- HuggingFace Deep RL course (https://huggingface.co/deep-rl-course/unit0/introduction?fw=pt)
- HuggingFace Deep RL course github (https://github.com/huggingface/deep-rl-class)
- Reinforcement Learning book: Sutton (http://incompleteideas.net/book/RLbook2020.pdf)

Reinforcement Learning

We have previously consider two branches of Machine Learning

- Supervised Learning
- Unsupervised Learning

There is a third branch called *Reinforcement Learning*.

Reinforcement Learning is the process whereby an *agent*

- learns to perform a task involving a *sequence* of decisions.
- through the *experience* gained by making decisions

An analogy: think about how someone learns to play a game of cards

- No initial knowledge (other than rules of the game)
- Plays multiple games
    - After each move: player receives a measure of the quality of the move
    - Player updates the policy for making moves based on this feedback

The world in which the agent operates is called the *environment*.

This world contains a set of relevant features of the world (called the *state*) and the laws/rules that govern behavior in this world.

- rules of a game
- laws of physics

The agent may only have a partial view (an *observation*) of the state

- the agent's own hand, but not its opponent's hand, in a game of cards

At each step: the agent must make a decision (choose an *action*) based on the observation, resulting in

- an updated environment (and resulting observations)
- the agent receiving a *reward*

The goal is for the agent to maximize the rewards received over the lifetime of decisions (the *episode* or *trajectory*)

The *policy* is the function that guides the actions decisions.

An *episode* (or *trajectory*) is a sequence that records the events as agent follows its policy in making decisions.

Here is a timeline of an episode

- column labeled "Agent": actions chosen by the Agent
- column labeled "Environment": the responses generated in reaction to the decision

Step	Agent	Environment	Notes
0		S_0	Environment chooses initial state
	$\pi(S_0)$	R_1, S_1	
1	$\pi(S_1)$	R_2, S_2	

$\vdots$ t | $\pi(S_t)$ | R_{t+1} , S_{t+1} | Agent follows policy to choose action $\pi(S_t)$ | | | Environment responds to action by giving reward R_{t+1} and changing state to S_{t+1} $\vdots$

Consider how an agent might learn to play a two-player game of cards called "21"

- The *state*
    - agent's cards
    - opponent's cards
    - cards in the deck
- An *observation*
    - the agent's cards
    - the *number* of cards remaining in the deck
- The *actions* of the agent
    - ask for an additional card
    - hold
- The environment updates as a result of the agent's action
    - The player's hand is updated
    - The opponent makes its own decision, updating it's hand
    - The decks is updated (if the agent receives a card)
    - The agent receives a reward
        - in some games, the only rewards are received at the game's end
- The game continues
    - until a terminal state is reached: both players do not/can not make additional decisions

On a superficial level we can try treating this as a case of Supervised Learning similar to the "predict the next" (Language model) task.

- the episode is encoded as a sequence of tuples
    - state, action, reward, next state
- the agent tries to extend the sequence to produce the next action
    - the environment then extends the sequence with the reward and next state

Offline RL can be cast as a Superivised Learning task on sequences.

The fundamental departure from Supervised Learning

- the *agent's action* at step t affects the future values of the sequence
- the *agent's policy* after step t is changed by the feedback (reward of action)

Formally, we have described a *Partially Observable Markov Decision Process (POMDP)*

If the state is *completely observable* to the agent (*observation* = state*) then this is called simply a*Markov Decision Process (POMDP)*

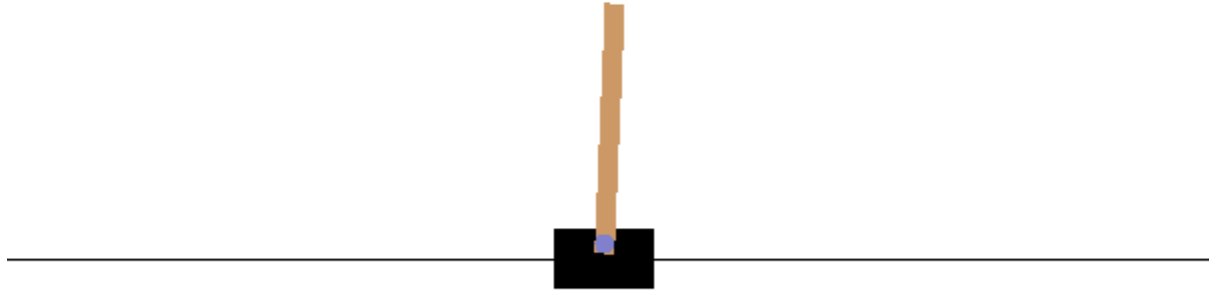
For easy of presentation: we will assume *observation = state*

Here is some code (external/handson-ml2/18_reinforcement_learning.ipynb#A-simple-hard-coded-policy) from Geron's book

- the game: agent tries to balance a pole
- actions: move base (black box at bottom) right/move left
- reward: if move in direction that keeps pole upright

In [2]:
```
from IPython.display import Image
Image(open('images/cart_pole.gif','rb').read())
```

Out[2]:



Code

- each episode has 200 moves
- the game is played for 500 episodes
- the *observation* is
    - direction base is moving (right == 1)
    - the angle of the pole (positive: leaning right)
    - angular velocity of the pole (positive: tilting right)
- action $\in 0, 1$: move left/move right
- policy: move left if angle is negative; move right otherwise

env.seed(42) def basic_policy(obs): angle = obs[2] return 0 if angle < 0 else 1 totals = [] for episode in range(500): episode_rewards = 0 obs = env.reset() for step in range(200): action = basic_policy(obs) obs, reward, done, info = env.step(action) episode_rewards += reward if done: break totals.append(episode_rewards)

Challenges

- On-line versus off-line training
    - On-line training data: training examples obtained through *experience*: play the game
    - Your initially un-trained, poor action choices affect the examples you learn from
- Sparse rewards
    - If rewards are only received at game's end:
        - how do you attribute the end reward with the action choice at each move ?
    - how do you rank the possible action choices at each move without immediate reward feedback ?
- The optimal reward may not be knowable (no *right* answer)
    - what is the *best* move in a game of chess ?
        - may be conditional on the quality of your opponent
        - may not be an ominiscient opponent to learn from
    - can possibly find better answers through *exploration* of previously untested actions
        - rather than *exploitation* of the best previously-tried action
    - Sutton: Evaluative vs Instructive feedback (ncompleteideas.net/book/RLbook2020.pdf#page=4)
        - you are not *instructed* with the best action
        - you are given an evaluation of your action through feedback

Deep Reinforcement Learning

Deep Reinforcement Learning refers to the special case of Reinforcement Learning where

- the *policy* is a parameterized (by θ) function mapping states S to (a probability distribution) actions

$$\pi_\theta(A|S)$$

- implemented as a Neural Network

Our initial presentation will be of fixed (non-parameterized) policies.

- We will subsequently introduce parameterized Neural Networks to implement the functions we define

Notation

Term	Definition
$\mathcal{S}$	**Set** of possible states
$\mathcal{A}$	**Set** of possible actions
$\mathcal{R}$	**function** $\mathcal{S} \times \mathcal{A} \to \mathbb{R}$
	maps state and action to a reward
γ	discount factor for reward one step in future
S_t	The state at beginning of time step t
A_t	The action performed at time step t
R_t	The reward resulting form the action performed at time step t
P	**Function** $\mathcal{S} \times \mathcal{A} \to \mathcal{S} \times \mathcal{R}$
	Transition probability: maps state and chosen action
	to new state and reward received

$S_0, A_0, R_1, \ldots S_t, A_t, R_{t+1}, \ldots$ **sequence** : Episode/Trajectory

sequence of states, action performed, reward r

$P(s', r|s, a)$ **Transition probability** (rules of the game)

$= P(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a)$

Markov Decision Process: depends only or

and not on history $S_0, \ldots, S_{t-1}$

$\pi(a|s)$ **probability** Policy (decision process for agen

$\pi(A_t|S_t) = (R_{t+1}, S_{t+1})$

action A_t occurs in state S_t

resulting in reward R_{t+1} and transition to stat

Notes on episodes

- The triple of elements corresponding to experience number t
  - is S_t, A_t, R_{t+1}
  - **not** S_t, A_t, R_t
  - **not** R_t, S_t, A_t
    - i.e., there is a reward (without action) from being in the initial state
    - some presentations use this; we will adopt the notational standard of the Sutton and Barto book (http://incompleteideas.net/book/RLbook2020.pdf).
- The Algorithms evaluating experience number t
  - **do not have access** to future experiences numbered $t' > t + 1$
  - they gain access to the next experience $t + 1$
    - by "playing the game"
    - submitting A_t to the environment and receiving R_{t+1} and S_{t+1}
  - Under the assumption of MDP (Markov Decision Process)
    - the algorithm *does not need* access to experiences numbered $t' < t$
- Episodes are more a notation/record than a piece of data used by an algorithm

Solving an RL system

A policy π is a function mapping a state to an action.

It is the "algorithm" that guides the actions behavior.

The "solution" to an RL system is the optimal policy π^* that maximizes *return* from the initial state

- return is sum of discounted rewards accumulated by following the policy from a given state $$ \begin{array} \ G*tt & = & \sum*{k=0}^\tt { \gamma^k * \rewseq_{\tt+k+1} } \
  ```
  & = & \rew_{\tt+1}  + \gamma * G_{\tt+1} \\
  ```
  \end{array} $$ where γ is a factor for discounting future returns.

One approach to finding the optimal policy is to

- learn a "model" of the environment (state and transition probabilities)
- derive optimal actions (given a state) through knowledge (via the model) of how the environment will respond

This solution method is called *model-based*.

We will focus on *model-free* methods

How do we find the optimal policy π ?

The way we find the optimal policy is typically via an iterative process

- We construct a sequence of improving policies
  $$\pi_0, \ldots, \pi_p, \ldots$$
  that hopefully converges to π^*.

There are two main directions for constructing the policy

- Value-based
- Policy-based

Value-based methods: concepts

Suppose we could assign a *value* to each state

- the sum of rewards over the episode starting with this state
- this is called the *return*

We could define a policy based on these state values

- choose the action that leads to the state with highest value ("greedy" policy)

This is the basis of valued-based methods.

A valued-based method assigns a value to each state

- constructs a function $v_\pi : \mathcal{S} \to \mathbb{R}$
- the process of assigning values to states will also be iterative
- We will build a sequence of hopefully more accurate approximations of v_π

$$v_{\pi,0} \ldots v_{\pi,k} \ldots$$

For simplification, let us assume for the moment that

- the sets of states, actions and rewards be *finite*.

Evaluating a policy: State-Value function:

We can formalize the value of a policy with the help of a little notation.

The *state-value function* for policy π is a map from states to (discounted) future rewards

- let G_t denote the *return* discounted future rewards from state number t in the episode sequence

$$G_t = \sum_{k=0}^{t} \gamma^k * R_{t+k+1}$$

$$v_\pi(s) : \mathcal{S} \to \mathbb{R}$$

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi(G_t | S_t = s) \\ &= \mathbb{E}_\pi\left(\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s\right) \end{aligned}$$

One can express the state-value function in recursive form

- the *Bellman equation* for the state-value function

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi(G_t|S_t = s) \\
&= \mathbb{E}_\pi\left(R_{t+1} + \gamma G_{t+1}|S_t = s\right) && \text{immediate reward } R_{t+1} \\
& && \text{plus discounted future rewards } G_t \\
&= \mathbb{E}_\pi\left(R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s\right) && \text{since } v_\pi(S_{t+1}) = \mathbb{E}_\pi\left(G_{t+1}|S_t = s\right)
\end{aligned}
$$

Notice the recursive relationship between the action-value function of current state $v_\pi(s)$ and the successor state $v_\pi(s')$

This formulation leads to the recursive Dynamic Programming solution.

If we had perfect knowledge of the environment (P in particular) we could mathematically evaluate $v_\pi(s)$.

Continuing the derivation (by evaluating the Expectation):

$$
\begin{aligned}
&= \sum_a \pi(a,s) \sum_{s'} \sum_r P(s',r|s.a) \left(r + \gamma \mathbb{E}_\pi(G_{t+1}|S_{t+1}=s')\right) && \text{expectation} \\
& && \text{of immediat} \\
& && \text{and discoun} \\
& && \text{resulting fro} \\
&= \sum_a \pi(a,s) \sum_{s',r} P(s',r|s.a) \left(r + \gamma v_\pi(s')\right)
\end{aligned}
$$

Of course, we can't evaluate the equation without knowing the Transition Probability function P so this is just theoretical.

What we need to do in practice is to create an *approximation* of v_π.

- by gaining experience from playing episodes

Learning from experience is one of the key aspects of Reinforcement Learning.

Evaluating a policy: Action-Value function

The state-value function associates the return (discounted future rewards) with a state, without reference to an action.

We can get more fine-grained with an *action-value function* that maps (state, action) pairs to (discounted) future rewards.

$$q_\pi : s \times a \to \mathbb{R}$$

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi(G_t | S_t = s, A_t = a) \\ &= \mathbb{E}_\pi \left(R_{t+1} + \gamma \max_{a'} q_\pi(s_{t+1}, a') \right) \\ &= \mathbb{E}_\pi \left(\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right) \end{aligned}$$

$\text{immediate reward } R_{t+1}$

$\text{plus discounted future rew}$

$\text{choosing best action } a' \text{ in}$

This helps us to discover the optimal action given a state

- Find the action a
- that maximizes the value of future rewards

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, q(s, a)$$

Iterative approximation of v_π

Reinforcement Learning algorithms usually seek to *approximate* the value of a strategy

- through an iterative process
- based on experience (episodes)

We will build a sequence of hopefully more accurate approximations of v_π

$$v_{\pi,0} \ldots v_{\pi,k} \ldots$$

beginning with an uninformed initial approximation $v_{\pi,0}$

Approximation $v_{\pi,k+1}$ improves on $v_{\pi,k}$ through the experience gained in one (or more) episodes.

Iterative improvement of policy

The optimal policy π^* is the one with greatest v_π for all states $s \in \mathcal{S}$

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, v_\pi(s)$$

We often find π^* by an iterative process.

That is, we create a sequence of improving policies

$$\pi_0, \ldots, \pi_p, \ldots$$

which hopefully converges to π^*.

Once we have an approximation of v_π for the current policy π_p

- we can derive an improved policy π_{p+1}
- by a policy update:
  - for each $s \in \mathcal{S}$

$$\begin{aligned} \pi'(s) &= \operatorname*{argmax}_a \textstyle\sum_{s',r} P(s', r|s, a)(r + \gamma v_\pi(s')) \quad \text{chose value r} \\ &= \operatorname*{argmax}_a v_\pi(s) \end{aligned}$$

  - if $\pi' \neq \pi$, replace v_π with $v_{\pi'}$

We can perform repeated policy updates until $\pi_{p+1} = \pi_p$

- note that the policy produced is *deterministic* (because of the argmax)

Updating slowly: Monte-Carlo and Temporal Difference

Improvement of both State-value function or Action-value functions is iterative.

One might imagine that, once we have a better approximation, we should use it immediately.

In fact, we will typically phase in improvements using a *learning rate* α

- similar to how we use a learning rate in conjunction with gradients in Gradient Descent.

  new value = old value + alpha * (new value - old value)

The reasons for this are

- the update is *locally correct* (e.g., for state S)
- but perhaps not *globally* correct
    - a transition *from* a source state s' leading to updated target state s may no longer be optimal for the source state s'

Let's illustrate with updates to v_π and q_π.

By playing a single episode we can calculate G_t for each step t in the episode.

G_t affects v_π for state s through the above equations

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi(G_t|S_t = s) \\
&= \mathbb{E}_\pi\left(R_{t+1} + \gamma G_{t+1}|S_t = s\right) \quad \text{immediate reward } R_{t+1} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{plus discounted future rewards } G_{t+1}
\end{aligned}
$$

Since $v_\pi(s_t) = G_t$, we might be tempted to update

$$v_{\pi,k+1}(s_t) \quad = \quad G_t$$

Instead, we use a *learning rate* α, just like Gradient Descent, to slowly introduce the updated value.

$$v_{\pi,k+1}(s_t) \quad = \quad v_{\pi,k}(s_t) + \alpha * (G_t - v_{\pi,k}(s_t)) \quad \text{Update by a fraction of the cha}$$

This update is called the *Monte Carlo* method.

There is an alternative to the Monte Carlo approach.

Rather than using G_t (sum of rewards from a state) for updating, we use the *immediate* reward R_t.

$$v_{\pi,k+1}(s_t) \quad = \quad v_{\pi,k}(s_t) + \alpha * (R_{t+1} + \gamma v_{\pi,k+1}(S_{t+1}) - v_{\pi,k}(s_t)) \quad \text{since } G_t = R_t$$

$$\text{immediate re}$$

$$\text{plus discount}$$

$$\text{Update by a f}$$

This is called the *Temporal Difference (TD)* method

- can be applied at *each step* of the episode
- rather than at the *end* of the episode as in Monte Carolo

Greed is not always good: ϵ-Greedy Policy

The way we hope to learn the optimal (or at least, a good) policy is through experience

- many episodes

This may tempt us to *always* choose the "best" action (known from experience thus far) in a given state.

Greed is not the friend of learning

Recall: We initialize v_π and q_π with uninformed choices.

- the initial "best" action a for state s is uninformed

If we always choose the same action a when visiting state s

- We may never getter better information on the return of an adjacent state s' reachable by action a'
- The alternate action a' be have higher return

A *greedy policy* is one that always chooses the action with highest return

- as estimated by our past experience

An ϵ*-greedy policy* is a combination of a greedy policy and random action

- chooses the action of the greedy policy with probability $(1 - \epsilon)$
- chooses a random action with probability ϵ.

An ϵ-greedy policy tries to balance

- *exploitation*: choose the best action (as know from past experience)
- *exploration*: gaining a better mapping of the environment by choosing alternative actions

Value-based algorithms to find π^*

Value-based algorithm 1: Policy iteration

Policy iteration is an algorithm that improves π_p to π_{p+1} by alternating two steps during round p

- Evaluation step: computes v_{π_p}
- Policy improvement step: creates π_{p+1} using v_{π_p}

The Evaluation step is *also* iterative:

- we find v_{π_p} as the limit of a sequence of increasingly better approximations
$$v_{\pi,0}, \ldots, v_{\pi,k}, \ldots$$
which hopefully converges to the true v_π.

Here is some pseudo-code (details to follow)

$\pi_0 = \text{initialize}$

$v_{\pi,0} = \text{initialize}$

$\text{for } p \in 0, \ldots$

$\quad \# \text{ Evaluation step}$

$\quad \text{for } k \in 0, \ldots$

$\quad\quad v_{\pi,k+1}(s_t) = \text{update from } v_{\pi,k}$

$\quad\quad \text{break when } \left|v_{\pi,k+1} - v_{\pi,k}\right| < \epsilon_1$

$\quad \# \text{ Policy improvement step}$

$\quad \pi_{p+1} = \text{ update from } \pi_p$

$\quad \text{break when } \pi_{p+1} \approx \pi_p$

Iterative improvement of $v_{\pi,k}$ to $v_{\pi,k+1}$ is via the equation

$$v_{\pi,k+1}(s_t) \quad = \quad \sum_{s',r} P(s', r|s, \pi(s)) * (r + \gamma v_{\pi,k}(s')) \quad \text{Expectation of return a environment responses given that agent's actio}$$

We continue iterating (increasing k) until, for all states s,

- the difference between $v_{\pi,k+1}(s)$ and $v_{\pi,k}(s)$ is smaller than a threshold value.

Given the Evaluation of the current policy v_{π_p} we improve π_p to π_{p+1} with the Policy improvement step

$$\pi'_{p+1}(s) \quad = \quad \underset{a}{\operatorname{argmax}} \sum_{s',r} P(s', r|s, a)(r + \gamma v_\pi(s')) \quad \text{chose value maximizing}$$

That is: the agent in state s chooses the action with maximal return.

We can continue with another round $(p+1)$ which again alternates Evaluation and Policy Improvement until no change in policy results

$$\pi_{p+1} = \pi_p$$

Value-based algorithm 2: Value iteration

One issue with Policy iteration is the potentially slow convergence of the Evaluation of the current policy (performed by iterative approximation).

Convergence can be improved by changing the Iterative improvement of $v_{\pi,k}$ to $v_{\pi,k+1}$ to

$$v_{\pi,k+1}(s_t) \quad = \quad \max_a \sum_{s',r} P(s', r|s, a)(r + \gamma v_\pi(s'))$$

Observe that this Evaluation improvement step is identical to the Policy improvement step

- except for the change of argmax to max
- which is necessary since
    - v needs the maximum value
    - π needs the *action* associated with the maximum value

Sutton (http://incompleteideas.net/book/RLbook2020.pdf#page=105)

Value-based algorithm 3: Q-learning

Q-learning is a form of updating the Action-Value function.

It implements the q_π function (mapping state/action pairs to return) via a *tabular* lookup

- table is built dynamically through experience

[See algo in HF (https://huggingface.co/deep-rl-course/unit2/q-learning?fw=pt)](https://huggingface.co/deep-rl-course/unit2/q-learning?fw=pt)

Here is some pseudo-code dervied from [HuggingFace Unit 2 (https://huggingface.co/deep-rl-course/unit2/hands-on?fw=pt#hands-on)]

- We will build the action-value function
$$q_\pi : s \times a \to \mathbb{R}$$
as a table `Qtable`
- We will `Qtable` by exploring multiple episodes (max `n_training_episodes`)
- In each epsiode, we will create a sequence of steps(max: `max_steps`)
- In each step,
    - Given that we are in state S_t
    - we will select an action A_t by choosing the "best" action from the current `Qtable`
- We will update our estimate of $q_\pi(S_t, A_t)$
    - given the reward R_{t+1} received as a result of the chosen action

The obvious choice for action A_t is the one with

$$\max_{a'} q_\pi(s_t, a')$$

This choice is called a *greedy policy*

The problem with greedy policies is that, initially, our estimate of the true q_π is inaccurate.

- by choosing the current estimate of "optimal" action
- we may fail to ever choose the true optimal
- and we will never learn the optimal action as a result

Choosing the current "best" is called *exploitation*.

Sometimes *exploration* (making a seemingly sub-optimal choice) sacrifices short term gain for long term gain.

This is called the *exploration-exploitation* trade-off.

We will encourage exploration by choosing an action using an *ϵ-greedy* policy

- with probability $(1 - \epsilon)$: choose $\max_{a'} q_\pi(s_t, a')$
- with probability ϵ: choose a random action

In [ ]:

```
Qtable = initialize_q_table(state_space, action_space)

for episode in range(n_training_episodes):
    # Reduce epsilon (because we need less and less exploration)
    epsilon = min_epsilon + (max_epsilon - min_epsilon)*np.exp(-decay_rate*episo
de)

    # Reset the environment
    state = env.reset()
    step = 0
    done = False

    # repeat
    for step in range(max_steps):
      # Choose the action At using epsilon greedy policy
      action = epsilon_greedy_policy(Qtable, state, epsilon)

      # Take action At and observe Rt+1 and St+1
      # Take the action (a) and observe the outcome state(s') and reward (r)
      new_state, reward, done, info = env.step(action)

      # Update Q(s,a):= Q(s,a) + lr [R(s,a) + gamma * max Q(s',a') - Q(s,a)]
      Qtable[state][action] = Qtable[state][action] + learning_rate * ( reward +
gamma*np.max( Qtable[new_state][:])
        - Qtable[state][action] )

      # If done, finish the episode
      if done:
        break

      # Our next state is the new state
      state = new_state
```

On-policy versus Off-policy

Notice a subtle difference between choices made in

- the statement that chooses the action action A_t
- the statement that updates $q_\pi(S_t, A_t)$
- the update of $q_\pi(S_t, A_t)$

The **action choice** A_t chosen was made via an *epsilon-greedy* choice

```
action = epsilon_greedy_policy(Qtable, state, epsilon)
```

But the **update choice** assumes that all future choices are *greedy* by choosing the **max** over all actions

- the sub-statement

  ```
  np.max( Qtable[new_state][:])
  ```

  in

```
Qtable[state][action] =
 Qtable[state][action] + learning_rate *
    ( reward + gamma*np.max( Qtable[new_state][:]) - Qtable[state][action] )
```

When the action choice and the update choice are the same

- we call the algorithm *on-policy*

When the action choice differs from the update choice

- we call the algorithm *off-policy*

Thus: Q-learning is Off-Policy

Off-policy is a subtle [distributional shift (https://youtu.be/k08N5a0gG0A?t=1101)]

- write

$$q_\pi(s,a) = R_{t+1} + \max_{a'} q_\pi(s_{t+1},a') \quad \text{immediate reward } R_{t+1}$$

plus discounted future reward
choosing best action a' in new

$$= R_{t+1} + \mathbb{E}_{a' \in \pi'}\left(q_\pi(s_{t+1},a')\right) \quad \text{where } \pi' \text{ is strategy}$$

$$\pi'(a|s_t) = \operatorname*{argmax}_{a'} q_\pi(s_{t+1},a')$$

Update uses distribution of actions generated by π' (i.e., greedy): all probability at o

Action choice uses the distribution that mixes the distribution generated by π' with parameter ϵ

  - the distribution generated by π' is an *adversarial* distribution in that the "er maximized

Value-based algorithm 3: Deep Q-learning

The method for Q-learning presented involved creating a *table* implementing the mapping q_π.

This is only practical when the size of the table is small

- the number of states for many problems (e.g., games) is extremely large
- not practical

Deep Q-Learning

- treats q_π as a function
- which is approximated by a Neural Network (the *Deep Q-Network (DQN)*

[See algo in HF (https://huggingface.co/deep-rl-course/unit3/deep-q-algorithm?fw=pt)](https://huggingface.co/deep-rl-course/unit3/deep-q-algorithm?fw=pt)

The basic Q-learning algorithm must be adapted.

- The `Qtable` is replaced by a Neural Network that creates function Q_θ that maps a state S_t and an action A_t to a Real.
- The NN is trained by creating examples
    - Examples are created in each step t of the episode
    - The features of the example are the state S_t and action A_t
    - The target of the example of step t is identical to the original algorithm's updated value for $q_\pi(S_t, A_t)$

That is:

- We train the NN function $Q_\theta(S_t, A_t)$ to approximate true value $q_\pi(S_t, A_t)$
- Using an MSE per-example loss

$$(Q_\theta(S_t, A_t) - q_\pi(S_t, A_t))^2$$

After creating examples

- We sample a mini-batch of examples _ Minimize the average (across examples) loss
- By Gradient Descent

One subtlety:

- The target for an example is recursively defined. See the term on the RHS
  - $\max_{a'}(q_\pi(S_t, a'))$
  - implemented as

    ```
    np.max( Qtable[new_state][:] )
    ```

    in the original algorithm
  - the terms (in the max) $q_\pi(S_t, A')$ are approximated by the NN
    - as $Q_{\theta^-}(S_t, A')$
    - where θ^- is a lagged value of the weights
    - rather than current value θ

If we don't lag the weights: the targets computed for other mini-batches will be based on different weights

- so we have a moving target as well as a moving function

The lagged weights are periodically synchronized with the most recent weights.

Value-based method: Credit assignment (implied intermediate rewards)

In many episodes,

$$\dots S_t, A_t, R_{t+1}, \dots$$

the only reward comes from entering the terminal state, thus

$$R_{t+1} = 0$$

for many time steps t.

With v_π in-hand

- we can interpret the *increment* in value
  $$v_\pi(S_{t+1}) - v_\pi(S_t)$$
  of the action that takes us from S_t to S_{t+1}
- as an implicit reward that provides immediate feedback

Value-based methods: draw-backs

- Policy is deterministic; can't have stochastic policy

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, q(s, a)$$

- actions are discrete, not continuous
  - the magnitude of angles (when turning) or velocity (when moving) are not continuous
  - a consequence of the $\max_{a'}$

Policy-based methods

Value-based methods define a policy on the basis of the value assigned to a state or state/action pair.

Policy-based methods, by contracts, constructs the policy (a function) directly.

We are concerned with *Deep Reinforcement Learning*

- the *policy* is a parameterized (by θ) function mapping states S to (a probability distribution) actions

$$\pi_\theta(A|S)$$

Policy-based methods are *necessary*

- when actions are continuous rather than discrete

Policy-based methods are *desirable* (even if not necessary)

- when there are a *large* number of discrete actions
  - algorithms that involve $\max$ over actions

$$\pi^*(s) = \operatorname*{argmax}_a q(s, a)$$

or policy-iteration update

$$\pi'_{p+1}(s) = \operatorname*{argmax}_a \sum_{s',r} P(s', r|s, a)(r + \gamma v_\pi(s'))$$

Policy Gradient

There are various methods (e.g., search) for finding the optimal π_θ^*.

Of particular interest to us are methods that use Gradients to improve π

That is, we create a sequence of improving policies

$$\pi_0, \ldots, \pi_p, \ldots$$

by creating a sequence of improved parameter estimates

$$\theta_0, \ldots, \theta_p, \ldots$$

using Gradient Ascent on some objective function $J(\theta)$ to improve θ_p

$$\theta_{p+1} = \theta_p + \alpha * \nabla_\theta J(\theta_p)$$

Since we are trying to maximize objective function $J(\theta)$ rather than minimize a loss objective

- we use Gradient Ascent rather than Gradient Descent
- hence we add the gradient rather than subtract it, in the update

RL Book Chapt 12 (http://incompleteideas.net/book/RLbook2020.pdf#page=343)

Stochastic policy and environment

With the policy gradient method: the policy can be stochastic (action is a probability distribution)

$$\pi(a|s;\theta) = p(A_t = a|S_t = s, \theta_t = \theta)$$

The environment can *also* be stochastic

$$P(s', r|s, a) = P(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a)$$

- the response (s', r) by the environment is not deterministic

This poses a challenge to Value-based methods

- a single observation of (s', r) is a *high variance* estimate of $P(s', r|s, a)$

Objective function

Recall that the return G_t of a single episode is the expected value of rewards accumulated starting in the state of step t of the episode

$$\begin{aligned} G_t &= \sum_{k=0}^{t} \gamma^k * R_{t+k+1} \\ &= r_{t+1} + \gamma * G_{t+1} \end{aligned}$$

The performance measure $J(\theta)$ that we define will be the *expected value* (across all possible episodes) of the return G_0 from initial state S_0 of the episode

$$J(\theta) = \mathop{\mathbb{E}}_{\tau}(G_{0,\tau})$$

- using the notation

  $$G_{t,\tau}$$

  to denote the return within episode τ of step t of the episode.

Note that $G_{t,\tau}$ is equivalent to $v_\pi(S_t)$ (relative to episode τ)

- the value function evaluated on the initial state

Taking the gradient of the Objective

This objective function presents some challenges in computing $\nabla_\theta J(\theta_p)$

The first is: how to take gradient of an expectation ?

We can do away with the expectation by replacing it with the sum

$$\mathop{\mathbb{E}}_{\tau}(G_{t,\tau}) = \sum_{\tau} p(\tau;\theta) * G_{t,\tau}$$

where

$$p(\tau;\theta)$$

is the probability of episode τ.

In practical terms

- we don't sum over every possible episode
- we can approximate the Expectation through *trajectory sampling*
    - accumulate a batch of episodes
    - approximate the expectation as the average across the episodes in the batch

Note that the gradient of a sum is equal to the sum of the gradients

- so being able to compute the gradient of $J(\theta)$ depends on being able to compute the terms in the sum.

But this too presents a challenge

The probability of episode τ occurring is thus the product of each step occurring $$ \begin{array} \ \pr{\tau; \theta} & = & \prod*{t=0} { *transp({ *state'*, *rew* | *state* = *stateseq*\tt, \act = \actseq_\tt })

```
 * \pi( \actseq_\tt | \stateseq_\tt )
  }
```

\end{array} $$

The problem is the Transition Probability term in the product

$$P(s', r|s = S_t, a = A_t)$$

- the reaction of the Environment to the agent choosing action A_t in state S_t
- is controlled by the environment
- generally: unknown

The Policy-Gradient Theorem

Fortunately, the *Policy Gradient Theorem* provides a solution to the challenges

$$\nabla_\theta J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) * \nabla_\theta \pi_\theta(a|s;\theta)$$

where $\mu(s)$ is the probability of being in state s

- derived from counting the number of times state s is encountered across all episodes

Note that

$$\pi_\theta(a|s;\theta)$$

- is the output of the NN (parameterized by θ)
- computing a probability distribution (over actions) $\pi_\theta(A_t|S_t;\theta)$ on input S_t

Note that the RHS is

- the weighted (by $\mu(s)$, the probability of state s) sum
- of a term ($\sum_a \ldots$) that varies over s

which is the definition of an Expectation of the term over states.

Thus we can re-write $\nabla_\theta J(\theta)$:

$$
\begin{aligned}
\nabla_\theta J(\theta) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) * \nabla_\theta \pi_\theta(a|S;\theta) \\
&= \mathop{\mathbb{E}}_{\pi} \left(\sum_a q_\pi(S_t,a) * \nabla_\theta \pi_\theta(a|S_t;\theta) \right) \qquad \text{replace } s \text{ with the sample } S
\end{aligned}
$$

We can also convert the sub-term $$ \sum*act* { *actvalfun*\pi(\stateseq_\tt, \act)

```
        *
        \nabla_\theta {
            \pi_\theta( \act | \stateseq_\tt; \theta)
        }
}
```

$$ into an expectation of $ \nabla*theta* { *pi*\theta(\act | \stateseq_\tt; \theta) } $ by a little algebra that makes the summand weighted by its probability

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathop{\mathbb{E}}_{\pi}\left(\sum_a q_\pi(S_t, a) * \nabla_\theta \pi_\theta(a|S_t;\theta)\right) \\
&= \mathop{\mathbb{E}}_{\pi}\left(\sum_a \frac{\pi_\theta(a|S_t;\theta)}{\pi_\theta(a|S_t;\theta)} * q_\pi(S_t, a) * \nabla_\theta \pi_\theta(a|S_t;\theta)\right) && \text{since } \frac{\pi(a|S_t;\theta)}{\pi(a|S_t;\theta)} = 1 \\
&= \mathop{\mathbb{E}}_{\pi}\left(\sum_a \pi(a|S_t;\theta) * q_\pi(S_t, a) * \frac{\nabla_\theta \pi_\theta(a|S_t;\theta)}{\pi_\theta(a|S_t;\theta)}\right) && \text{rearranging terms} \\
&= \mathop{\mathbb{E}}_{\pi}\left(q_\pi(S_t, A_t) * \frac{\nabla_\theta \pi_\theta(A_t|S_t;\theta)}{\pi_\theta(A_t|S_t;\theta)}\right) && \text{replace } \sum_a \pi_\theta(a|S_t \\
& && \text{with the Expectati}
\end{aligned}
$$

We often see the last term re-written $$ \begin{array} \ \nabla*theta { J(\theta) }* & = & *Exp{\pi}* *left(*actvalfun*\pi(\stateseq*tt*, *actseq*\tt)

```
                *
                \nabla_\theta {
                \log{\pi_\theta( \actseq_\tt | \stateseq_\tt; \theta)}
                }
        \right)
```

& \text{ since } \nabla*theta { \log(x) }* = *frac{\nabla*\theta { x }}{x} \text{ by derivative of log} \ \end{array} $$

In practical terms, we approximate the gradient by sampling to evaluate the expectations $$ \begin{array}{l} \nabla*theta { J(\theta) } & = & *frac{1}{m}* *sum*{i=1}^m \sum*{*tt=1}^T { *actvalfun*\pi(\stateseq*tt*, *actseq*\tt)

```
*
\nabla_\theta {
\log{\pi_\theta( \actseq_\tt | \stateseq_\tt; \theta)}
} }
```

& \text{sample } m \text{ episodes} \\ \end{array} $$

- the frequency of each term over the sample is an approximation of the true probability

Notes

- We approximate the Q-function $q_\pi(S_t, A_t)$ by the G_t of step t of episode τ

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathop{\mathbb{E}}_{\pi}\left(q_\pi(S_t, A_t) * \nabla_\theta \log \pi_\theta(A_t|S_t;\theta)\right) \\ &= \mathop{\mathbb{E}}_{\pi}\left(G_{t.\tau} * \nabla_\theta \log \pi_\theta(A_t|S_t;\theta)\right) \qquad \text{since } q_\pi(S_t, A_t) = \mathbb{E}(G_{t,\tau}|S_t\end{aligned}$$

- It seems strange to compute the Q-function in a **policy**-based method
  - Would be expected in a **value-based** method
  - But we don't use the Q-function to determine the policy
  - Just for the objective

- $\nabla_\theta { J(\theta) } = \Exp{\pi} \left(

```
        G_{\tt. \tau}
        *
        \nabla_\theta {
        \log{\pi_\theta( \actseq_\tt | \stateseq_\tt; \theta)}
        }
\right)
```

  $ is the amount by which we change the weights

  - in the *direction* $\nabla_\theta {

    ```
    \log{\pi_\theta( \actseq_\tt | \stateseq_\tt; \theta)}
    } $
    ```

    that increase probability of action A_t

  - proportional to $G_{t,\tau}$:

    - increase is greater for steps with high positive return
    - decrease for steps with negative return
  - this "reinforces" the choice of actions leading to favorable trajectories

See

- the Sutton book (http://incompleteideas.net/book/RLbook2020.pdf#page=348) for a good explanation
- the Lilian Weng blog (https://lilianweng.github.io/posts/2018-04-08-policy-gradient/) for a clear proof of the Theorem

Policy-based algorithm 1: REINFORCE

REINFORCE is an example of a policy-based method that uses a NN computing $\pi_\theta(A_t|S_t;\theta)$

We update θ_p to θ_{p+1} by

- Creating episode τ, which is episode number $p+1$

$$\tau = S_0, A_0, R_1, \ldots S_t, A_t, R_{t+1}, \ldots S_{T-1}, A_{T-1} R_T$$

Policy-based algorithm 1': REINFORCE with a baseline

In the basic REINFORCE: the gradient for step t of episode τ is computed as $$ G_{\tt. \tau}

```
*
\nabla_\theta {
\log{\pi_\theta( \actseq_\tt | \stateseq_\tt; \theta)}
}
```

$$Ratherthanusing\$G_{t,\tau}\$, wecansubtracta * baseline * \$b(S_t)\$ from \$G_{t,\tau}$$

$$\$leadingtogradient$$

```
\left(G_{\tt. \tau} - b(\stateseq_\tt) \right)
*
\nabla_\theta {
\log{\pi_\theta( \actseq_\tt | \stateseq_\tt; \theta)}
}
```

$$

Regardless of how we define the baseline $b(S_t)$

- This results in a *lower variance* estimate of $\nabla_\theta J(\theta)$
- without affecting changing its value

The Policy Gradient Theorem tells us $$ \begin{array}{l} \nabla_\theta J(\theta) \propto \sum_{s} \mu(s) \sum_{a} q_\pi(s, a) \nabla_\theta \pi(s, a)

```
                    *
                    \nabla_\theta {
                        \pi_\theta( \act | \state; \theta)
                    }
 }
```

\end{array} $$

Subtracting baseline $b(S_t)$ from $q_\pi(s, a)$ transforms the RHS to $$ \sum_{s} \mu(s) \sum_{a} \left(q_\pi(s, a) - b(s) \right) \nabla_\theta \pi(s, a)

```
                    *
                    \nabla_\theta {
                        \pi_\theta( \act | \state; \theta)
                    }
 }

}
```

$$
\begin{aligned}
& \text{and the sum over actions of the subtracted value} \\
\sum_a b(s) * \nabla_\theta \pi_\theta(a|s;\theta) &= b(s) * \sum_a \nabla_\theta \pi_\theta(a|s;\theta) && \text{since } b(s) \text{ is not a functio} \\
&= b(s) * \nabla_\theta \sum_a \pi_\theta(a|s;\theta) && \text{since the sum of gradient} \\
&= b(s) * \nabla_\theta 1 && \text{since } \sum_a \pi_\theta(a|s;\theta) = 1 \\
&= 0 && \text{since } \nabla_\theta 1 = 0
\end{aligned}
$$

$$

Thus, subtracting the baseline does not affect the outcome of the calculation of the gradient.

Actor-Critic

Value-based methods learn a function approximation of the *value* of a state or a state/action pair.

- policy is chosen based on the value of successor states

Simple Policy-based methods learn a parameterized policy function.

- using a NN to learn the policy
- using an objective function $J(\theta)$ that depends on an approximation of either
    - the value $v(s)$ or G_t
    - or action/value function $q(s, a)$

Actor-Critic-Policy-based methods used Neural Networks to learn

- *both* the value function and policy function approximations
- the agent is called the *Actor*
- the NN providing estimates of G_t or $q(s, a)$ is called the *Critic*

Notice that, in the REINFORCE algorithm, $G_{t,\tau}$ is computed for *each trajectory* τ independently

- there is no memory of the prior stochastic response
  $P(s', r|s, a) = P(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a)$
  for the same state s and action a of a previous episode
    - this leads to high variance estimates of $G_{t,\tau}$

By using a NN to estimate G_t

- our estimate includes multiple examples of the stochastic response to action a in state s
- hopefully leading to a lower variance approximation

- RL tips and tricks (https://stable-baselines3.readthedocs.io/en/master/guide/rl_tips.html)
- RL book contents (http://incompleteideas.net/book/RLbook2020.pdf#page=7)
- RL book notation (http://incompleteideas.net/book/RLbook2020.pdf#page=20)

Policy-basd algorithm 2: Actor-Critic REINFORCE

This algorithm is REINFORCE with a baseline

- where the baseline is the value function $v(s)$

The gradient used for the update becomes

$$\begin{aligned} & (G_{t.\tau} - b(S_t)) * \nabla_\theta \log \pi_\theta(A_t|S_t;\theta) && \text{definition of gradient with baselin} \\ = \quad & (G_{t.\tau} - v_w(S_t)) * \nabla_\theta \log \pi_\theta(A_t|S_t;\theta) && \text{baseline defined as value function} \end{aligned}$$

We used the subscript w in v_w to indicate that v is computed by a NN parameterized by w.

Parameters θ (for the actor's policy) and w (for the critic) are updated during each iteration of REINFORCE.

- the updates to v_w used Temporal Differences
    - just as we did for Value-based methods
    - update to w is
        - learning rate for w (not necessarily the same as learning rate for θ
        - times ∇_w
        - times temporal difference term
          $\delta \quad =$

          $$\begin{array}{rll} & R_{t+1} + \gamma * v_w(S_{t+1}) & \text{updated value for } v_w(S \\ - & v_w(S_t) & \text{baseline equals } v_w(S_t) \\ & & \text{which is also the old va} \end{array}$$
        - the baseline was chosen to create the temporal difference

OSOLETE: Derivation of Policy Gradient Theorem

$$\nabla_\theta J(\theta) \quad = \quad \mathop{\mathbb{E}}_{\pi} \sum_a q_\pi(s_t, a) * \nabla_\theta \pi_\theta(a|s;\theta)$$

- run multiple episodes

The probability of step t in τ occurring is the product of $$ \begin{array} \ \pi(\actseq*tt* | *stateseq*\tt) & \text{prob. agent chooses action } \actseq_\tt \

- \ \transp({ \state', \rew | \state = \stateseq*tt*, *act* = *actseq*\tt }) & \text{prob. environment transitions to } \state' \text{ in response }\ \end{array} $$

The probability of episode τ occurring is thus the product of each step occurring $$ \begin{array} \ \pr{\tau; \theta} & = & \prod*{t=0}{ *transp*({ *state'*, *rew* | *state* = *stateseq*\tt, \act = \actseq_\tt })

```
* \pi( \actseq_\tt | \stateseq_\tt )
 }
```

\end{array} $$

- the probability of an episode (used to calculate the expectation over τ)
    - the term involving the probability of trajectory τ

Policy gradient theorem

Let us consider an episode $\tau = S_0, A_0, R_1, \ldots S_t, A_t, R_{t+1}, \ldots$.

The probability of step t in τ occurring is the product of $$ \begin{array} \ \pi(\actseq*tt* | *stateseq*\tt) & \text{prob. agent chooses action } \actseq_\tt \

- \ \transp({ \state', \rew | \state = \stateseq*tt*, *act* = *actseq*\tt }) & \text{prob. environment transitions to } \state' \text{ in response }\ \end{array} $$

The probability of episode τ occurring is thus the product of each step occurring $$ \begin{array} \ \pr{\tau; \theta} & = & \prod*{t=0}{ \transp({ \state', \rew | \state = \stateseq*\tt, \act = \actseq_\tt })

```
* \pi( \actseq_\tt | \stateseq_\tt )
 }
```

\end{array} $$

Notes

This objective function presents some challenges in computing $\nabla_\theta J(\theta_p)$

- how to take gradient of an expectation
- the term involving the probability of trajectory τ

$$p(\tau;\theta) \quad = \quad \prod_{t=0} P(s', r|s = S_t, a = A_t) * \pi(A_t|S_t)$$

The Transition Probability $P(s', r|s = S_t, a = A_t)$

- controlled by the environment
- generally: unknown

See here for derivation from HF (https://huggingface.co/deep-rl-course/unit4/pg-theorem?fw=pt)

See here for derivation from RL book (http://incompleteideas.net/book/RLbook2020.pdf#page=343)

Lilian Weng blog (https://lilianweng.github.io/posts/2018-04-08-policy-gradient/)

First, we can do away with the expectation by replacing it with the sum

$$\mathop{\mathbb{E}}_{\tau}(G_{t,\tau}) = \sum_{\tau} p(\tau;\theta) * G_{t,\tau}$$

In practical terms

- we don't sum over every possible episode
- we can approximate the Expectation through *trajectory sampling*
    - accumulate a batch of episodes
    - approximate the expectation as the average across the episodes in the batch

Note that we are generalized to $G_{t,\tau}$ rather than the specific $G_{0,\tau}$ that we need

- this will come in useful later

$$
\begin{aligned}
\nabla_\theta \mathop{\mathbb{E}}_{\tau}(G_{t,\tau}) &= \nabla_\theta \textstyle\sum_\tau p(\tau;\theta) * G_{t,\tau} && \text{definition of expectation} \\
&= \textstyle\sum_\tau \nabla_\theta p(\tau;\theta) * G_{t,\tau} && \text{gradient of sum is sum of gradients}
\end{aligned}
$$

- To simplify
- rather than carrying along the expectation over all episodes τ
- we compute the gradient for a single τ
- and show how to work in the expectation

To generalize the result , we will use $G_{t,\tau}$ rather than $G_{0,\tau}$

$\nabla_\theta p(\tau;\theta) * G_{t,\tau}$	$=$
$\nabla_\theta \prod_{t=0} P(s', r \vert s = S_t, a = A_t) * \pi(A_t \vert S_t) * G_{t,\tau}$	expand definition of $p(\tau;\theta)$
	$=$

$$\nabla_\theta p(\tau;\theta) * G_{t,\tau} \quad = \quad p(\tau;\theta)\nabla_\theta G_{t,\tau} + G_{t,\tau} * \nabla_\theta p(\tau;\theta) \quad \text{derivative of a product}$$

Let's handle the sum in parts.

For the first part (we will multiply by $G_{0,\tau}$ at the end)

$$\begin{aligned}
\nabla_\theta p(\tau;\theta) &= \frac{p(\tau;\theta)}{p(\tau;\theta)} * \nabla_\theta p(\tau;\theta) && \text{fraction equals 1} \\
&= p(\tau;\theta)\nabla_\theta \log p(\tau;\theta) && \text{since } \nabla_\theta \log p(\tau;\theta) = \frac{\nabla_\theta p(\tau;\theta)}{p(\tau;\theta)} \\
& && \text{by derivative of } \nabla_\theta \log x = \frac{\nabla x}{x}
\end{aligned}$$

We can expand the $p(\tau;\theta)$ term in the log to the product $$ \pr{\tau; \theta} = \prod*{t=0}* { *transp({ *state'*, *rew* | *state* = *stateseq*\tt, \act = \actseq_\tt })

```
* \pi( \actseq_\tt | \stateseq_\tt )
 }
```

$$
\begin{aligned}
& and convert the log of a product into the sum of logs. \\
\nabla_\theta p(\tau;\theta) &= p(\tau;\theta)\nabla_\theta \log p(\tau;\theta) && \text{copy} \\
&= p(\tau;\theta) * \nabla_\theta \textstyle\sum_{t=0} \log\left(P(s',r|s=S_t,a=A_t) * \pi(A_t|S_t)\right) && \log \\
&= p(\tau;\theta) * \textstyle\sum_{t=0} \nabla_\theta \log\left(P(s',r|s=S_t,a=A_t) * \pi(A_t|S_t)\right) && \text{deri}
\end{aligned}
$$

$$