Reinforcement Learning introduction for AI safety

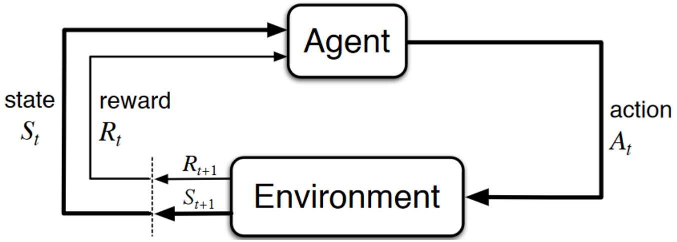
Aude Maier

Swiss AI Safety Camp 2023

Program

- Introduction to Markov Decision Processes
- TD-learning
  - Tabular Q-learning
  - Deep Q-learning
  - Coding exercise on deep Q-learning
- Policy gradient
  - Vanilla Policy Gradient
  - Coding exercise on Vanilla Policy Gradient
  - Policy Gradient with baseline
  - Actor Critic architecture
  - Proximal Policy Optimization
- Reinforcement Learning from Human Feedback

Markov Decision Process (MDP)



- S_t, A_t, R_t are Random Variables;
  $S_t \in \mathcal{S}$, $A_t \in \mathcal{A}$, $R_t \in \mathbb{R}$.
- The interaction between Agent and Environment creates the **Trajectory**: $S_0, A_0, R_0, S_1, A_1, R_1, \ldots$
- R_t, S_t only depend on S_{t-1}, A_{t-1} (Markov Property), they are distributed according to the **Dynamic** of the MDP:

$$p(s', r|s, a) := Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$$

Markov Decision Process (MDP)

Definitions

- **Return**:
$$G_t := R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \dots$$
What we want to maximize (in expected value).
- **Policy**:
$$\pi(a|s) := Pr\{A_t = a | S_t = s\}$$
Rule according to which the agent selects an action. "Making the agent learn" means modifying its policy in order to maximize the expected return.
- **V-values**:
$$V_\pi(s) := \mathbb{E}_\pi[G_t | S_t = s]$$
- **Q-values**:
$$Q_\pi(s, a) := \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$
Property - $V_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) Q_\pi(s, a)$

Markov Decision Process (MDP)

Commonly used policies

- Greedy: $\pi(s,a) = \mathbb{1}\{a = \mathrm{argmax}_a Q_(s,a)\}$ (deterministic)
- Epsilon-greedy: follow the greedy policy with probability $0 < \varepsilon < 1$ and choose a random action with probability ε (stochastic)
- Softmax: $\pi(s,a) = \frac{\exp(\beta Q(s,a))}{\sum_{\tilde{a}} \exp(\beta Q(s,\tilde{a}))}$ (stochastic)

Markov Decision Process (MDP)

- **Optimal policy**: a policy π_* is optimal if $V_{\pi_*}(s) \geq V_\pi(s)$ for any state s and policy π

Theorem

There always exists at least one optimal policy.

- **Optimal V/Q-values**: $v_*(s) := v_{\pi_*}(s) = max_\pi V_\pi(s)$, $q_*(s,a) := q_{\pi_*}(s,a) = max_\pi q_\pi(s,a)$

Exercises - Build the Bellman Equation

1. Write G_t as a function of G_{t+1}.
2. Write a recursive equation for the V-values $v_\pi(s)$. Hint: plug your result to the previous exercise in the definition of the V-values.
3. Do the same for the Q-values.

Model free / Model based RL

- **Model based**: The agent has an explicit representation of $p(s', r|s, a)$, allows explicit planning
- **Model free**: No explicit representation of $p(s', r|s, a)$, learns heuristics (e.g. taking action a in state s usually leads to high reward later in the episode), learns directly v/q-values or policy

Model based Reinforcement Learning (optional

- Given the model:
  Optimization problem - can be solved optimally by dynamic programming methods, not always computationally tractable therefore we use decision time planning methods such as monte carlo tree search. This is the case of AlphaZero
- Learns the model:
  Approximate the dynamic and learn the policy / v-values using the learned dynamic. For a complex game such as go, use a latent space to represent states. This is the case of MuZero

Model free Reinforcement Learning

- **Temporal Difference (TD)-learning**
  First learn the optimal Q-values using the Bellman equation. Then find the optimal policy by taking $\pi_*(a|s) = \mathbb{1}\{a = \mathrm{argmax}_a Q_*(s, a)\}$.
- Policy learning
  Learn directly the policy $\pi(a|s)$ such as to maximize the expected return.

TD-learning

Bellman Optimality Equation

$$\begin{aligned} q_*(s,a) &= \mathbb{E}[R_t + \gamma \mathrm{max}_{a'} Q_*(S_{t+1}, a') | S_t = s, A_t = a] \\ &= \sum_{r,s'} p(r, s' | s, a) \left(r + \gamma \mathrm{max}_{a'} Q_*(s', a') \right) \end{aligned}$$

Theorem

1. There always exists a unique set of Q-values $\{Q(s,a)\}_{s \in \mathcal{S}, a \in \mathcal{A}}$ that satisfy the Bellman Optimality Equation;
2. The Q-values that satisfy the Bellman Optimality Equation are the optimal Q-values.

In other words - A set of Q-values $\{Q(s,a)\}_{s \in \mathcal{S}, a \in \mathcal{A}}$ satisfy the Bellman Optimality Equation if and only if they are the optimal Q-values.

TD-learning

Idea

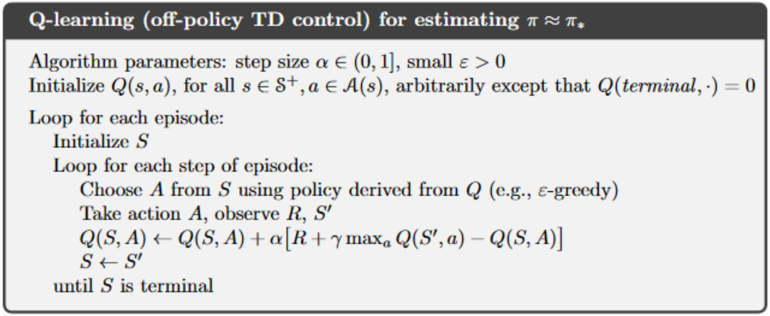
Update the Q-values iteratively to minimze the difference between the RHS and LHS of the Bellman equation. If the iteration converges, it means that we have found the optimal Q-values.

Update Rule

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_t + \gamma \mathrm{max}_a Q(S_{t+1}, a) - Q(S_t, A_t))$$

TD-learning

Algorithm



Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Exercise tabular Q-learning

Consider a rat navigating in a 1-armed maze (=linear track). The rat is initially placed at the upper end of the maze (state s), with a food reward at the other end. This can be modeled as a one-dimensional sequence of states with a unique reward ($r = 1$) as the goal is reached. For each state, the possible actions are going up or going down. When the goal is reached, the trial is over, and the rat is picked up by the experimentalist and placed back in the initial position s and the exploration starts again.

- Describe what happens during the first episode.
- Compute the Q-values at the end of the first episode.
- Compute the Q-values at the end of the second episode.
- Give the Q-values that will reached after an arbitrarily large number of episodes.

Speed up the propagation of the values - Optional

- N-step TD-learning - Write the Q-values $Q(S_t, A_t)$ as a function of $Q(S_{t+n}, A_{t+n})$.
- Eligibility traces - associate a variable to every Q-value, called its trace, which represents the contribution of the corresponding state-action pair to the reward obtained.

TD-learning with function approximation: deep Q-learning

Problems of tabular Q-learning

- Many entries
- Cannot handle continuous spaces
- Does not take into account the proximity between states

Idea

Define a neural network that takes a continuous state as input and returns the Q-value of every action.
$Q(s,a)$ becomes $Q(s,a|\vec{\theta})$ where $\vec{\theta}$ are the parameters of the Q-network.

Question

Consider a continuous version of the previous exercise, what would be the input and output dimension of the Q-network?
Same question for a 2D version of the exercise.

Deep Q-learning - training process

Bellman Equation reminder

On average, we want $Q(S_t, A_t) = R_t + \gamma \mathrm{max}_a Q(S_{t+1}, a)$.

- Use a mean squared error between the RHS and LHS.
  - Loss function:
    $E = \frac{1}{2} \sum_t \left[R_t + \gamma \mathrm{max}_a Q(S_{t+1}, a|\vec{\theta}) - Q(S_t, A_t|\vec{\theta}) \right]^2$
  - Update rule: $\theta_i = \theta_t + \frac{\partial E}{\partial \theta_i}$

Replay Buffer

How to choose the batch on which you compute the error function?

- First option - let the agent evolve during b steps (b is the batch size) and use these b steps to compute E.
  Problem: the steps aren't at all independent, can cause instabilities.
- Alternative - Store e.g. the 10 000 last steps in a replay buffer, sample the batches from the replay buffer.
  Subtelty - you revisit steps that have been generated with an old network, it does not make sense to update the current network based on mistakes done by an older version. Hence, need to compute the action A_{t+1} that would be taken by the current network.

Target Network

$R_t + \gamma \max_a Q(S_{t+1}, a|\vec{\theta})$ is called the target Q-value.
To make the learning more stable, use a different network to compute the Q-value in the target

$$E = \frac{1}{2} \sum_t \left[R_t + \gamma \max_a Q(S_{t+1}, a|\vec{\theta}^{target}) - Q(S_t, A_t|\vec{\theta}) \right]^2$$

Update of target network

The target networks use "soft updates", meaning that their weights are updated with a weighted average of the current Q-network weights and the current target network weights.

$$\theta^{target} \leftarrow \tau\theta + (1 - \tau)\theta^{target}$$

Deep Q-learning pseudocode

```
Algorithm 1 Deep Q-Learning with Replay Buffer
Input: Environment $p$
 1: Initialise replay buffer $\mathcal{D}$ to capacity $N$
 2: Initialize state-action value network $Q(\cdot\,;\theta)$ with parameters $\theta$
 3: for episode = 1 to $M$ do
 4:     Sample initial state $s$ from environment
 5:     $d :=$ False
 6:     Initalize target parameters $\theta_{\text{target}} := \theta$
 7:     while $d =$ False do
 8:         $a := \begin{cases} \arg\max_a Q(s,a;\theta) & \text{prob } 1-\epsilon \\ \text{random action} & \text{prob } \epsilon \end{cases}$
 9:         Sample $(s_{\text{new}}, r, d)$ from environment given $(s, a)$.
10:         Store experience $(s, a, r, s_{\text{new}}, d)$ in $\mathcal{D}$.
11:         if Learning on this step then
12:             Sample minibatch $B := \{(s^i, a^i, r^i, s^i_{\text{new}}, d^i)\}$ from $\mathcal{D}$
13:             $y^j := r^j + \gamma \max_{a'} Q(s^i_{\text{new}}, a'; \theta_{\text{target}})[\![d^i = \text{False}]\!]$
14:             Let loss $L(\theta) := \frac{1}{|B|}\sum_{i=1}^{|B|}(y^i - Q(s^i, a^i; \theta))^2$
15:             Gradient descent step $\theta := \theta - \eta \nabla_\theta L(\theta)$
16:         end if
17:         if Update target this step then
18:             $\theta_{\text{target}} := \theta$
19:         end if
20:     end while
21: end for
```

Deep Q-learning pseudocode (written)

```
Initialise replay buffer D to capacity N
Initialise state-action value network $Q(.;\theta)$ with parameters $\theta$
for episode = 1 to M do:
Sample initial state s from environment
d:= False
Initialise target parameters $\theta_{target} := \theta$
while d= False do:
sample action with an epsilon-greedy policy
sample $(s_new, r, d)$ from environment given $(s, a)$
store experience $(s, a, r, s_new, d)$ in D
if Learning on this step then:
sample minibatch $B := (s^i, a^i, r^i, s^i_{new}, d^i)$ from D
$y^j := r^j + \gamma \max_{a'} Q(s^i_{new}, a'; \theta_{target})[d^i = False]$
Let Loss $L(\theta) := \frac{1}{|B|} \sum_{i=1}^{|B|} (y^i - Q(s^i, a^i; \theta))^2$
Gradient descent step $\theta := \theta - \eta \nabla_\theta L(\theta)$
end if
if Update target this step then:
$\theta_{target} := \theta$
```

Exercise DQN

Let’s Code!!

Model free Reinforcement Learning

- Temporal Difference (TD)-learning
  First learn the optimal Q-values using the Bellman equation.
  Then find the optimal policy by taking
  $\pi_*(a|s) = \mathbb{1}\{a = \mathrm{argmax}_a Q_*(s,a)\}$.
- **Policy learning**
  Learn directly the policy $\pi(a|s)$ such as to maximize the expected return.

Policy learning

- Directly learn the policy using a neural network that takes the state as input and returns the probability distribution over all possible actions.
- Change the parameters of the network so as to maximize the expected return.

Policy learning - Update Rule

- Maximize the expected total return

$$J(\vec{\theta}) = \mathbb{E}_{\pi_\theta}[G_0|S_0] = \mathbb{E}_{\pi_\theta}[R_0 + \gamma R_1 + \gamma^2 R_2 + ...|S_0]$$

  using gradient ascent

$$\vec{\theta} \leftarrow \vec{\theta} + \alpha \nabla_{\vec{\theta}} J(\vec{\theta})$$

- The gradient of $J(\vec{\theta})$ can be found to be

$$\nabla_{\vec{\theta}} J(\vec{\theta}) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t G_t \nabla_{\vec{\theta}} \ln \pi_\theta(A_t|S_t)|S_0\right]$$

  with $G_t = \sum_{k=t}^{T} \gamma^{k-t} R_k$

Policy learning - Subtract baseline

Theorem

In the previous update rule, G_t can be replaced by $(G_t - b(S_t))$, for any b that does not depend on actions, without changing the parameters that maximize $J(\vec{\theta})$.

In general, the baseline leaves the expected value of the update unchanged, but it can have a large effect on its variance.

Property

The baseline that maximally reduces the variance of the training process is the mean.

Hence, to obtain the optimal update rule, replace $G_t = \sum_{k=t}^{T} \gamma^{k-t} R_k$ by

$$\mathcal{A}_t = G_t - V(S_t)$$

. Update rule for the policy network:

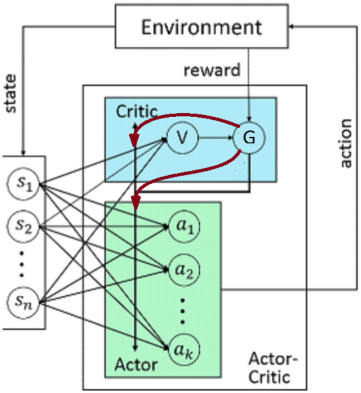
$$\vec{\theta}_{t+1} \leftarrow \vec{\theta}_t + \alpha \gamma^t \mathcal{A}_t \nabla_{\vec{\theta}} \ln[\pi_\theta(A_t|S_t)]$$

Actor-Critic Architecture

Problem: We do not know $V(S_k)$

- Learn the V-values in parallel of the policy by mean of a V-network.
- $\mathcal{A}_t = \sum_{k=t}^{T} \gamma^{k-t} R_k - V(S_t)$ is both what needs to be maximized by the policy network and minimized by the V-network $V(s|\vec{w})$.
- Update of the V-network:

$$\vec{w}_{t+1} \leftarrow \vec{w}_t + \alpha \gamma^t \mathcal{A}_t \nabla_{\vec{w}} V(S_t|\vec{w})$$



Advantage Actor-Critic

The previous update rules did not take into account the consistency condition between the V-values (Bellman equation), which is sub-optimal. Implementing it gives:

$$\mathcal{A}_t = R_t + \gamma V(S_{t+1}) - V(S_t)$$

- No need to wait for the end of the episode to compute the updates of the networks parameters anymore.

Proximal Policy Optimization

Idea

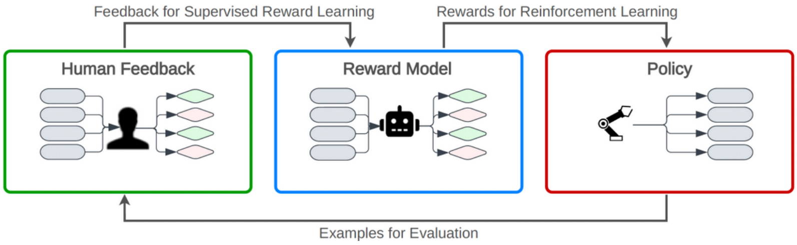
Even small changes in the parameters can lead to large policy update, which makes the training process prone to instabilities. The idea of PPO is to control the update of the policy.

PPO is an Advantage Actor-Critic architecture which additionally performs a sub-optimization at each training step to find the optimal change of parameters.

Ressource

- https://huggingface.co/blog/deep-rl-ppo

Reinforcement Learning from Human Feedback


Feedback for Supervised Reward Learning
Rewards for Reinforcement Learning
Human Feedback
Reward Model
Policy
Examples for Evaluation

RLHF - Reward Model

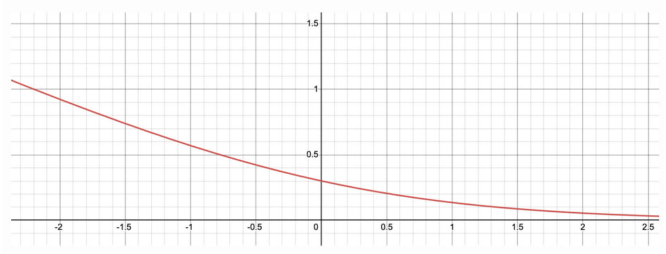
- Generate a dataset composed of triplets (x, y_1, y_2) where y_1, y_2 are two responses generated from the same input x using the pre-trained language model that must be fine-tuned.
- Use human feedback to determine for each triplet which response is best.
- Take a (second) pre-trained language model and replace the last unembedding layer (softmax) by a linear layer outputting a scalar to create a reward model.
- Train the reward model using

$$\mathcal{L}(\vec{\psi}) = \log\left[\sigma(r(x, y_1|\vec{\psi}) - r(x, y_2|\vec{\psi}))\right]$$

where y_1 is the preferred response.

RLHF - Reward Model

$$\mathcal{L}(\vec{\psi}) = \log\left[\sigma(r(x, y_1|\vec{\psi}) - r(x, y_2|\vec{\psi}))\right]$$



RLHF - Fine-tune large language model

Express LLM with a RL framework

- Action: choice of next token
- Language model: policy
- State: sequence of tokens
- Episode: complete output generated from a single prompt
- Reward: output of the reward model at the end of the episode (all tokens generated during the episode receive the same reward)

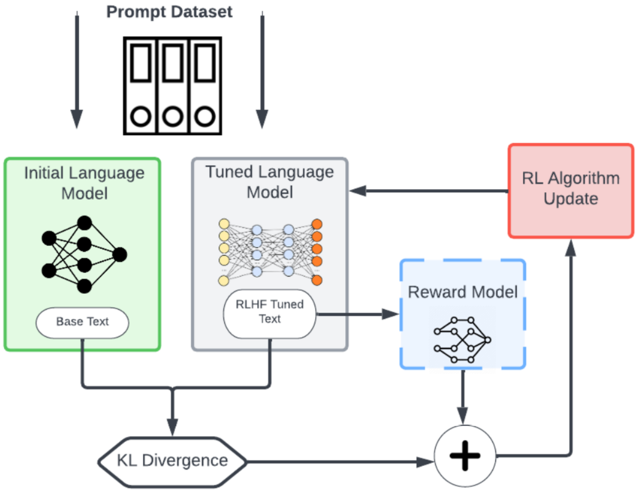
RLHF - Reward

- Add a KL divergence in the reward function to ensure that RL policy's output does not deviate drastically from the samples that the reward model encountered during its training phase.

$$r_{total} = r(x, y) - \eta \mathrm{KL}(\pi^{RL}(y|x) - \pi^{init}(y|x))$$

- Take a third pre-trained language model for the V-network and fine-tune the real language model using PPO algorithm.

RLHF


Prompt Dataset
Initial Language Model
Base Text
Tuned Language Model
RLHF Tuned Text
RL Algorithm Update
Reward Model
KL Divergence
+