coger lo peor de los dos mundos. Es posible que se produzca el *distributional shift* en el primer paso y que después la política sea demasiado determinista como para explorar estados nuevos.

Si se tiene el primer batch de observaciones generadas que sean malas, puede destruir la inicialización hecha previamente.

Una de las cosas que se puede hacer para solucionar esto es usar un algoritmo *off-policy*, de forma que se pueda entrenar sobre las demostraciones en cada iteración.

Policy Gradient con demostraciones

Se incluyen muestras del demostrador en el dataset D . Aunque en D para Policy Gradient se ponían observaciones *on-policy*, poner demostraciones no es mala idea debido al IS óptimo, que se basa en que si se quiere calcular $E_{p(x)}[f(x)]$, la $q(x)$ que da la menor varianza es $q(x) \propto p(x)|f(x)|$.

Problemas:

- ¿De qué distribución vienen las demostraciones?
 - Opción 1: usar clonado de comportamiento supervisado para aproximar π_{demo} . No será una estimación buena pero será suficiente.
 - Opción 2: Se asume una distribución del tipo delta de Dirac: $\pi_{demo}(\tau) = \frac{1}{N}\delta(\tau \in D)$. Esto funciona mejor con IS autonormalizado (suma hasta 1):

$$E_{p(x)}[f(x)] \approx \frac{1}{\sum_j \frac{p(x_j)}{q(x_j)}} \sum_i \frac{p(x_i)}{q(x_i)} f(x_i) \quad (16.10)$$

- ¿Qué hacer si D viene de múltiples distribuciones? IS sólo funciona para una distribución, si se quiere hacer esto de manera correcta se tiene que usar una *fusion distribution*: $q(x) = \frac{1}{M} \sum_i q_i(x)$

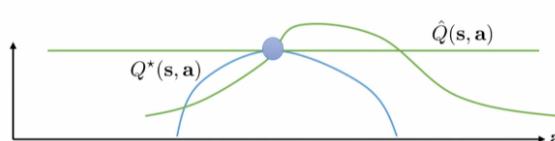
Q-Learning con demostraciones

Es bastante fácil hacerlos funcionar con demostraciones ya que estas simplemente se meten en el buffer, pero hay que gestionar algunas cosas.

Q-Learning ya es *off-policy* por lo que no hay que preocuparse con los pesos.

Problemas con estos planteamientos

- Importance Sampling es una receta para quedarse atascado. Ya que hace un buen trabajo aprendiendo de las demostraciones pero no tan bueno con RL. Esto es porque los productos que aparecen en PG tienden a hacer que en la función de coste hayan mesetas por cada observación. Como las demostraciones están más cercas unas de otras, el algoritmo tenderá a preferirlas ya que las mesetas serán menores.
- Con Q-Learning que los datos sean buenos no es suficiente. Y es que existen infinitas funciones que se ajustan a las acciones óptimas como se ve en la imagen.



Imitación como una función de pérdida auxiliar

El objetivo de la imitación es: $\sum_{(s,a) \in D_{demo}} \log \pi_\theta(a|s)$.

El objetivo RL es: $E_{\pi_\theta}[r(s, a)]$.

Por lo que se puede crear un objetivo híbrido:

$$E_{\pi_\theta}[r(s, a)] + \lambda \sum_{(s, a) \in D_{demo}} \log \pi_\theta(a|s) \quad (16.11)$$

Los problemas de esto son:

- Se tiene un hiperparámetro más: λ .
- El diseño del objetivo necesita mucha atención
- El algoritmo se vuelve dependiente al problema que se quiere resolver.

Tema 17

Meta Reinforcement Learning

17.1. ¿Cuál es el problema?

Normalmente los agentes inteligentes suelen ser especialistas en una tarea, pero se quiere que sean generalistas de tal forma que aprendan tareas de forma mucho más eficiente.

Por ejemplo un robot entrenado para hacer surf, no tiene por qué aprender de cero totalmente para aprender a montar en moto.

17.1.1. Enunciado del problema

En el aprendizaje por refuerzo clásico, se tiene que aprender un MDP:

$$\theta^* = \arg \max_{\theta} E_{\pi_{\theta}(\tau)}[R(\tau)] \quad (17.1)$$

$$= f_{RL}(M) \quad (17.2)$$

Donde θ son los parámetros de la política.

En Meta-RL, se tiene que aprender la **regla de adaptación**:

$$\theta^* = \arg \max_{\theta} \sum_{i=1}^n E_{\pi_{\phi_i}(\tau)}[R(\tau)] \quad (17.3)$$

$$\text{Donde } \phi_i = f_{\theta}(M_i) \quad (17.4)$$

Donde θ no tiene por qué ser los parámetros de la política, si no que puede ser los parámetros de la regla de adaptación. M_i hace referencia al MDP de la tarea i .

La expresión 17.3 se llama *meta-training* o bucle externo y la expresión 17.4 se llama adaptación o bucle interno.

17.1.2. Relación con las políticas orientadas a objetivos

En RL clásico se tiene un objetivo que queda representado por una función de recompensa. Por lo que el agente entrena una política condicionada a un conocimiento del entorno para actuar con respecto a ese objetivo.

En Meta-RL se pretende que el agente pueda actuar con nuevas funciones de recompensa o dinámicas del entorno.

Las recompensas son una generalización estricta de los objetivos, ya que estos se pueden representar con recompensas pero no al revés. Por ejemplo, no se puede representar como un estado objetivo la función de recompensa que describe buscar mientras se evita, o penalizaciones sobre las acciones.

17.1.3. Adaptación

La adaptación consiste en:

- Explorar: coleccionar los datos que mayor información den.
- Adaptar: Usar esos datos para obtener la política óptima.

Algoritmo 30: Boceto de un algoritmo Meta-RL general

Muestrar la tarea i para recoger los datos D_i

Adaptar la política calculando $\phi_i = f(\theta, D_i)$

Recoger los datos D'_i con la política adaptada π_{ϕ_i}

Actualizar θ de acuerdo con $L(D'_i, \phi_i)$

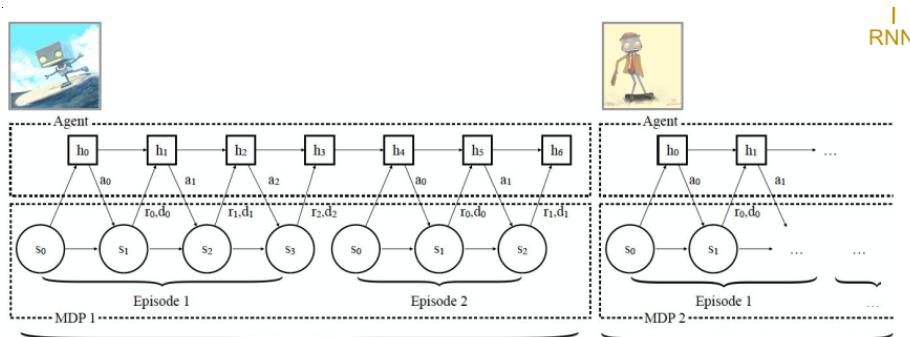
De este algoritmo, los primeros 3 pasos conforman la adaptación y pueden realizarse más de una vez en cada iteración. El cuarto paso en la práctica se suele actualizar sobre un *batch* de tareas.

Los diferentes algoritmos consisten en la elección de la función f y la función de pérdida L .

17.2. Métodos solución

17.2.1. Solución 1: Recurrencia

Se entrena una red neuronal recurrente con datos recogidos en forma de tuplas (estado, acción, recompensa) con *batches* de cada una de las tareas.



Algoritmo 31: Boceto de recursividad

mientras se entrene hacer

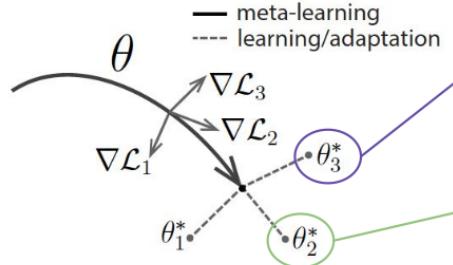
```

para i en tareas hacer
    Inicializar el estado oculto  $h_0 = 0$ 
    para t en pasos hacer
        1. Muestrear 1 transición  $D_i = D_i \cup \{(s_t, a_t, s_{t+1}, r_t)\}$  de  $\pi_{h_t}$ 
        2. Actualizar el estado oculto de la política  $h_{t+1} = f_\theta(h_t, s_t, a_t, s_{t+1}, r_t)$ 
    fin
fin
Actualizar los parámetros de la política  $\theta \leftarrow \theta - \nabla_\theta \sum_i L_i(D_i, \pi_h)$ 
fin
  
```

- Ventajas: es general y expresivo ya que hay RNNs que pueden calcular cualquier función.
- Desventajas: no es consistente. Un modelo es consistente si converge a la política óptima dados los suficientes datos.

17.2.2. Solución 2: Optimización

Extrapolando la idea de *fine-tuning*, la idea de la optimización radica en aprender una parametrización de modo que hacer fine-tuning para cada una de las tareas sea muy rápido.



En este caso, se tiene Policy Gradients tanto en la adaptación como en el bucle externo.

Algoritmo 32: Boceto de optimización

mientras se entrene hacer

para i en tareas hacer

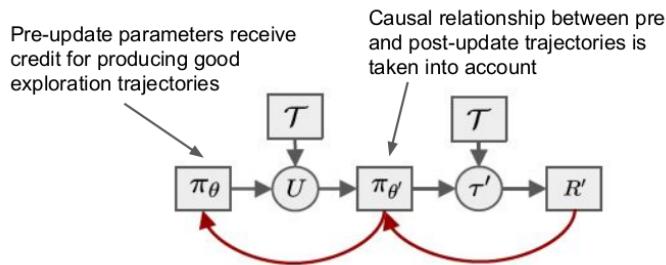
- Muestrear k episodios $D_i = \{(s, a, s', r)\}_{1:k}$ de π_θ
- Calcular los parámetros adaptados $\phi_i = \theta - \alpha \nabla_\theta L_i(\pi_\theta, D_i)$.
- Muestrear k episodios $D'_i = \{(s, a, s', r)\}_{1:k}$ de π_ϕ

fin

Actualizar los parámetros de la política $\theta \leftarrow \theta - \nabla_\theta \sum_i L_i(D'_i, \pi_{\phi_i})$.

fin

Nótese que en el último paso, cuando se calcula el gradiente se está haciendo con respecto al gradiente previo, por lo que se está calculando la derivada de segundo orden.



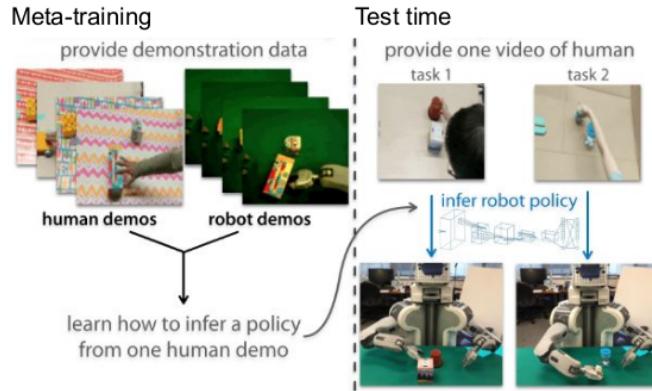
- Ventajas: es consistente ya que se está haciendo descenso por gradiente.
- Desventajas: no es tan expresivo. Sobretodo en entornos donde no se obtenga señal de recompensa la adaptación no cambiará la política, aunque los datos dan información sobre qué estados se deben de evitar.

17.3. Meta-RL en sistemas robóticos

17.3.1. Meta-imitation learning

Se puede entrenar un agente para que aprenda a realizar una tarea a partir de una demostración humana. Para el entrenamiento inicial se supone que se tienen pares de episodios tanto del humano como del agente. Se puede entrenar un agente para que aprenda a realizar una tarea a partir de una demostración humana. Para el entrenamiento inicial se supone que se tienen pares de episodios tanto

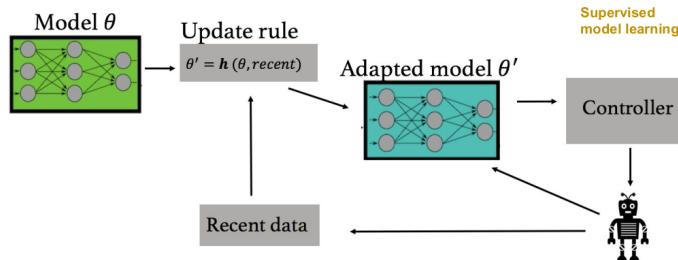
del humano como del robot, realizando la misma acción. Al mostrar la nueva acción del humano, el robot intentará imitarla.



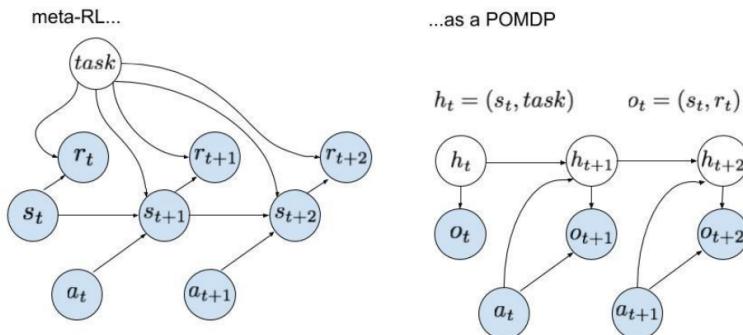
En este método se aprende la función de pérdida f_θ .

17.4. Meta-RL basado en modelo

En el aprendizaje basado en modelo, puede ocurrir que el entorno cambie ligeramente, haciendo que el modelo aprendido sea inútil hasta que no se reentrene el agente, por lo que es de interés que en vez de reentrenarlo con todos los datos para obtener el modelo se usen pocos datos nuevos y se parta de los conocimientos ya obtenidos.



17.5. POMDP



Hay dos aproximaciones para resolverlo:

- RNN
- Estimación explícita del estado.

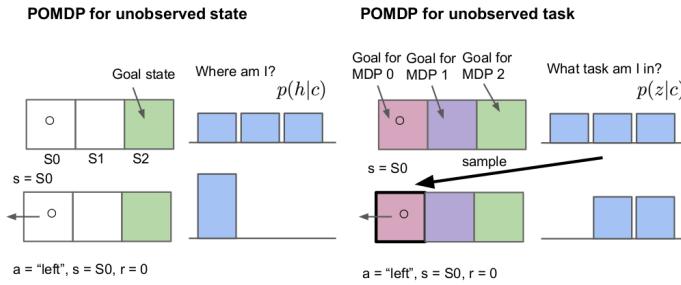
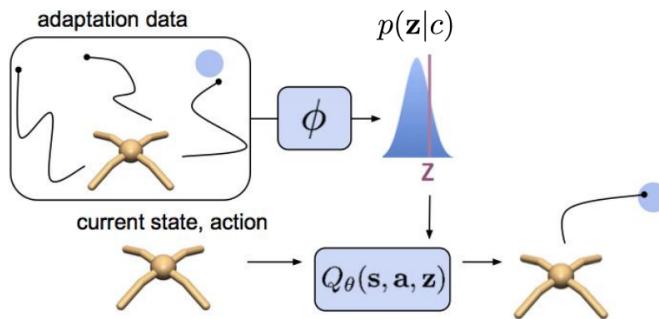


Figura 17.1: Model belief over latent task variables

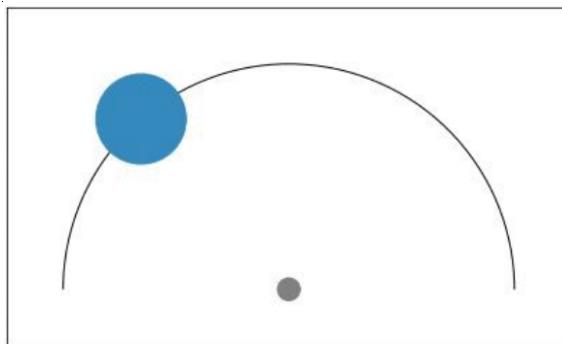
En este tema se va a tratar la segunda.

17.6. Solución 3: Estados de creencia de tarea (task-belief states)

Al entrenar para varias tareas, se va a ajustar un encoder estocástico para saber a qué tarea pertenecen los estados observados.



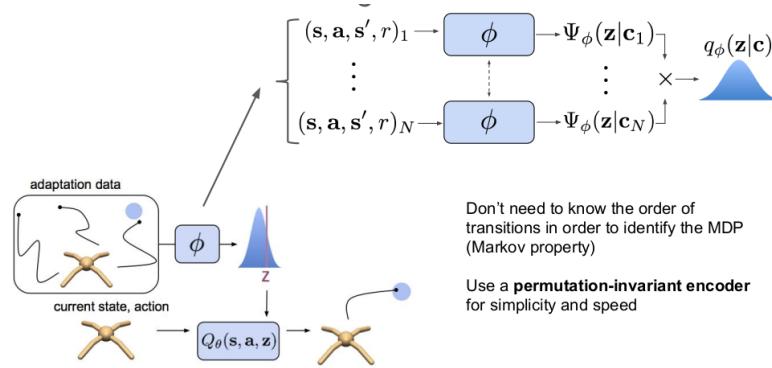
Se puede entrenar generando objetivos que se creen posibles, por ejemplo:



En este entorno se quiere ir del centro del círculo al círculo azul. Una vez el agente haya aprendido a llegar, se pueden generar objetivos 'falsos' pensando que el círculo azul puede estar en cualquier parte de la circunferencia, lo que hace que se aprenda muy rápido en caso de que el área azul se mueva.

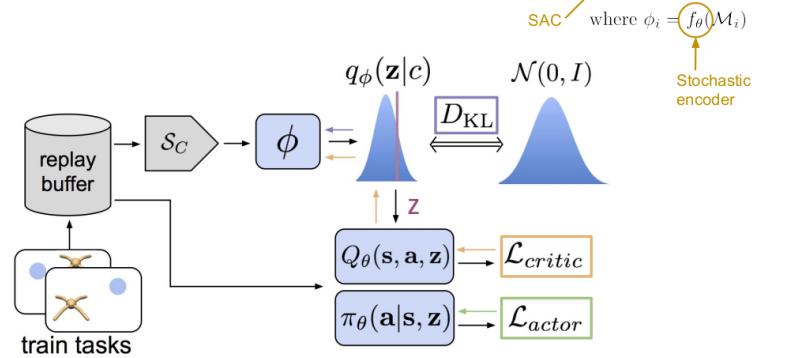
Aprender $p(z|c)$ es intratable, por lo que se hará uso de la inferencia variacional.

$$\mathbb{E}_\tau[\mathbb{E}_{z \sim q_\phi(z|c^\tau)}[R(\tau, z) + \beta D_{KL}(q_\phi(z|c^\tau)||p(z))]] \quad (17.5)$$

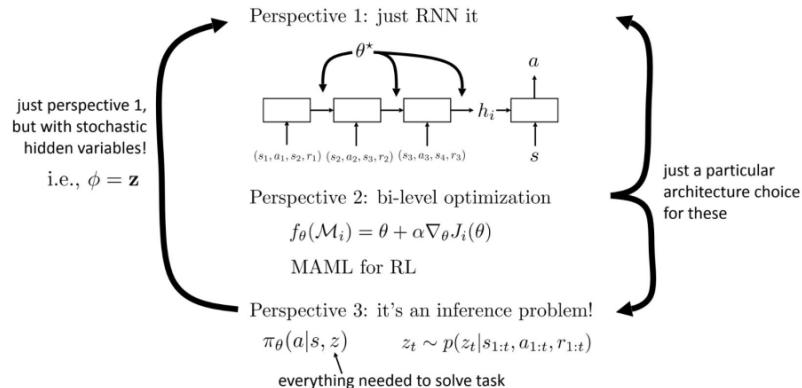


Soft Actor-Critic (SAC) + creencia de la tarea

Solution #3: task-belief + SAC



17.7. Resumen



Tema 18

Exploración Información-Teórica, Desafíos y Problemas Abiertos

Clase 20: Information-Theoretic Exploration, Challenges and Open Problems

2020-08-07

18.1. Exploración sin una función de recompensa

Hay varios motivos para conseguir un comportamiento de esta manera:

- Aprender habilidades sin supervisión y usarlas después para cumplir objetivos.
- Aprender sub-habilidades para usarlas en RL jerárquico.
- Explorar el espacio de los posibles comportamientos.

18.1.1. Definiciones y conceptos de la teoría de la información

- $p(x)$: distribución.
- $H(p(x)) = -E_{x \sim p(x)}[\log p(x)]$: entropía
- Información mutua, dice cuánto de dependientes son las dos variables. Es la identidad más importante de las tres:

$$I(x; y) = D_{KL}(p(x, y) || p(x)p(y)) \quad (18.1)$$

$$= E_{(x,y) \sim p(x,y)} \left[\log \frac{p(x, y)}{p(x)p(y)} \right] \quad (18.2)$$

$$= H(p(y)) - H(p(y|x)) \quad (18.3)$$

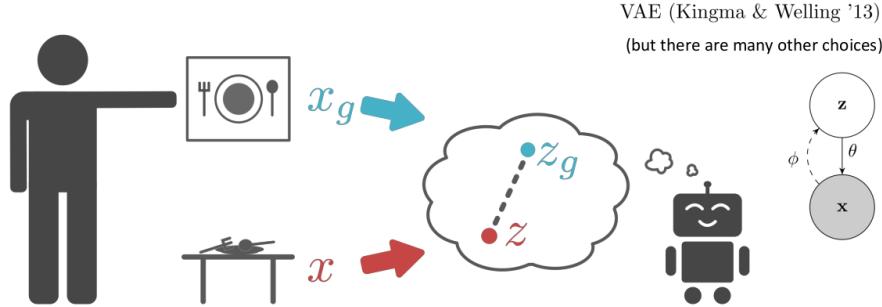
Cantidades de la teoría de la información en RL:

- $\pi(s)$: distribución marginal de estados de la política π
- $H(\pi(s))$: cuantifica la cobertura de $\pi(s)$.
- *Empowerment*: la habilidad que tiene el agente de afectar al mundo, se calcula como:

$$I(s_{t+1}; a) = H(s_{t+1} - H(s_{t+1}|a_t)) \quad (18.4)$$

18.1.2. Aprender sin una función de recompensa, completando objetivos

En este caso, se tiene un estado inicial y se le dice al agente que tiene que llegar al estado deseado interaccionando con el mundo. El agente construye una representación interna, mediante un VAE por ejemplo.



Una forma de aprender sería que el agente intente llegar del estado inicial a otros estados aleatorios, entrenando su política para alcanzar estos estados. Pero esto tiene el problema de que no se explora el espacio de forma óptima, ya que el agente sólo explorará entornos cercanos porque son los que su modelo generativo puede generar. Lo que hace que 'vaya en círculos'.

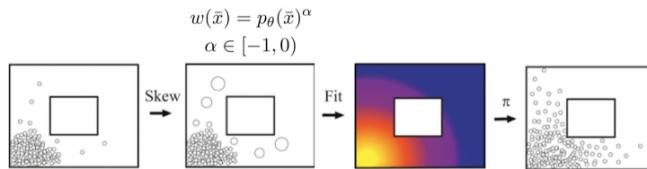
En general, el algoritmo es así:

1. Se propone un objetivo: $z_g \sim p(z)$, $x_g \sim p_\theta(x_g|z_g)$
2. Se intenta alcanzar el objetivo con $\pi(a|x, x_g)$, se llega a \bar{x} .
3. Se usan los datos para actualizar π .
4. Se usan los datos para actualizar $p_\theta(x_g, z_g)$, $q_\phi(z_g|x_g)$

Pero se tiene que encontrar una forma de diversificar los objetivos. La clave está en modificar el paso 4, de tal forma que se haga más énfasis en los estados nuevos. Por lo que en vez de usar MLE estándar, se pasa a usar MLE ponderado:

$$\theta, \phi \leftarrow \arg \max_{\theta, \phi} E[w(\bar{x}) \log p(\bar{x})] \quad (18.5)$$

Si se elige $w(\bar{x}) = p_\theta(\bar{x})^\alpha$ $\alpha \in [-1, 0]$, la entropía $H(p_\theta(x))$ se incrementa.

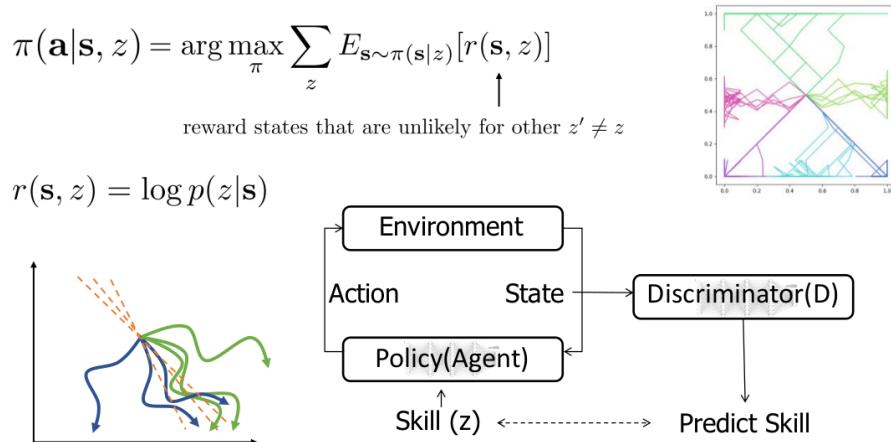


Esto hace que el objetivo sea maximizar $H(p(G))$. Pero por el otro lado, RL está entrenando $\pi(a|S, G)$ para alcanzar G , por lo que mientras π va mejorando, el estado final S se va aproximando a G , lo que significa que $p(G|S)$ se va haciendo más determinista.

Por lo que el objetivo se puede expresar como $\max H(p(G)) - H(p(G|S))$, lo que significa que es la información mutua: $\max I(S; G)$.

18.1.3. Mas allá del cubrimiento de estados: cubriendo el espacio de habilidades

Intuición: habilidades diferentes deben de visitar diferentes regiones del espacio de estados.



Lo anterior equivale a maximizar la información mutua pero de z y s :

$$I(z, s) = H(z) - H(z|s) \quad (18.6)$$

El primer término es maximizado usando $p(z)$ y el segundo es minimizado mediante la maximización de $\log p(z|s)$.

18.2. Desafíos en Deep Reinforcement Learning

Problemas en los algoritmos principales:

- Estabilidad: convergencia.
- Eficiencia: muestras necesarias.
- Generalización: después de la convergencia, generaliza?

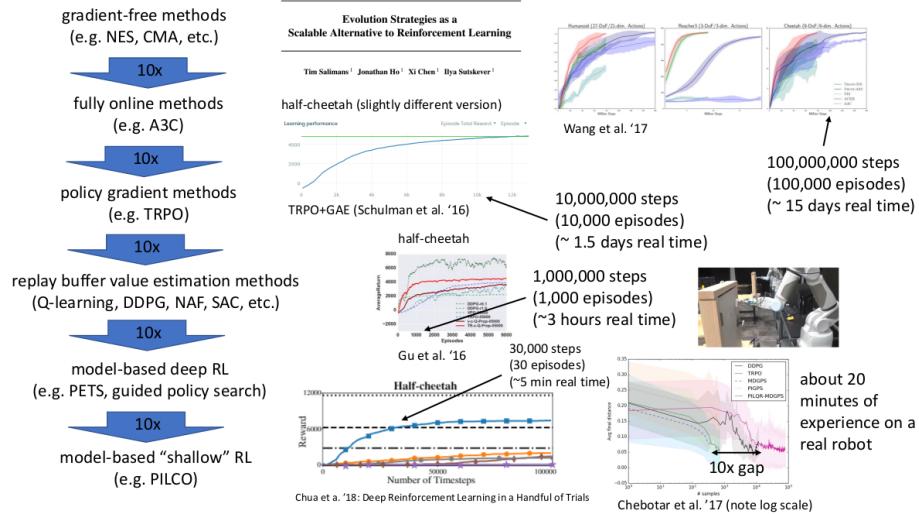
Problemas con las asunciones:

- RL provee de una buena formulación a problemas pero la pregunta es si esta formulación es la correcta.
- ¿De dónde viene el conocimiento?

Problemas con los hiperparámetros:

- No se puede hacer un barrido de hiperparámetros en la vida real. Normalmente los parámetros obtenidos en simuladores no son representativos.
- Existen algoritmos que buscan una mejora de las propiedades de convergencia como TRPO u otros que ajustan adaptativamente sus hiperparámetros.

18.2.1. Complejidad del muestreo



18.2.2. Escalado y generalización

DL en el aprendizaje supervisado consigue resultados a grande escala, enfatiza la diversidad y está evaluada en la generalización. Por otra parte, DRL es de escala pequeña y enfatiza el dominio de una tarea evaluada, por lo que no hay generalización.

Índice general

1. Course Introduction, Imitation Learning	1
1.1. ¿Qué es RL y por qué es importante?	1
1.2. ¿Que otros problemas tienen que resolverse para habilitar un proceso de toma de decisiones en el mundo real?	1
1.3. ¿De donde vienen las recompensas?	2
1.4. ¿Qué se puede hacer con un modelo perfecto?	2
1.5. ¿Cómo se crean las máquinas inteligentes?	2
1.6. ¿Qué tendría que hacer una algoritmo así?	2
1.7. ¿Por qué usar DRL?	2
1.8. ¿Qué puede hacer bien actualmente DRL?	2
1.9. ¿Cuáles son actualmente los desafíos de DRL?	3
1.10. Terminología y notación	3
1.11. Imitation Learning	3
1.11.1. ¿Se puede hacer trabajar Imitation Learning con menos datos?	5
1.11.2. ¿Por qué podemos fallar al entrenar el experto?	5
1.11.3. ¿Qué funciones de coste son buenas para Imitation Learning?	7
2. Introducción al aprendizaje por refuerzo	9
2.1. Procesos de decisión de Markov (MDP)	9
2.2. Definición del problema de RL	9
2.2.1. El objetivo de RL	9
2.3. Anatomía de un algoritmo de RL	11
2.3.1. ¿Cómo se gestionan todas las esperanzas?	11
2.4. Resumen de los tipos de algoritmos de RL	12
2.4.1. ¿Por qué hay tantos métodos distintos?	12
3. Policy Gradients	14
3.1. El algoritmo Policy Gradient	14
3.2. ¿Qué hace Policy Gradient?	15
3.3. Reducir la varianza: Causalidad	15
3.4. Reducir la varianza: Baselines	16
3.5. Derivación de Policy Gradient con Importance Sampling	17
3.6. Implementar Policy Gradient con diferenciación automática	17
3.7. Policy Gradients en la práctica	18
3.8. Resumen	18
4. Métodos Actor Critic	19
4.1. Mejorando Polocy Gradient con un crítico	19
4.2. El problema de evaluación de la política	20
4.3. Factores de descuento	21
4.4. El algoritmo Actor Critic	22
4.4.1. Diseño de la arquitectura	23

4.4.2. Online Actor-Critic en la práctica	23
4.4.3. Críticos como baselines dependientes del estado	24
4.5. Elegibility traces y n-step returns	25
4.5.1. Generalized Advantage Estimation	25
5. Métodos de Funciones Valor	27
5.1. Usando solamente un crítico, sin un actor	27
5.1.1. Programación Dinámica	28
5.1.2. Fitted Value Iteration	29
5.2. Q-Learning	30
5.2.1. Online Q-Learning	30
5.2.2. Exploración con Q-Learning	30
6. Deep RL con funciones Q	32
6.1. Hacer funcionar Q-Learning con redes neuronales	32
6.1.1. Muestras relacionadas en <i>online</i> Q-Learning	32
6.1.2. Q-Learning con Target Networks	33
6.2. Una vista generalizada de los algoritmos Q-Learning	33
6.3. Trucos prácticos para mejorar Q-Learning	34
6.3.1. ¿Son los valores Q precisos?	34
6.3.2. Double Q-Learning	35
6.3.3. Multi-step returns	36
6.4. Métodos Q-Learning continuos	36
6.5. Consejos prácticos para Q-Learning	37
7. Policy Gradient Avanzado	39
7.1. ¿Por qué los métodos Policy Gradient funcionan?	39
7.2. Policy Gradient es un tipo de iteración de política	39
7.3. Policy Gradient como una optimización con restricciones	43
7.3.1. ¿Es todo esto necesario?	45
7.4. Resumen	46
8. Control Óptimo y Planificación	47
8.1. Introducción a RL basado en modelo	47
8.2. Si conocemos las dinámicas, ¿cómo se toman las decisiones?	48
8.3. Métodos estocásticos de optimización	48
8.4. Monte Carlo Tree Search	49
8.5. Optimización de trayectorias	51
9. RL basado en modelo	56
9.1. Lo básico de RL basado en modelo.	56
9.1.1. El por qué una aproximación sencilla no funciona	56
9.2. Incertidumbre en RL basado en modelo	58
9.2.1. Planificación con incertidumbre	60
9.3. RL basado en modelo con observaciones complejas	61
9.3.1. State space (latent space) models	61
9.3.2. Aprender directamente en el espacio de observaciones	62
10. Inferencia variacional y modelos generativos	63
10.1. Modelos probabilísticos de variables latentes	63
10.1.1. Modelos con variables latentes en RL	64
10.2. Inferencia variacional	64
10.3. Inferencia variacional amortizada	66
10.3.1. Policy Gradient vs el truco de la reparametrización	68
10.4. Modelos generativos: autoencoder variacional	68

10.4.1. Modelos condicionales	68
10.5. Ejemplos	69
11. Aprendizaje de política basado en modelo	70
11.1. Optimización sin modelo con un modelo	71
11.2. Modelos locales	72
11.3. Combinar modelos locales en un modelo global	74
12. Reformulación del Control como un problema de inferencia	76
12.1. ¿El control óptimo y RL son capaces de proveer un modelado razonable del comportamiento humano?	76
12.1.1. Control Óptimo como un modelo del comportamiento humano	76
12.1.2. Inferencia en este modelo	77
12.1.3. Resumen	81
12.2. ¿Hay una explicación mejor?	81
12.2.1. El problema del optimismo	81
12.3. Q-Learning con soft-optimality	82
12.4. Policy Gradient con soft-optimality	83
12.5. Beneficios de la optimalidad blanda	83
13. Aprendizaje por refuerzo inverso	84
13.1. ¿Por qué deberíamos preocuparnos en aprender las recompensas?	84
13.1.1. La perspectiva del aprendizaje por imitación	84
13.1.2. La perspectiva del aprendizaje por refuerzo	84
13.2. Aprendizaje por refuerzo inverso	84
13.2.1. Un poco más formalmente	85
13.2.2. Aprendizaje de la variable de optimalidad	85
13.2.3. La función de partición de RL inverso	86
13.3. Estimar la esperanza	86
13.3.1. Con dinámica desconocida y espacio de acciones/estados grandes	88
14. Aprendizaje transferible y multi-tarea	91
14.1. ¿Cuál es el problema?	91
14.1.1. ¿Puede RL usar el mismo conocimiento previo que nosotros?	91
14.2. Terminología del aprendizaje transferible	91
14.3. ¿Cómo se pueden plantear los problemas de aprendizaje por transferencia?	92
14.3.1. Forward Transfer	92
14.3.2. Finetuning	92
14.3.3. Multi-task transfer	94
15. RL Distribuido	97
15.1. Patrones computacionales comunes en RL	97
15.1.1. Otras arquitecturas distribuidas interesantes	99
15.2. RLLib: Abstractions for Distributed Reinforcement Learning (ICML 2018)	100
16. Exploración	102
16.1. ¿Qué es la exploración? ¿Por qué es un problema?	102
16.2. <i>Multi-armed bandits</i>	102
16.3. Exploración basada en optimismo	103
16.4. Posterior matching exploration	103
16.5. Information-theoretic exploration	103
16.6. Métodos de exploración en DRL	104
16.7. Aprendizaje por Imitación vs. Aprendizaje por refuerzo	106
17. Meta Reinforcement Learning	109

17.1. ¿Cuál es el problema?	109
17.1.1. Enunciado del problema	109
17.1.2. Relación con las políticas orientadas a objetivos	109
17.1.3. Adaptación	110
17.2. Métodos solución	110
17.2.1. Solución 1: Recurrencia	110
17.2.2. Solución 2: Optimización	111
17.3. Meta-RL en sistemas robóticos	111
17.3.1. Meta-imitation learning	111
17.4. Meta-RL basado en modelo	112
17.5. POMDP	112
17.6. Solución 3: Estados de creencia de tarea (task-belief states)	113
17.7. Resumen	114
18. Exploración Información-Teórica, Desafíos y Problemas Abiertos	115
18.1. Exploración sin una función de recompensa	115
18.1.1. Definiciones y conceptos de la teoría de la información	115
18.1.2. Aprender sin una función de recompensa, completando objetivos	116
18.1.3. Mas allá del cubrimiento de estados: cubriendo el espacio de habilidades	116
18.2. Desafíos en Deep Reinforcement Learning	117
18.2.1. Complejidad del muestreo	118
18.2.2. Escalado y generalización	118