Deep Reinforcement Learning

Matteo Gallici-Mario Martin

MIRS

December 3, 2024

Before we start

https://github.com/mttga/intro_rl

Recap

Monte-Carlo Policy Evaluation

- Goal: learn V^π from episodes of experience under policy π

$$S_1, A_1, r_2, S_2, A_2, r_3, \ldots, S_k \sim \pi$$

- Recall that the *return* is the total discounted reward:

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

- Recall that the value function is the expected return:

$$V^\pi(s) = \mathbb{E}_\pi[R_t | S_t = s] = \sum_\tau R_\tau p^\pi(\tau) \approx \frac{1}{N} \sum_{i=1}^{N} R_i$$

  where R_i is obtained from state s under π distribution (following π)
- Monte-Carlo policy evaluation uses *empirical mean* return instead of *expected* return

Monte-Carlo reinforcement learning

- MC uses the simplest possible idea: value = mean return. Instead of computing expectations, sample the long term return under the policy
- MC methods learn directly from episodes of experience
- MC is *model-free*: no explicit knowledge of environment mechanisms
- MC learns from complete episodes
  - Caveat: can only be applied to complete *episodic* environments (all episodes must terminate).

Monte-Carlo *policy learning*

Monte Carlo *policy learning*

Initialize π and Q randomly:

repeat

Generate trial using exploration method on greedy policy derived from Q values

for each s, a in trial **do**

$R \leftarrow$ return following the first occurrence of s, a

$Q(s, a) \leftarrow Q(s, a) + \alpha(R - Q(s, a))$

end for

until false

Temporal Differences *policy evaluation*

- Monte-Carlo methods compute expectation of Long-term-Reward averaging the return of several trials.
- Average is done after termination of the trial.
- We saw in previous lecture that Bellman equation also allow to estimate expectation of Long-term-Reward

$$\begin{aligned} Q^{\pi}(s,a) &= \mathbb{E}_{\pi}[R_t|S_t = s, A_t = a] \\ &= \mathbb{E}_{\pi}[r_{t+1} + \gamma Q^{\pi}(S_{t+1}, \pi(S_{t+1}))|S_t = s, A_t = a] \end{aligned}$$

- Computing expectations with world model:

$$Q^{\pi}(s,a) = \sum_{s'} P^{a}_{ss'} \left(r(s') + \gamma\, Q^{\pi}(s', \pi(s'))\right)$$

Temporal Differences *policy evaluation*

- How to get rid of the world-model?
- Q-value function and averaging, like in the case of MC

$$Q(S_t, a) \leftarrow Q(S_t, a) + \alpha(R_t(s_t) - Q(S_t, a))$$

- But now substitute R_t with Bellman equation:

$$Q(S_t, a) \leftarrow Q(S_t, a) + \alpha\left[r_{t+1} + \gamma Q(S_{t+1}, \pi(S_{t+1})) - Q(S_t, a)\right]$$

or

$$Q(S_t, a) \leftarrow (1 - \alpha)Q(S_t, a) + \alpha\left[r_{t+1} + \gamma Q(S_{t+1}, \pi(S_{t+1}))\right]$$

- This is called *bootstrapping*

Temporal Differences *policy learning*

Q-learning: Temporal Differences policy learning

Given π initialize Q randomly:
repeat
  $s \leftarrow$ initial state of episode
  repeat
    Set a using f.i. $\epsilon-$greedy strategy based on Q values
    Take action a and observe s' and r
    $Q(s,a) \leftarrow Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$
    $s \leftarrow s'$
  until s is terminal
until false

Off-policy vs. On-policy learning

- When learning value functions of a policy, we sample *using the policy* to estimate them
- In Q-learning, the method tries to learn the value function of the optimal policy (V^*) when in fact samples are obtained from different policy (ϵ-greedy policy)
- A subtle point with implications about the convergence of the algorithms to the optimal solution
- We'll do the following distinction:

  On-policy learning: When learning the value function V^π of the current policy π

  Off-policy learning: When Learning the value function V^π using another policy π'

Goal of this lecture

- Methods we have seen so far work well when we have a *tabular* representation for each state, that is, when we represent value function with a lookup table.

Goal of this lecture

- Methods we have seen so far work well when we have a *tabular* representation for each state, that is, when we represent value function with a lookup table.
- This is not reasonable on most cases:
  - In Large state spaces: There are too many states and/or actions to store in memory (f.i. Backgammon: 10^{20} states, Go 10^{170} states)
  - and in continuous state spaces (f.i. robotic examples)

Goal of this lecture

- Methods we have seen so far work well when we have a *tabular* representation for each state, that is, when we represent value function with a lookup table.
- This is not reasonable on most cases:
  - In Large state spaces: There are too many states and/or actions to store in memory (f.i. Backgammon: 10^{20} states, Go 10^{170} states)
  - and in continuous state spaces (f.i. robotic examples)
- In addition, we want to generalize from/to similar states to speed up learning. It is too slow to learn the value of each state individually.

Goal of this lecture

- We'll see now methods to learn policies for large state spaces by using *function approximation to estimate value functions*:

$$V_\theta(s) \approx V^\pi(s) \quad (1)$$

$$Q_\theta(s,a) \approx Q^\pi(s,a) \quad (2)$$

- θ is the set of parameters of the function approximation method (with size much lower than $|S|$)
- Function approximation allow to generalize from seen states to unseen states and to save space.
- Now, instead of storing V values, we will update θ parameters using MC or TD learning so they fulfill (1) or (2).

Which Function Approximation?

- There are many function approximators, e.g.
  - Artificial neural network
  - Decision tree
  - Nearest neighbor
  - Fourier/wavelet bases
  - Coarse coding
- In principle, any function approximator can be used. However, the choice may be affected by some properties of RL:
  - Experience is not i.i.d. – Agent's action affect the subsequent data it receives
  - During control, value function V(s) changes with the policy (non-stationary)
  - Continuous Learning.

Deep RL

Using Neural Networks for Regression

Mean Squared Error Loss:

- $\hat{y}_i$: Predicted output (or model's estimate for the i-th sample).
- y_i: Target output (ground truth for the i-th sample).
- N: Total number of samples in the dataset.

$$\mathcal{L}_{\text{MSE}} = \frac{1}{2N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2.$$

Using Neural Networks for Regression

Updating Parameters with SGD:

- $\mathbf{W}$: Neural network parameters (e.g., weights).
- η: Learning rate, which controls the step size in gradient descent.
- $\frac{\partial \mathcal{L}_{\text{MSE}}}{\partial \mathbf{W}}$: Gradient of the loss with respect to the parameters.

The parameters are updated iteratively using Stochastic Gradient Descent (SGD):

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial \mathcal{L}_{\text{MSE}}}{\partial \mathbf{W}}.$$

Key Concept: The goal of training is to adjust $\mathbf{W}$ such that $\hat{y}_i$ closely matches y_i, minimizing the loss.

Neural Networks in Practice

```
for i in range(n_epoch):
    y_pred = reg_model(X)
    step_loss = mse_loss(y_pred, y)
    optimizer.zero_grad()
    step_loss.backward()
    optimizer.step()
```

Value Function Approx. by SGD

Minimizing *Loss function* of the approximation

Goal: Find NN's parameters θ minimizing mean-squared error between approximate value function $V_\theta(s)$ and true value function $V^\pi(s)$

$$L(\theta) = \mathbb{E}_\pi \left[(V^\pi(s) - V_\theta(s))^2 \right]$$

- $V_\theta(s)$ is a neural network parametrized by θ that takes s as input.
- We can optimize the parameters with **Stochastic gradient descent** (SGD)
- But how do we get the targets for computing the loss?

Recap of FA solutions

Two possible approaches for function approximation:

1. Incremental:
   - Pro: Learning on-line
   - Cons: No convergence due to (a) Data not i.i.d., that can lead to *catastrophic forgetting*, and (b) Moving target problem

Recap of FA solutions

Two possible approaches for function approximation:

1. Incremental:
   - Pro: Learning on-line
   - Cons: No convergence due to (a) Data not i.i.d., that can lead to *catastrophic forgetting*, and (b) Moving target problem
2. Batch Learning:
   - Cons: Learn from collected dataset (not own experience)
   - Pro: Better convergence

Neural Fitted Q-learning (Riedmiller, 2005)

Neural Fitted Q-learning

Initialize weights θ for NN for regression
Collect $\mathcal{D}$ of size T with examples $(s_t, a_t, r_{t+1}, s_{t+1})$
repeat
  $\theta' \leftarrow \theta$
  repeat
    Sample $\mathcal{B}$ mini-batch of $\mathcal{D}$
    $\theta \leftarrow \theta - \alpha \sum_{t \in \mathcal{B}} \frac{\partial Q_\theta}{\partial \theta}(s_t, a_t)\left(Q_\theta(s_t, a_t) - [r_{t+1} + \gamma \max_{a'} Q_{\theta'}(s_{t+1}, a')]\right)$
  until convergence on learning or maximum number of steps
until maximum limit iterations
return π based on greedy evaluation of Q'_θ

- Notice target does not change during supervised regression

How to get the data?

- So now, we have learning stabilized just any batch method but using NN.
- However, now there is the problem of dependence of dataset $\mathcal{D}$. How we obtain the data?
- Data can be obtained using a random policy, but we want to minimize error on states visited by the policy!

$$L(\theta) = \mathbb{E}_\pi \left[(V^\pi(s) - V_\theta(s))^2 \right] = \sum_{s \in \mathcal{S}} \mu^\pi(s) \left[V^\pi(s) - V_\theta(s) \right]^2$$

where $\mu^\pi(s)$ is the time spent in state s while following π

How to get the data?

- Data should be generated by the policy
- But it also has to be probabilistic (to ensure exploration)
- So, collect data using the policy and add them to $\mathcal{D}$
- Also remove old data from $\mathcal{D}$.
  - Limit the size of the set
  - Remove examples obtained using old policies
- So, collect data using a *buffer* of limited size (we call **replay buffer**).

DQN algorithm (Mnih, *et al.* 2015)

- **Deep Q-Network** algorithm breakthrough
  - In 2015, Nature published DQN algorithm.
  - It takes profit of "then-recent" *Deep Neural Networks* and, in particular, of *Convolutional NNs* so successful for vision problems
  - Applied to Atari games directly from pixels of the screen (no hand made representation of the problem)
  - Very successful on a difficult task, surpassing in some cases human performance
- It goes back to incremental learning
- But is also batch learning

DQN algorithm (Mnih, *et al.* 2015)

DQN algorithm

Initialize weights θ for NN for regression
Set s to initial state, and k to zero
repeat
  Choose a from s using policy π_θ derived from Q_θ (e.g., ϵ-greedy)
  $k \leftarrow k + 1$
  Execute action a, observe r, s', and add $\langle s, a, r, s' \rangle$ to buffer $\mathcal{D}$
  Sample $\mathcal{B}$ mini-batch of $\mathcal{D}$
  $\theta \leftarrow \theta - \alpha \sum_{t \in \mathcal{B}} \frac{\partial Q_\theta}{\partial \theta}(s_t, a_t)\,(Q_\theta(s_t, a_t) - [r_{t+1} + \gamma \max_{a'} Q_{\theta'}(s_{t+1}, a')])$
  if k==N **then**
    $\theta' \leftarrow \theta$
    $k \leftarrow 0$
  end if
until maximum limit iterations
return π based on greedy evaluation of Q'_θ

Double Q-learning (Hasselt, et al. 2015)

- **Problem of overestimation of Q values**.
- We use "max" operator to compute the target in the minimization of:

$$L(s,a) = (Q(s,a) - (r + \gamma \max_{a'} Q(s',a')))^2$$

- Surprisingly here is a problem.
  1. Suppose $Q(s',a')$ is 0 for all actions, so $Q(s,a)$ should be r.
  2. But $\gamma \max_{a'} Q(s',a') \geq 0$ because random initialization and use of the max operator.
  3. So estimation $Q(s,a) \geq r$, overestimating true value
  4. All this because for max operator:

$$\mathbb{E}[\max_{a'} Q(s',a')] \geq \max_{a'} \mathbb{E}[Q(s',a')]$$

- This overestimation is propagated to other states.

Double DQN (Hasselt, et al. 2015)

- In DQN, in fact, we have 2 value functions: Q_θ and $Q_{\theta'}$
- so, no need to add another one:
  - Current Q-network θ is used to select actions
  - Older Q-network θ' is used to evaluate actions
- Update in Double-DQN (Hasselt, et al. 2015):

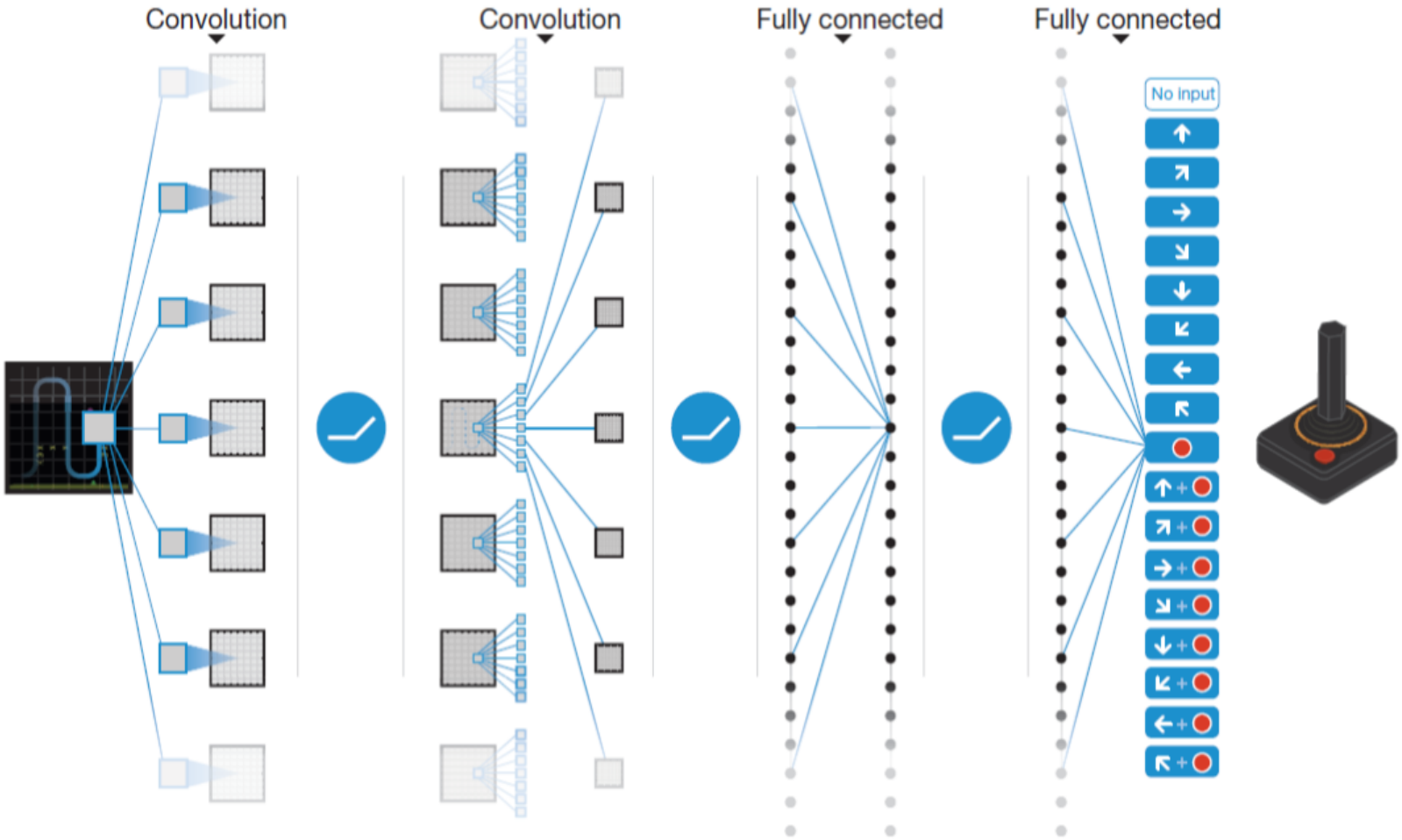
$$Q_\theta(s,a) \leftarrow r + \gamma \overbrace{Q_{\theta'}(s', \underbrace{\arg\max_{a'} Q_\theta(s',a')}_{\text{Action Selection}})}^{\text{Action Evaluation}}$$

- Works well in practice.

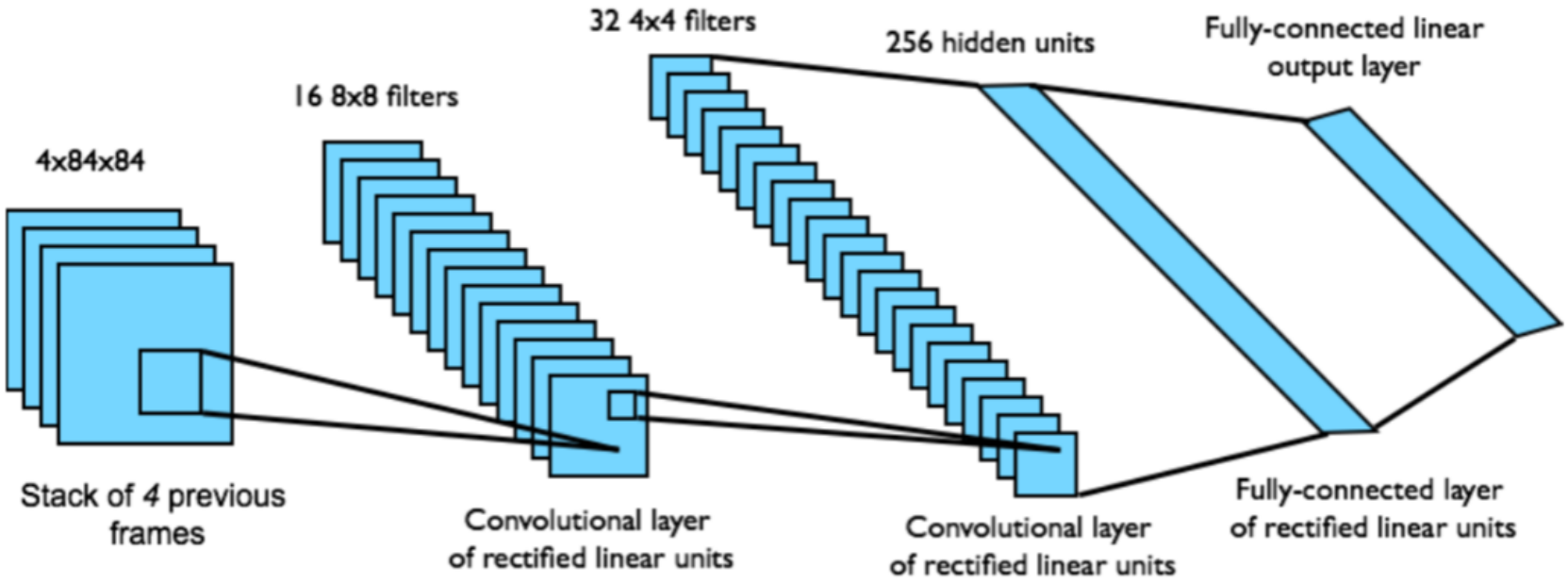
DQN algorithm on Atari

- End-to-end learning of values Q(s; a) from pixels:
  - **State:** Input state s is stack of raw pixels from last 4 frames
  - **Actions:** Output is $Q(s, a)$ value for each of 18 joystick/button positions
  - **Reward:** Reward is direct change in score for that step
- Network architecture and hyper-parameters **fixed across all games**, No tuning!
- Clipping reward -1,0,1 to avoid problem of different magnitudes of score in each game

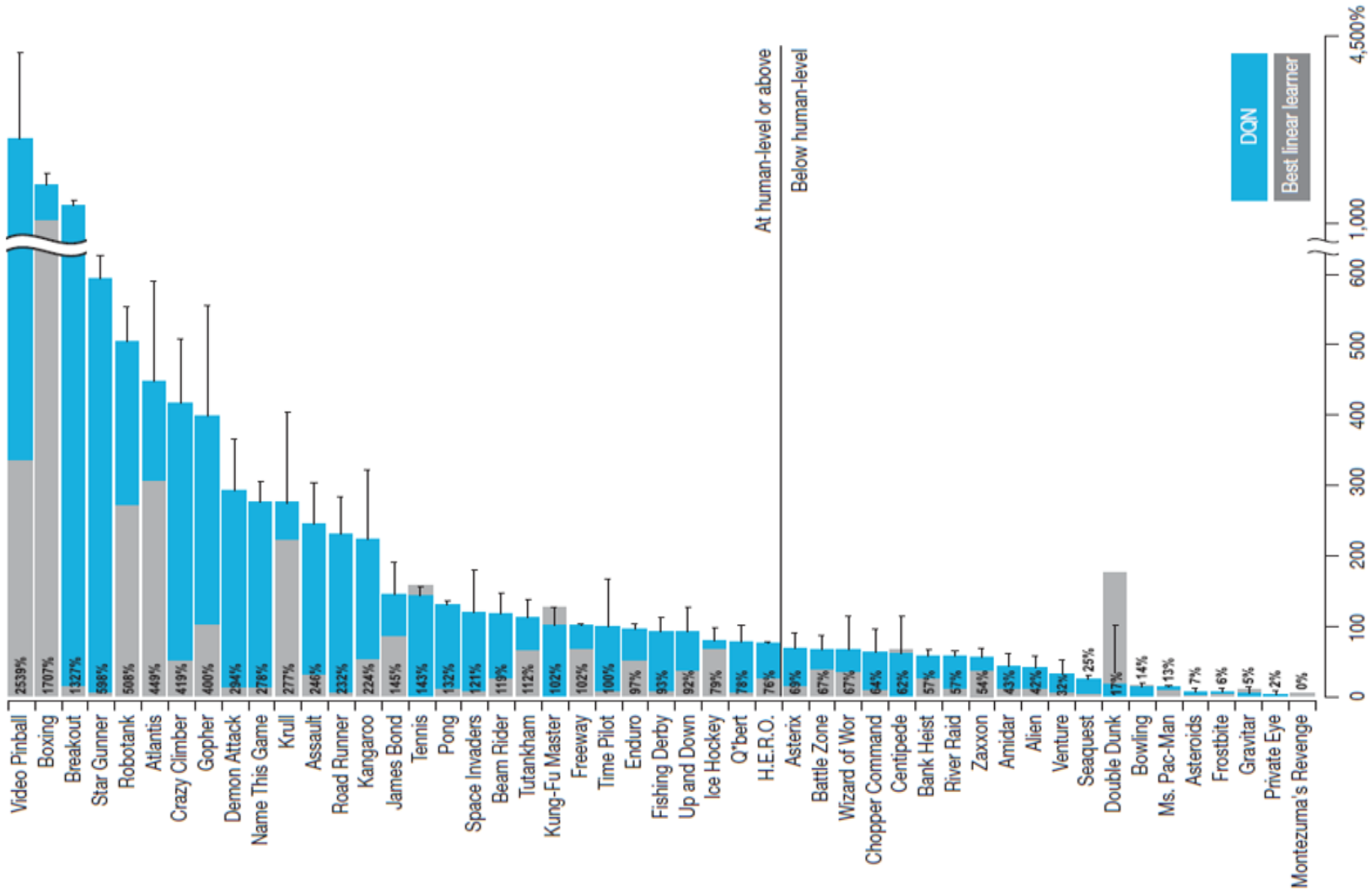
DQN algorithm on Atari



DQN algorithm on Atari



DQN algorithm on Atari



DQN algorithm on Atari

DQN algorithm on Atari

- What is the effect of each trick on Atari games?

DQN

	Q-learning	Q-learning + Target Q	Q-learning + Replay	Q-learning + Replay + Target Q
Breakout	3	10	241	**317**
Enduro	29	142	831	**1006**
River Raid	1453	2868	4103	**7447**
Seaquest	276	1003	823	**2894**
Space Invaders	302	373	826	**1089**

[Caveat: Applications of RL]

- Don't get it wrong. **RL main field of application is not games!**
- Some real world successful applications in RL:
  - Robotics: walking, manipulation, etc.
  - Autonomous driving.
  - Cooperation in a multi-agent team of marine UV.
  - Control of plasma in fusion reactors.
  - Adaptive optics in extra large telescopes.
  - Control of water or energy.
  - Medicine: Control of ventilation in Covid times or medication in Sepsis.
  - Design of chips (claimed superhuman level).
  - Design of medical drugs.
  - Chatbot (f.i. ChatGPT) training!
  - Design of boats in America's Cup.
- Applicable when goal can be expressed as a policy and we want an (almost) optimal (sometimes superhuman) behavior.

Policy Gradients

Policy Learning

- So far we approximated the value or action-value function using parameters θ (e.g. neural networks)

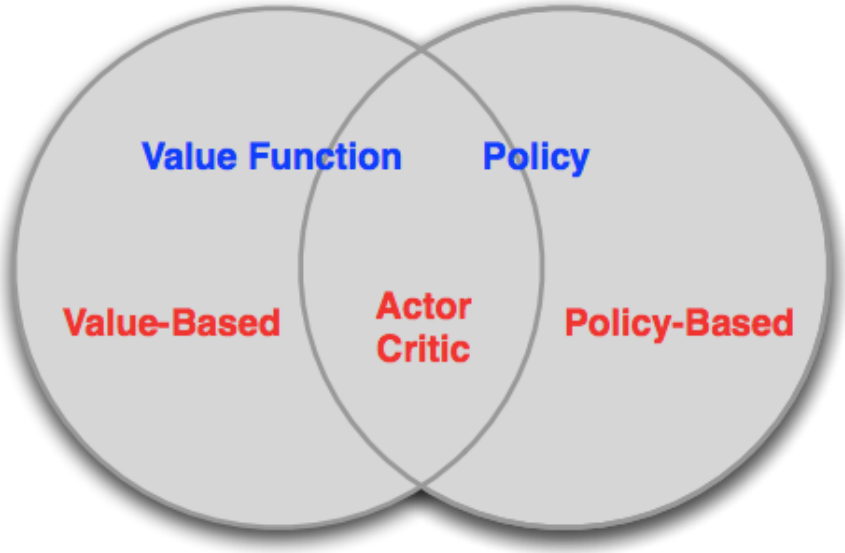
$$\begin{aligned} V_\theta &\approx V^\pi \\ Q_\theta(s,a) &\approx V^\pi(s) \end{aligned}$$

- A policy was generated directly from the value function e.g. using ϵ-greedy
- In this lecture we will directly parameterize the policy in a stochastic setting

$$\pi_\theta(a|s) = P_\theta(a|s)$$

- and do a direct **Policy search**
- Again on model-free setting

Three approaches to RL



Value based learning: **Implicit policy**

- Learn value function $Q_\theta(s, a)$ and from there infer policy $\pi(s) = \arg\max_a Q(s, a)$

Policy based learning: **No value function**

- Explicitly learn policy $\pi_\theta(a|s)$ that implicitly maximize reward over all policies

Actor-Critic learning: **Learn both Value Function and Policy**

Advantges of Policy over Value approach

- Advantages:
  - In some cases, computing Q-values is harder than picking optimal actions
  - **Better convergence properties**
  - **Effective in high dimensional or continuous action spaces**
  - Exploration can be directly controlled
  - **Can learn stochastic policies**
- Disadvantages:
  - Typically converge to a **local optimum** rather than a global optimum
  - Evaluating a policy is typically **data inefficient and high variance**

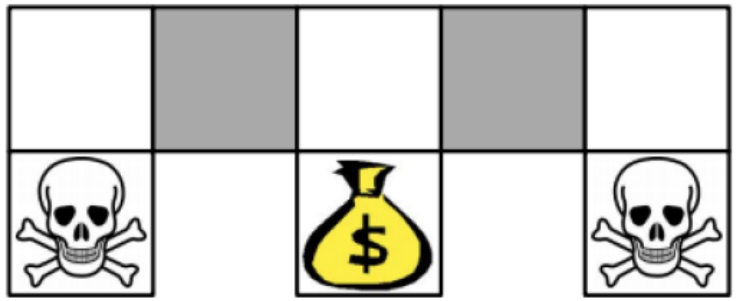
Stochastic Policies

- In general, two kinds of policies:
  - Deterministic policy
    $$a = \pi_\theta(s)$$
  - Stochastic policy
    $$P(a|s) = \pi_\theta(a|s)$$
- Nice thing is that they are **smoother** than greedy policies, and so, we can compute **gradients**!
- Not new: ϵ-greedy is stochastic...

Stochastic Policies

- In general, two kinds of policies:
  - Deterministic policy
    $$a = \pi_\theta(s)$$
  - Stochastic policy
    $$P(a|s) = \pi_\theta(a|s)$$
- Nice thing is that they are **smoother** than greedy policies, and so, we can compute **gradients**!
- Not new: ϵ-greedy is stochastic... but different idea. Stochastic policy is good on its own, not because it is an approx. of a greedy policy
- Any example where an stochastic policy could be better than a deterministic one?

Stochastic Policies when *aliased* states (POMDPs)



- The agent *cannot differentiate* the grey states
- Consider features of the following form:

$$\phi_d(s) = 1(\text{wall to } d) \quad \forall d \in \{N, E, S, W\}$$

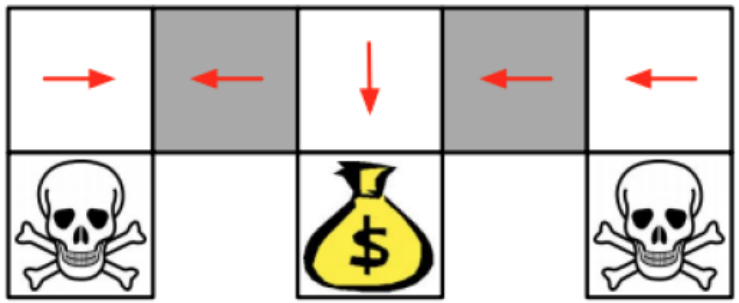
- Compare value-based RL, using an approximate value function

$$Q_\theta(s, a) = f_\theta(\phi(s, a))$$

- To policy-based RL, using a parametrized policy

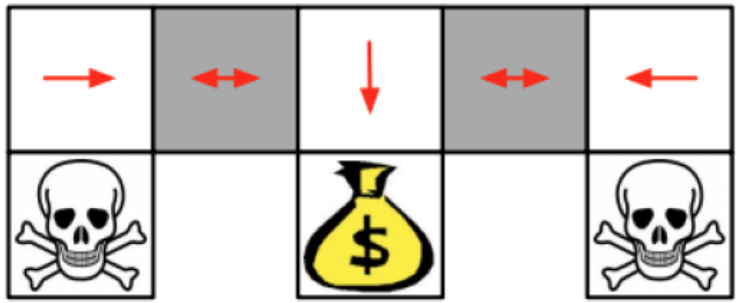
$$\pi_\theta(a|s) = g_\theta(\phi(s, a))$$

Stochastic Policies when *aliased* states (POMDPs)



- Under aliasing, an optimal deterministic policy will either
  - move W in both gray states
  - move E in both gray states
- Either way, it can get stuck and never reach the money
- So it will be stuck in the corridor for a long time
- Value–based RL learns a deterministic policy (or *near* deterministic when it explores)

Stochastic Policies when *aliased* states (POMDPs)



- An optimal stochastic policy will randomly move E or W in gray states
  - π_θ(move E | wall to N and S) $= 0.5$
  - π_θ(move W | wall to N and S) $= 0.5$
- It will reach the goal state in a few steps with high probability
- Policy–based RL can learn the optimal stochastic policy

Policy optimization

Policy Objective Functions

- Goal: given policy $\pi_\theta(a|s)$ with parameters θ, find best θ
- ... but how do we measure the quality of a policy π_θ?
- In episodic environments we can use the **start value**

$$J_1(\theta) = V^{\pi_\theta}(s_1)$$

Policy Objective Functions

- In continuing environments we can use the **average value**

$$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$$

  where $d^{\pi_\theta}(s)$ is stationary distribution of Markov chain for π_θ (can be estimated as the expected number of time steps on s in a randomly generated episode following π_θ divided by time steps of trial)
- Or the **average reward per time-step**

$$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(a|s) r(s,a)$$

- For simplicity, we will mostly discuss the episodic case, but can easily extend to the continuing / infinite horizon case

Policy optimization

- Goal: given policy $\pi_\theta(a|s)$ with parameters θ, find best θ
- Policy based reinforcement learning is an **optimization** problem
- Find policy parameters θ that maximize $J(\theta)$
- Two approaches for solving the optimization problem
  - Gradient-free
  - Policy-gradient

Policy gradient methods

- Policy based reinforcement learning is an **optimization** problem
- Find policy parameters θ that maximize V^{π_θ}
- We have seen gradient-free methods, but greater efficiency often possible using gradient in the optimization
- Pletora of methods:
  - Gradient descent
  - Conjugate gradient
  - Quasi-newton
- We focus on *gradient ascent*, many extensions possible
- And on methods that exploit sequential structure

Policy gradient differences wrt Value methods

- With Value functions we use Greedy updates:

$$\theta_{\pi'} = \arg\max_{\theta} \mathbb{E}_{\pi_\theta} \left[Q^{\pi}(s, a) \right]$$

- $V^{\pi_0} \xrightarrow{\text{small change}} \pi_1 \xrightarrow{\text{large change}} V^{\pi_1} \xrightarrow{\text{small change}} \pi_2 \xrightarrow{\text{large change}} V^{\pi_2}$
- Potentially unstable learning process with large policy jumps because arg max is not differentiable
- On the other hand, Policy Gradient updates are:

$$\theta_{\pi'} = \theta_{\pi'} + \alpha \frac{\partial J(\theta)}{\partial \theta}$$

- Stable learning process with smooth policy improvement

Policy gradient method

- Define $J(\theta) = J^{\pi_\theta}$ to make explicit the dependence of the evaluation policy on the policy parameters
- Assume episodic MDPs
- Policy gradient algorithms search for a local **maximum** in $J(\theta)$ by **ascending** the gradient of the policy, w.r.t parameters θ

$$\nabla\theta = \alpha\nabla_\theta J(\theta)$$

- Where $\nabla_\theta J(\theta)$ is the *policy gradient* and α is a step-size parameter

Computing the gradient analytically

- We now compute the policy gradient analytically
- **Assume policy is differentiable whenever it is non-zero**

Computing the gradient analytically

- We now compute the policy gradient analytically
- **Assume policy is differentiable whenever it is non-zero**
- and that we know the gradient $\nabla_\theta \pi_\theta(a|s)$
- Denote a state-action **trajectory** (or trial) τ as

$$\tau = (s_0, a_0, r_1, s_1, a_1, r_2, \ldots s_{T-1}, a_{T-1}, r_T, s_T)$$

- Define long-term-reward to be the sum of rewards for the trajectory ($R(\tau)$)

$$R(\tau) = \sum_{t=1}^{T} r(s_t)$$

- It works also for discounted returns.

Computing the gradient analytically

- The value of the policy $J(\theta)$ is:

$$J(\theta) = \mathbb{E}_{\pi_\theta}[R(\tau)] = \sum_{\tau} P(\tau|\theta)R(\tau)$$

  where $P(\tau|\theta)$ denotes the probability of trajectory τ when following policy π_θ
- Notice that sum is for all possible trajectories
- In this new notation, our goal is to find the policy parameters *theta*) that:

$$\arg\max_{\theta} J(\theta) = \arg\max_{\theta} \sum_{\tau} P(\tau|\theta)R(\tau)$$

[Log-trick: a convenient equality]

- In general, assume we want to compute $\nabla \log f(x)$:

$$\begin{aligned} \nabla \log f(x) &= \frac{1}{f(x)} \nabla f(x) \\ f(x) \nabla \log f(x) &= \nabla f(x) \end{aligned}$$

- It can be applied to any function and we can use the equality in any direction
- The term $\frac{\nabla f(x)}{f(x)}$ is called *likelihood ratio* and is used to analytically compute the gradients
- Btw. Notice the caveat... *Assume policy is differentiable whenever it is non-zero*.

Computing the gradient analytically

- In this new notation, our goal is to find the policy parameters θ that:

$$\arg\max_{\theta} J(\theta) = \arg\max_{\theta} \sum_{\tau} P(\tau|\theta) R(\tau)$$

- So, taken the gradient wrt θ

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \sum_{\tau} P(\tau|\theta) R(\tau) \\
&= \sum_{\tau} \nabla_\theta P(\tau|\theta) R(\tau) \\
&= \sum_{\tau} \frac{P(\tau|\theta)}{P(\tau|\theta)} \nabla_\theta P(\tau|\theta) R(\tau) \\
&= \sum_{\tau} P(\tau|\theta) R(\tau) \frac{\nabla_\theta P(\tau|\theta)}{P(\tau|\theta)} \\
&= \sum_{\tau} P(\tau|\theta) R(\tau) \nabla_\theta \log P(\tau|\theta)
\end{aligned}$$

Computing the gradient analytically

- Goal is to find the policy parameters θ that:

$$\arg\max_{\theta} J(\theta) = \arg\max_{\theta} \sum_{\tau} P(\tau|\theta) R(\tau)$$

- So, taken the gradient wrt θ

$$\nabla_{\theta} J(\theta) = \sum_{\tau} P(\tau|\theta) R(\tau) \nabla_{\theta} \log P(\tau|\theta)$$

- Of course we cannot compute all trajectories...

Computing the gradient analytically

- Goal is to find the policy parameters θ that:

$$\arg\max_{\theta} J(\theta) = \arg\max_{\theta} \sum_{\tau} P(\tau|\theta) R(\tau)$$

- So, taken the gradient wrt θ

$$\nabla_{\theta} J(\theta) = \sum_{\tau} P(\tau|\theta) R(\tau) \nabla_{\theta} \log P(\tau|\theta)$$

- Of course we cannot compute all trajectories...but we can sample m trajectories because of the form of the equation

$$\nabla_{\theta} J(\theta) \approx (1/m) \sum_{i=1}^{m} R(\tau_i) \nabla_{\theta} \log P(\tau_i|\theta)$$

Computing the gradient analytically: at last!

- Sample m trajectories:

$$\nabla_\theta J(\theta) \approx (1/m) \sum_{i=1}^{m} R(\tau_i) \nabla_\theta \log P(\tau_i|\theta)$$

- However, we still have a problem, we don't know the how to compute $\nabla_\theta \log P(\tau|\theta)$
- Fortunately, we can derive it from the stochastic policy

$$\begin{aligned}
\nabla_\theta \log P(\tau|\theta) &= \nabla_\theta \log \left[\mu(s_0) \prod_{i=0}^{T-1} \pi_\theta(a_i|s_i) P(s_{i+1}|s_i, a_i) \right] \\
&= \nabla_\theta \left[\log \mu(s_0) + \sum_{i=0}^{T-1} \log \pi_\theta(a_i|s_i) + \log P(s_{i+1}|s_i, a_i) \right] \\
&= \sum_{i=0}^{T-1} \underbrace{\nabla_\theta \log \pi_\theta(a_i|s_i)}_{\text{No dynamics model required!}}
\end{aligned}$$

Computing the gradient analytically

- We assumed at the beginning that policy is differentiable and that we now the derivative wrt parameters θ
- So, we have the desired solution:

$$\nabla_\theta J(\theta) \approx (1/m) \sum_{i=1}^{m} \left(R(\tau_i) \sum_{(s_j, a_j) \in \tau_i} \nabla_\theta \log \pi_\theta(a_j | s_j) \right)$$

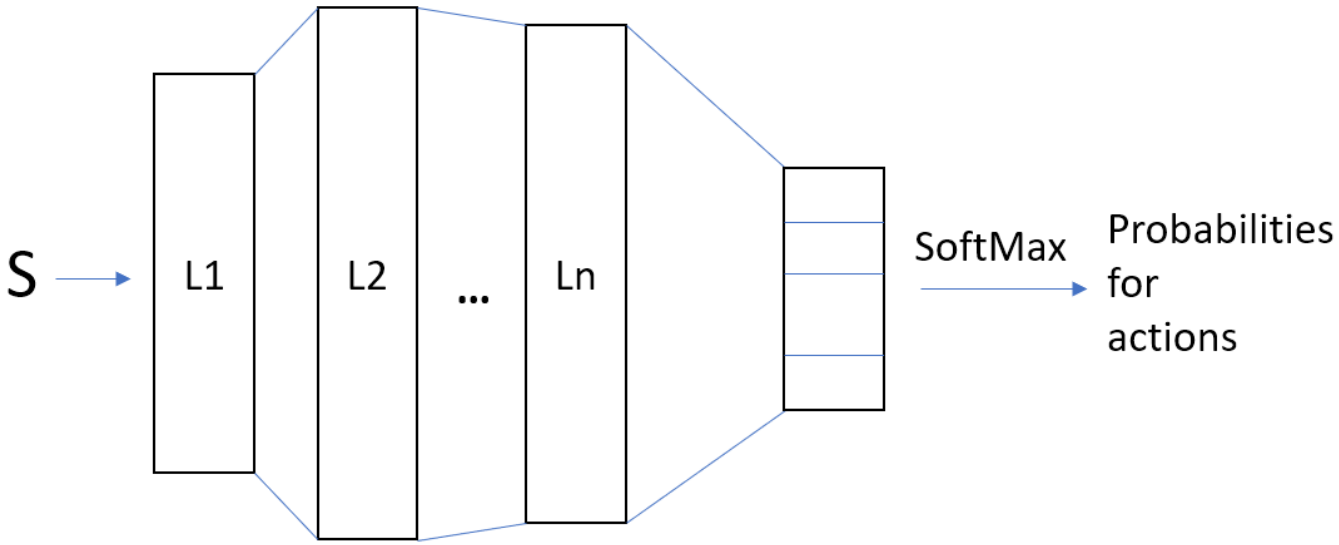
Differentiable policies? Deep Neural Network

- A very popular way to approximate the policy is to use a Deep NN with soft-max last layer with so many neurons as actions.
- In this case, use *autodiff* of the neural network package you use! In `pytorch`:

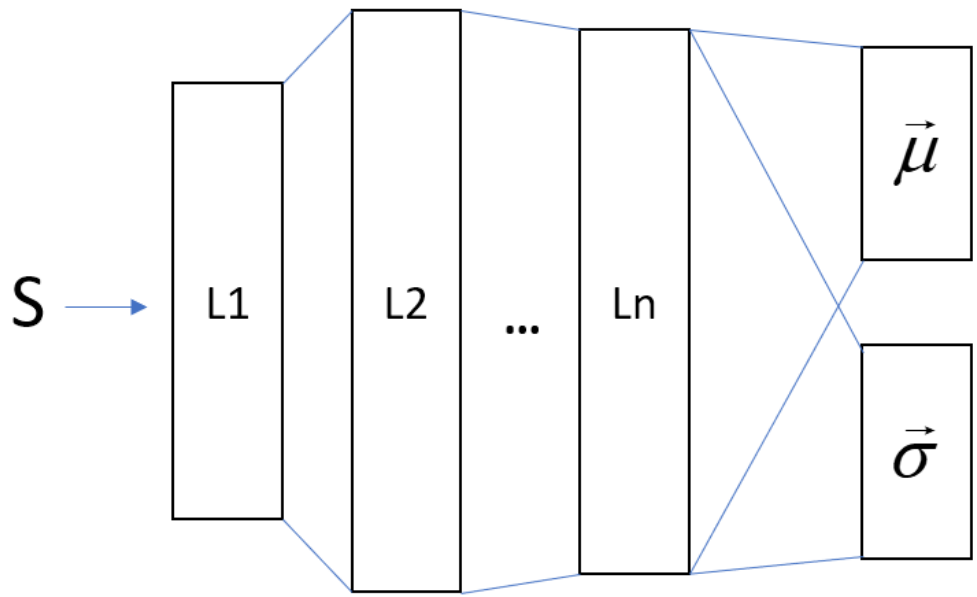
  ```
  loss = - torch.mean(log_outputs * R)
  ```

  where `prob_outputs` is the output layer of the DNN and `R` the long term reward.
- Backpropagation is implemented in `pytorch` and will do the work for you!
- Common approaches for stochastic policies:
  - Last *softmax* layer in discrete case
  - Last layer with μ and $\log \sigma$ in continuous case

Discrete action space

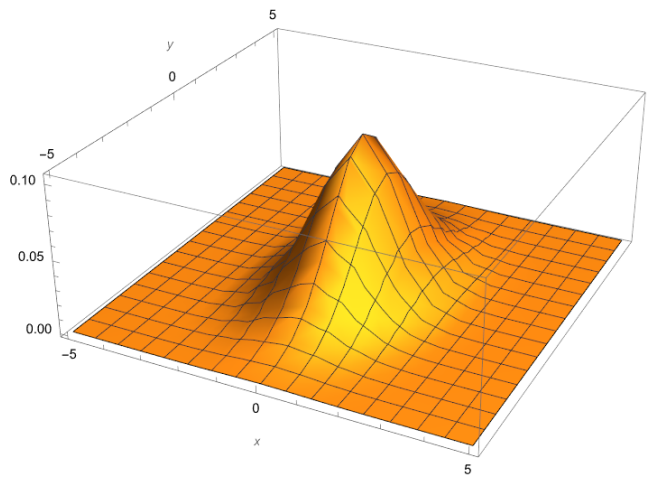


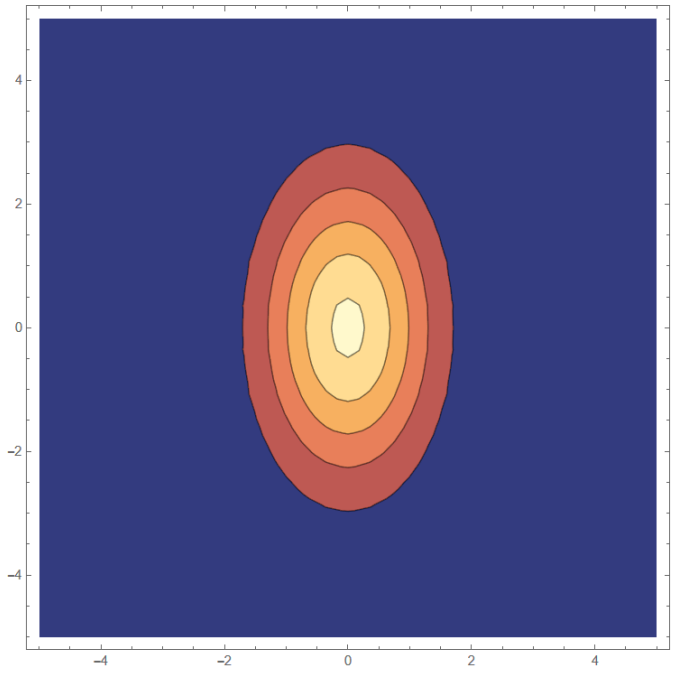
Continuous action space



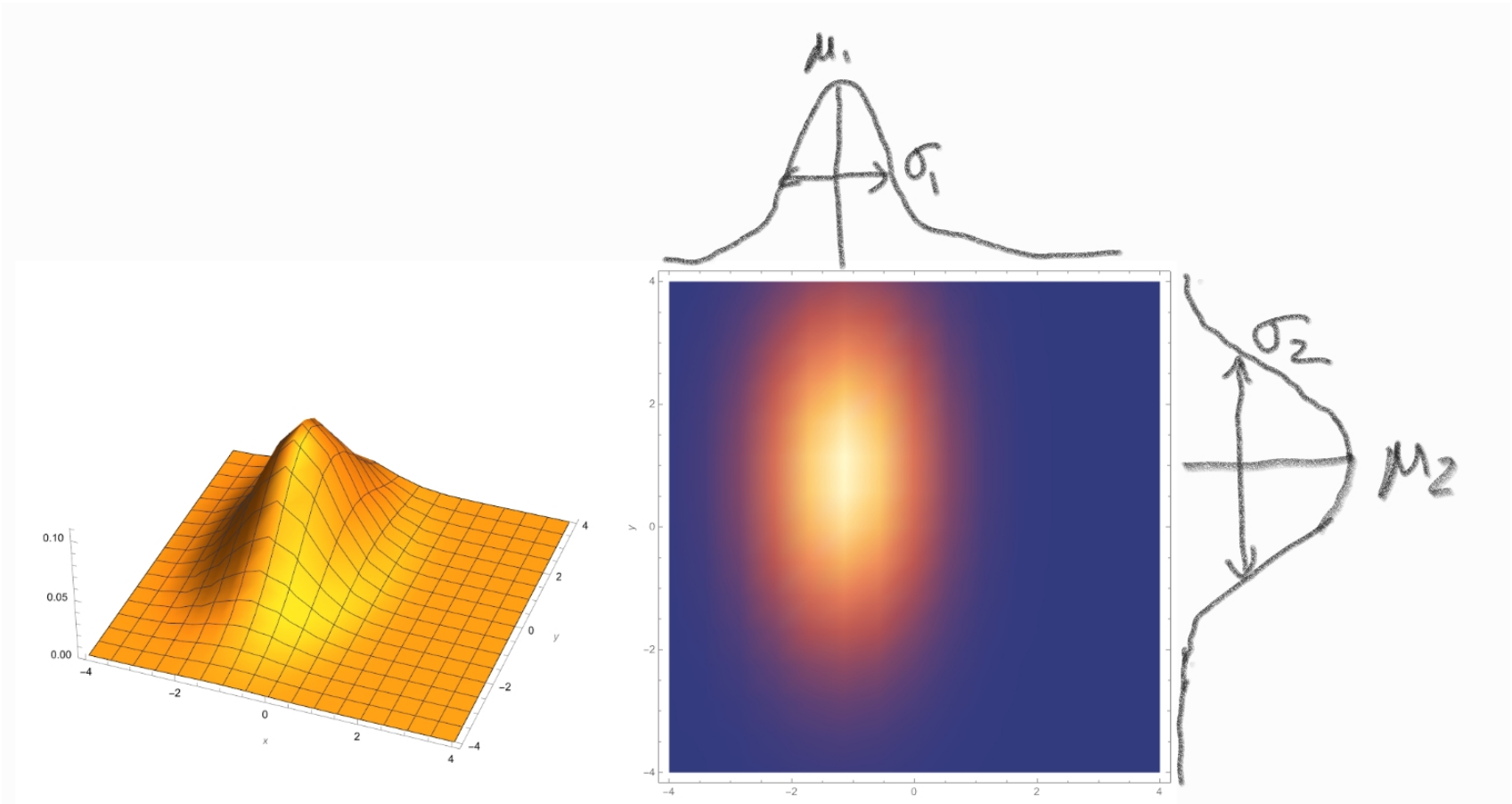
Define diagonal multivariate Gaussian probability distribution

Continuous action space

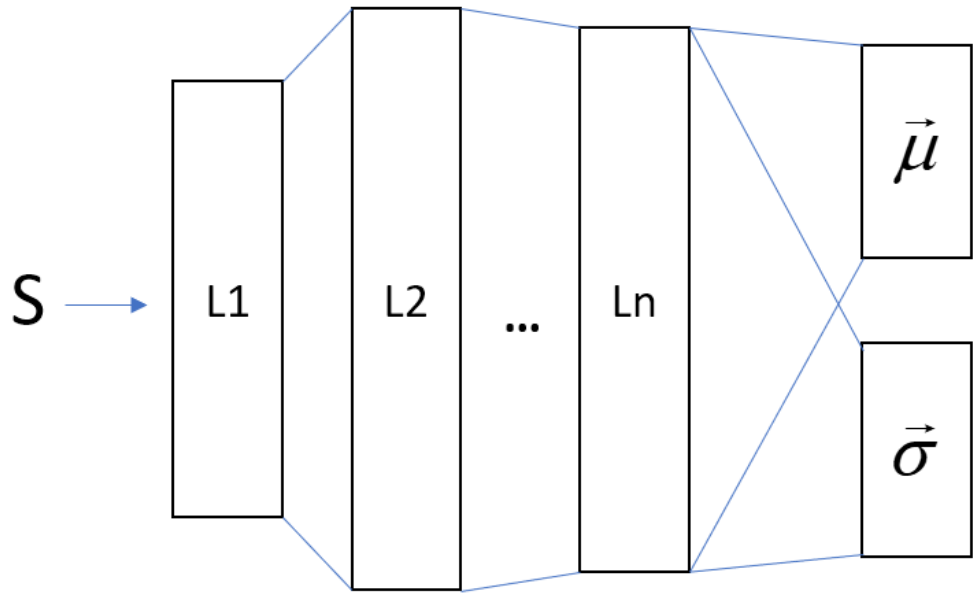




Continuous action space

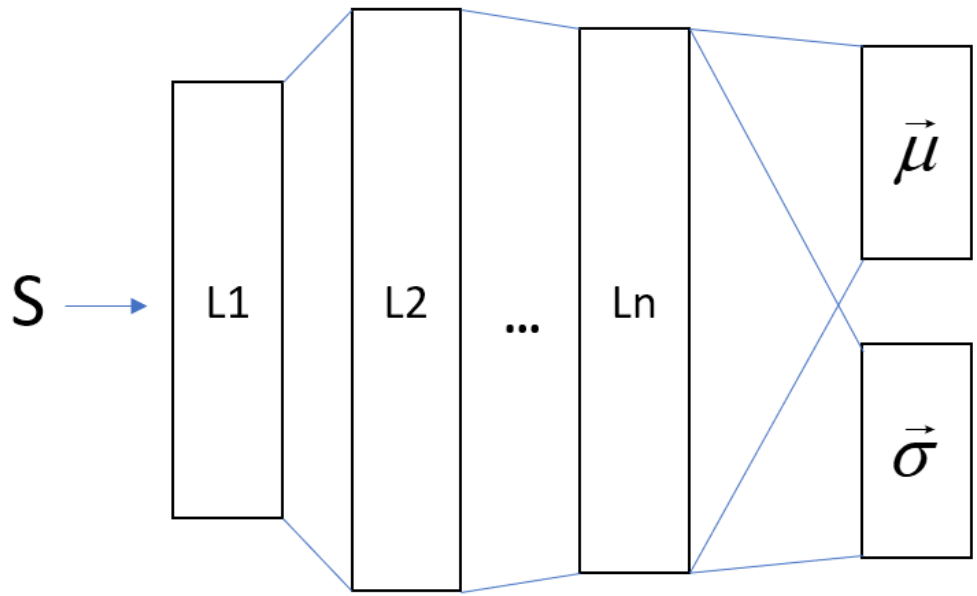


Continuous action space



Define diagonal multivariate Gaussian probability distribution

Continuous action space



Define diagonal multivariate Gaussian probability distribution

Yes!!!! **Continuous actions**! Big improvement in applicability of RL!

Vanilla Policy Gradient

Vanilla Policy Gradient

Given architecture with parameters θ to implement π_θ
Initialize θ randomly
repeat
  Generate episode $\{s_1, a_1, r_2, \ldots s_{T-1}, a_{T-1}, r_T, s_T\} \sim \pi_\theta$
  Get $R \leftarrow$ long-term return for episode
  for all time steps $t = 1$ to $T - 1$ **do**
    $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t) R$
  end for
until convergence

Substitute $\nabla_\theta \log \pi_\theta(a_t|s_t)$ with appropriate equation.

Btw, notice no explicit exploration mechanism needed when policies are stochastic (all on policy)!

Vanilla Policy Gradient

- Remember:

$$\nabla_\theta J(\theta) \approx (1/m) \sum_{i=1}^{m} R(\tau_i) \sum_{i=0}^{T-1} \nabla_\theta \log \pi_\theta(a_i|s_i)$$

- Unbiased but very noisy
- Fixes that can make it practical
  - Temporal structure
  - Baseline

Subsection 1

Reduce variance using temporal structure: Reinforce and Actor-Critic architectures

REINFORCE algorithm

- An deeper analysis shows we can also consider rewards-to-go for states instead of rewards of whole trajectory, adding temporal information to the algorithm and improving learning.

REINFORCE algorithm

Given architecture with parameters θ to implement π_θ
Initialize θ randomly
repeat
  Generate episode $\{s_1, a_1, r_2, \ldots s_{T-1}, a_{T-1}, r_T, s_T\} \sim \pi_\theta$
  for all time steps $t = 1$ to $T - 1$ **do**
    Get $R_t \leftarrow$ long-term return from step t to T[a]
    $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t) R_t$
  end for
until convergence

[a] See proof from Don't Let the Past Distract You if you are not convinced.

REINFORCE algorithm with baseline

- Monte-Carlo policy gradient still has high variance because R_t has a lot of variance
- We can **reduce variance subtracting** a **baseline** to the estimator

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t)(R_t - b(s_t))$$

- without introducing any bias *when baseline does not depend on actions* taken
- A good baseline is $b(s_t) = V^{\pi_\theta}(s_t)$ so we will use that

REINFORCE algorithm with baseline

- Monte-Carlo policy gradient still has high variance because R_t has a lot of variance
- We can **reduce variance subtracting** a **baseline** to the estimator

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t)(R_t - b(s_t))$$

- without introducing any bias *when baseline does not depend on actions* taken
- A good baseline is $b(s_t) = V^{\pi_\theta}(s_t)$ so we will use that
- **How to estimate** V^{π_θ}?
- **We'll use another set of parameters** w to approximate

REINFORCE algorithm with baseline

REINFORCE algorithm with baseline

Given architecture with parameters θ to implement π_θ and parameters w to approximate V

Initialize θ randomly

repeat

  Generate episode $\{s_1, a_1, r_2, \ldots s_{T-1}, a_{T-1}, r_T, s_T\} \sim \pi_\theta$

  for all time steps $t = 1$ to $T - 1$ **do**

    Get $R_t \leftarrow$ long-term return from step t to T

    $\delta \leftarrow R_t - V_w(s_t)$

    $w \leftarrow w + \beta\delta\nabla_w V_w(s_t)$

    $\theta \leftarrow \theta + \alpha\delta\nabla_\theta \log \pi_\theta(a_t|s_t)$

  end for

until convergence

Actor-Critic Architectures

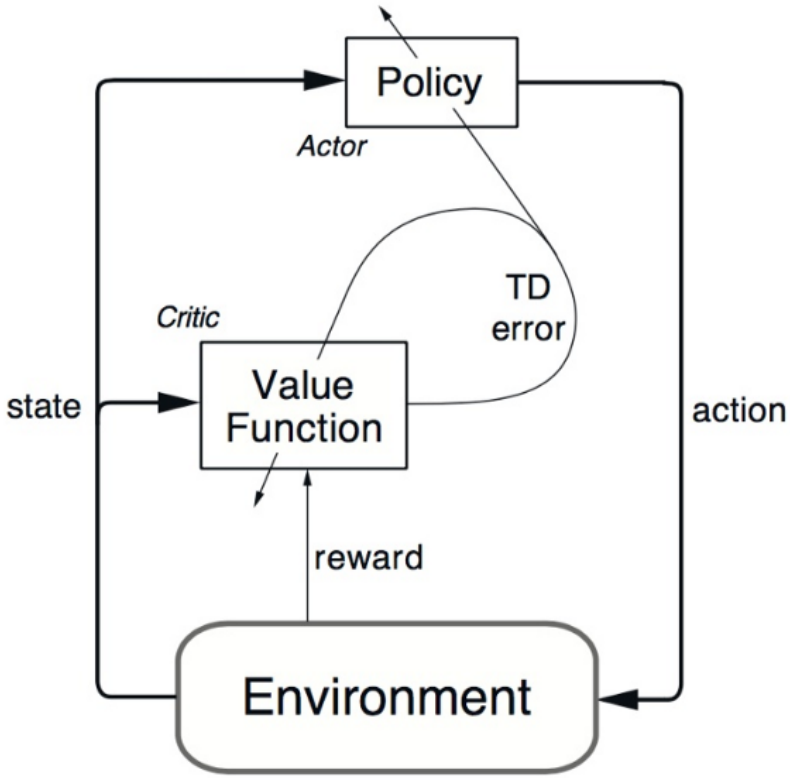
- Monte-Carlo policy gradient has high variance
- So we used a baseline to reduce the variance $R_t - V(s_t)$
- Can we do something to speed up learning like we did with MC using TD?

Actor-Critic Architectures

- Monte-Carlo policy gradient has high variance
- So we used a baseline to reduce the variance $R_t - V(s_t)$
- Can we do something to speed up learning like we did with MC using TD?
- Yes, **use different estimators of R_t that do bootstrapping** f.i. TD(0), n-steps, etc.
- These algorithms are called **Actor Critic**

Actor-Critic Architectures

- The **Critic**, *evaluates the current policy* and the result is used in the policy training
- The **Actor** *implements the policy* and is trained using Policy Gradient with estimations from the critic



Actor-Critic Architectures

- Actor-critic algorithms maintain two sets of parameters (like in REINFORCE with baseline):
  - **Critic** parameters: approximation parameters w for action-value function under current policy
  - **Actor** parameters: policy parameters θ
- Actor-critic algorithms follow an approximate policy gradient:
  - **Critic:** Updates action-value function parameters w like in *policy evaluation* updates (you can apply everything we saw in FA for prediction)
  - **Actor:** Updates policy gradient θ, in direction suggested by critic

Actor-Critic Architectures

- Actor updates are always in the same way:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t) G_t$$

  where G_t is the evaluation of long-term returned by the critic for s_t
- Critic updates are done to evaluate the current policy

$$w \leftarrow w + \alpha \delta \nabla_\theta V_w(a_t|s_t)$$

  where δ is the estimated error in evaluating the s state and that implements the kind of bootstrapping done.

Choices

- The policy gradient has many equivalent forms

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) R_t\right] \qquad \text{REINFORCE}$$

$$= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) Q_w(s,a)\right] \qquad \text{Actor-Critic}$$

$$= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) A_w(s,a)]\right] \qquad \text{Advantage Actor-Critic}$$

$$= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) \delta\right] \qquad \text{TD Actor-Critic}$$

$$= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) A_{GAE}\right] \qquad \text{Generalized Actor Critic}$$

- Each leads a stochastic gradient ascent algorithm
- Critic uses **policy evaluation** (e.g. MC or TD learning) to estimate $Q^\pi(s,a)$, $A^\pi(s,a)$ or $V^\pi(s)$

Advantage Actor Critic (AAC or A2C)

- In this critic Advantage value function is used:

$$A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$$

- The advantage function can significantly reduce variance of policy gradient
- So the critic should really estimate the advantage function, for instance, estimating **both** V(s) and Q using two function approximators and two parameter vectors:

$$V^{\pi_\theta}(s) \approx V_v(s) \tag{3}$$
$$Q^{\pi_\theta}(s,a) \approx Q_w(s,a) \tag{4}$$
$$A(s,a) = Q_w(s,a) - V_v(s) \tag{5}$$

- And updating both value functions by e.g. TD learning
- Nice thing, you only punish policy when not optimal (why?) Do you see resemblance with REINFORCE with baseline?

Other versions of A2C

- One way to implement A2C method without two different networks to estimate $Q_w(s,a)$ and $V_v(s)$ is to use estimators of $Q_w(s,a)$.
- For instance, TD Advantage estimator:

$$\begin{aligned} A^{\pi_\theta}(s,a) &= Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s) \\ &= \mathbb{E}_{\pi_\theta}\left[r + \gamma V^{\pi_\theta}(s')|s,a\right] - V^{\pi_\theta}(s) \end{aligned}$$

- or MonteCarlo Advantage estimator:

$$\begin{aligned} A^{\pi_\theta}(s,a) &= Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s) \\ &= \mathbb{E}_{\pi_\theta}\left[R|s,a\right] - V^{\pi_\theta}(s) \end{aligned}$$

- In practice these approaches only require one set of critic parameters v to approximate TD error

Generalized Advantage Estimator (GAE)

- Generalized Advantage Estimator (Schulman et al. 2016). [nice review]
- Use a version of Advantage that consider weighted average of n-steps estimators of advantage like in TD(λ):

$$A^{\pi}_{GAE} = \sum_{t'=t}^{\infty} (\lambda\gamma)^{t'-t} \underbrace{[r_{t'+1} + \gamma V^{\pi}_{\theta}(s_{t'+1}) - V^{\pi}_{\theta}(s_{t'})]}_{\text{t'-step advantage}}$$

- Used in continuous setting for locomotion tasks

Asyncrhonous Advantage Actor Critic (A3C)

- A3C (Mnih et al. 2016) idea: Sample for data can be parallelized using several copies of the same agent
  - use N copies of the agents (workers) working in parallel collecting samples and computing gradients for policy and value function
  - After some time, pass gradients to a main network that updates actor and critic using the gradients of all
  - After some time the worker copy the weights of the global network
- This parallelism decorrelates the agents' data, so **no Experience Replay Buffer needed**
- Even one can explicitly use different exploration policies in each actor-learner to maximize diversity
- Asynchronism can be extended to other update mechanisms (Sarsa, Q-learning...) but it works better in Advantage Actor critic setting

Asyncrhonous Advantage Actor Critic (A3C)

- What about exploration in Policy Gradient methods?

Asyncrhonous Advantage Actor Critic (A3C)

- What about exploration in Policy Gradient methods?
- Policy is stochastic, so naturally it explores
- But degree of exploration usually converges too fast
- Usually, in the loss function, a term is added that encourages exploration
- This is done computing the *Entropy* of the policy:

$$H\left(\pi\left(\cdot \mid s_t\right)\right) = -\sum_{a \in A} \pi\left(a \mid s_t\right) \log \pi\left(a \mid s_t\right)$$

Advanced Topics

Reinforcement learning with human Feedback

Reinforcement learning with human Feedback

- We know that we have problems with undesired behavior due to flaws or under-specification of the reward function.
- Also that in some cases it is complex to define a reward function

Reinforcement learning with human Feedback

- We know that we have problems with undesired behavior due to flaws or under-specification of the reward function.
- Also that in some cases it is complex to define a reward function
- What can we do? ... One approach learn from examples both directly or with IRL
- But **what happens when we don't have examples or they are very costly to obtain?**

Reinforcement learning with human Feedback

- We know that we have problems with undesired behavior due to flaws or under-specification of the reward function.
- Also that in some cases it is complex to define a reward function
- What can we do? ... One approach learn from examples both directly or with IRL
- But **what happens when we don't have examples or they are very costly to obtain?**

- In this case we can use another approach: Reward modelling using human feedback

DRL from Human Preferences (Christiano et al. 17)

- *Deep reinforcement learning from human preferences* (Christiano et al. 17)
- Problem: we don't know how to define a good reward function for a task,
- ... but we know to recognize a good trajectory
- Idea: Learn a reward function with human help
- However, some problems:
  - We don't have examples for learning

DRL from Human Preferences (Christiano et al. 17)

- *Deep reinforcement learning from human preferences* (Christiano et al. 17)
- Problem: we don't know how to define a good reward function for a task,
- ... but we know to recognize a good trajectory
- Idea: Learn a reward function with human help
- However, some problems:
  - We don't have examples for learning $\rightarrow$ let's the agent generate examples for evaluation

DRL from Human Preferences (Christiano et al. 17)

- *Deep reinforcement learning from human preferences* (Christiano et al. 17)
- Problem: we don't know how to define a good reward function for a task,
- ... but we know to recognize a good trajectory
- Idea: Learn a reward function with human help
- However, some problems:
  - We don't have examples for learning $\rightarrow$ let's the agent generate examples for evaluation
  - We don't know to assign numbers to each trajectory

DRL from Human Preferences (Christiano et al. 17)

- *Deep reinforcement learning from human preferences* (Christiano et al. 17)
- Problem: we don't know how to define a good reward function for a task,
- ... but we know to recognize a good trajectory
- Idea: Learn a reward function with human help
- However, some problems:
  - We don't have examples for learning $\rightarrow$ let's the agent generate examples for evaluation
  - We don't know to assign numbers to each trajectory ... but we know to generate *preferences* (if a trajectory is better than another)

DRL from Human Preferences (Christiano et al. 17)

Proposal

Learn a DNN that **modelize the reward function** and that **adheres to preferences of the human** for **trajectories generated by the agent**!

DRL from Human Preferences (Christiano et al. 17)

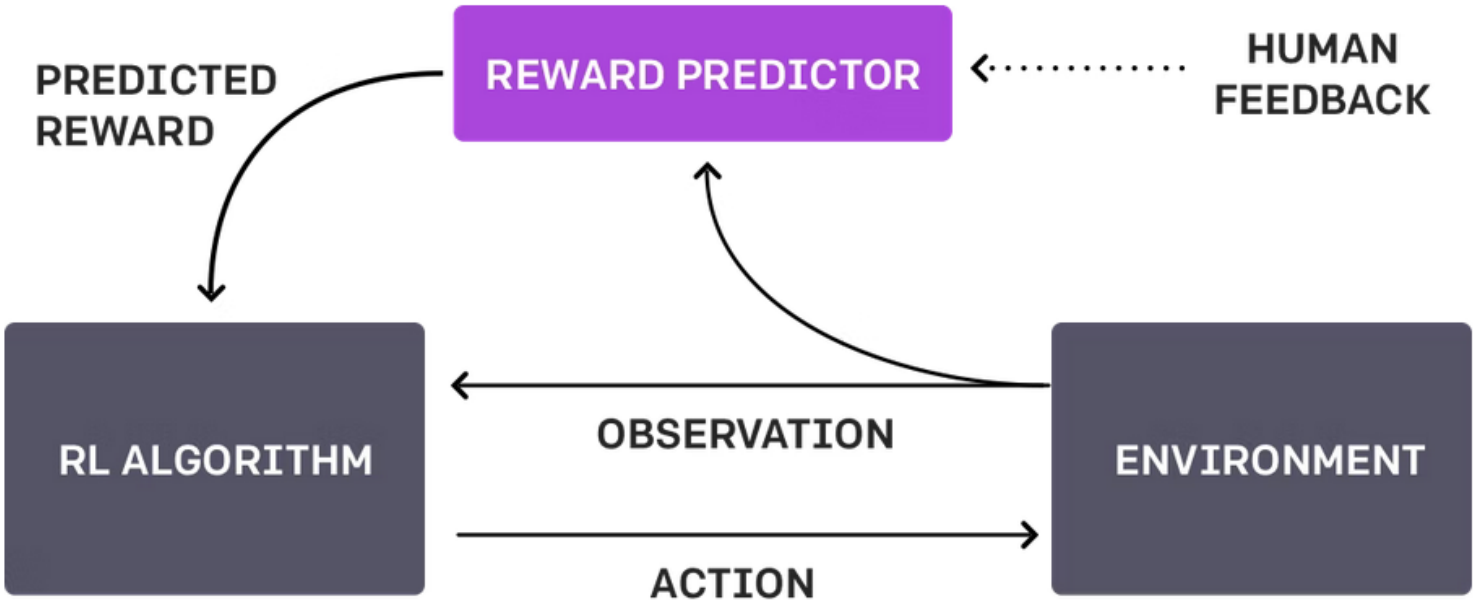
Proposal

Learn a DNN that **modelize the reward function** and that **adheres to preferences of the human** for **trajectories generated by the agent**!

- Still some problems:
  - Humans don't have time to evaluate a lot of examples
  - Examples change with the policy (become more competitive)
- Solution will be to include the human in learning loop

DRL from Human Preferences (Christiano et al. 17)

Reinforcement Learning with Human Feedback (RLHF)



DRL from Human Preferences (Christiano et al. 17)

- Example: Train Hopper to make flips
- Not defined task in Mujoco for Hopper[1]
- Authors applied the RLHF to the task.
- Repeat until good behavior is obtained:
  1. Agent generates two trajectories and ask for preferences to the human
  2. Human shows preference and it is added to set of preferences
  3. Reward function is trained to generate rewards according to preferences
  4. For some steps, agent is trained using RL with that reward function
- See video here

[1] Reward function in fact, can be defined by hand. However, it took two hours of hard work for researchers to develop it.

DRL from Human Preferences (Christiano et al. 17)

- Loss function for reward function:

$$\text{loss}(\hat{r}) = -\sum_{(\sigma^1,\sigma^2,\mu)\in\mathcal{D}} \mu(1)\log\hat{P}\left[\sigma^1 \succ \sigma^2\right] + \mu(2)\log\hat{P}\left[\sigma^2 \succ \sigma^1\right]$$

  where σ are trajectories (or pieces of trajectory) and $\mu(i)$ indicated preferences of trajectory i or not.

- Probability $\hat{P}\left[\sigma^1 \succ \sigma^2\right]$ of preference according r is defined as soft-max:

$$\hat{P}\left[\sigma^1 \succ \sigma^2\right] = \frac{\exp\sum\hat{r}\left(o_t^1, a_t^1\right)}{\exp\sum\hat{r}\left(o_t^1, a_t^1\right) + \exp\sum\hat{r}\left(o_t^2, a_t^2\right)}$$

  where the sums are done for all pairs o_i, a_i of state, action of the trajectory

DRL from Human Preferences (Christiano et al. 17)

- Other tricks in implementation (see paper): ensembles, normalization,

DRL from Human Preferences (Christiano et al. 17)

- Other tricks in implementation (see paper): ensembles, normalization,

- For Hopper to learn to flip 900 preferences were needed. Each preference was decided in 3-5 seconds $\rightarrow$ in approx. 1 hour they trained the agent.
- Compared with standard approach, only generating the reward function took two hours
- The behavior obtained was not so elegant (something common in Mujoco) compared with RLHF

DRL from Human Preferences (Christiano et al. 17)

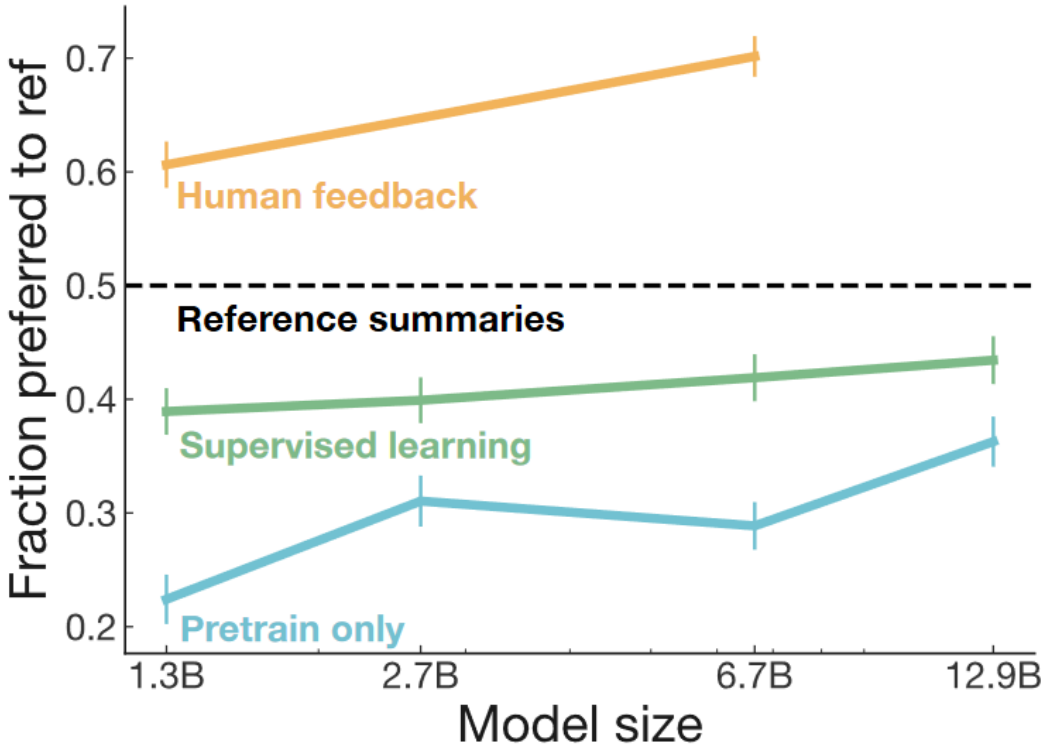
- Other tricks in implementation (see paper): ensembles, normalization,
- For Hopper to learn to flip 900 preferences were needed. Each preference was decided in 3-5 seconds $\rightarrow$ in approx. 1 hour they trained the agent.
- Compared with standard approach, only generating the reward function took two hours
- The behavior obtained was not so elegant (something common in Mujoco) compared with RLHF
- However, some problems: Still too many labeling due to a lot of time spent in warming the policy

Summarizing application (Stiennon et al. 20)

- *Learning to summarize from human feedback* (Stiennon et al. 20)
- Application of RLHF to NLP (generic proposal (Ziegler et al. 20))
- Task: Generate summaries of texts
- Generator of the summary is an autoregressive Transformer NN implementing a Language Model
- Procedure they propose is the following:
  1. Start from a standard language GPT-like model trained on perplexity (TL;DR prompt)
  2. The language model is trained to generate summaries using a training dataset of texts and trying to reproduce summaries generated by human experts
  3. After that, RL is used to fine tune the supervised trained model to generate better summaries

Summarizing application (Stiennon et al. 20)

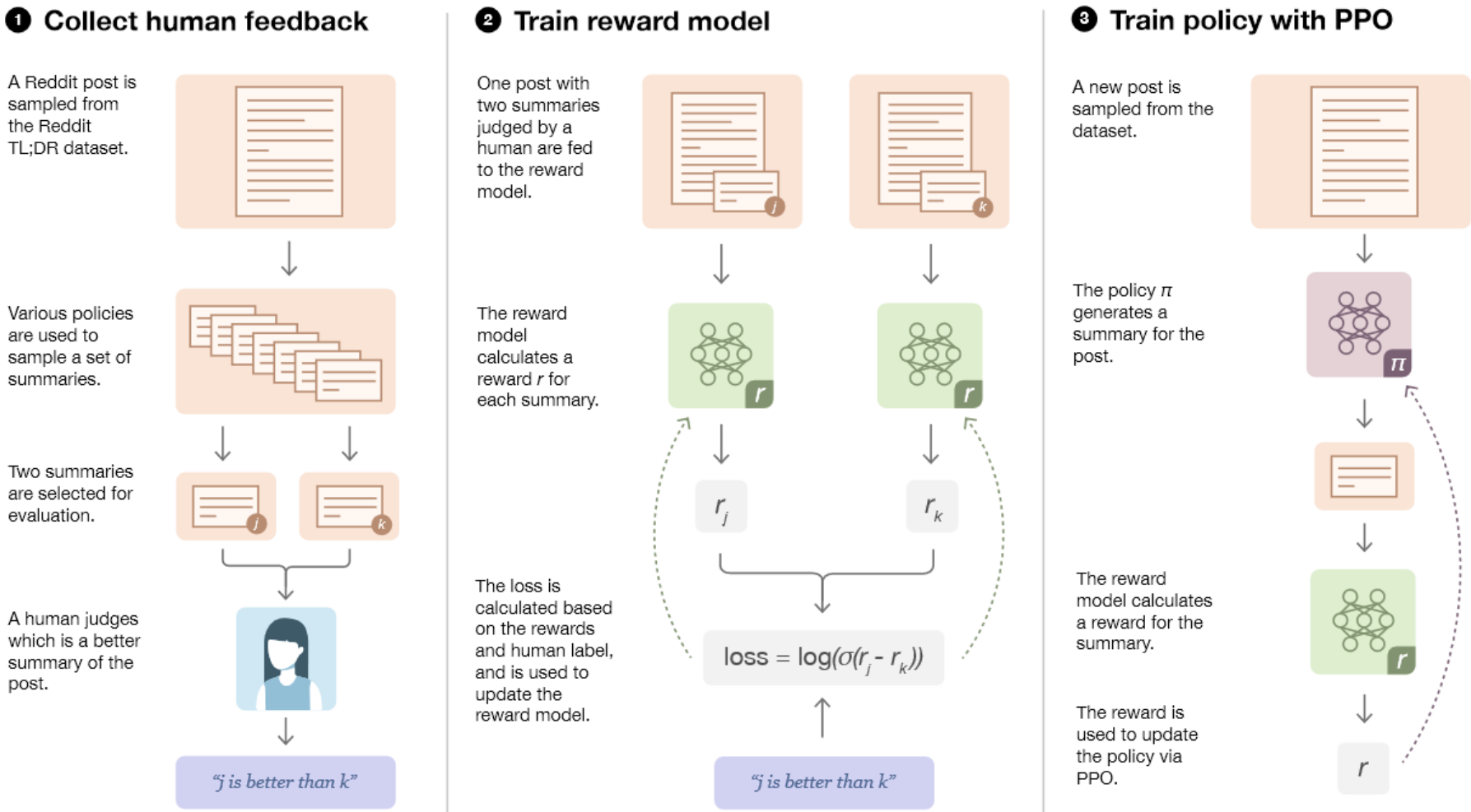
- Each stage is able to produce summaries.
- ROUGE evaluation (based on n-grams) of summaries is too simple so evaluation is done by humans



Summarizing application (Stiennon et al. 20)

- Empirically it worked better than supervised learnt model (see paper results)
- But how we obtained this result?

Summarizing application (Stiennon et al. 20)



Summarizing application (Stiennon et al. 20)

- Reward model is learnt trying to minimize the following loss:

$$\text{loss}\left(r_\theta\right) = -\mathbb{E}_{(x,y_0,y_1,i)\sim D}\left[\log\left(\sigma\left(r_\theta\left(x,y_i\right) - r_\theta\left(x,y_{1-i}\right)\right)\right)\right]$$

where $r_\theta(x,y)$ is reward model with parameters θ for text x and summary y. D is the dataset of human judgments.

Summarizing application (Stiennon et al. 20)

- Reward model is learnt trying to minimize the following loss:

$$\text{loss}\left(r_\theta\right) = -\mathbb{E}_{(x,y_0,y_1,i)\sim D}\left[\log\left(\sigma\left(r_\theta\left(x,y_i\right) - r_\theta\left(x,y_{1-i}\right)\right)\right)\right]$$

  where $r_\theta(x,y)$ is reward model with parameters θ for text x and summary y. D is the dataset of human judgments.

- However, PPO is trained in this another reward function:

$$R(x,y) = r_\theta(x,y) - \beta \log\left[\pi_\phi^{\mathrm{RL}}(y \mid x)/\pi^{\mathrm{SFT}}(y \mid x)\right]$$

  where the substracting term refer to the KL divergence between learnt policy π_ϕ^{RL} and initial policy with supervised fine learning π^{SFT} over GPT-like model

- Again, to avoid RL policy to take profit of flaws in the reward function

Summarizing application (Stiennon et al. 20)

- Some observations:
  - No warmup problem
  - No "human in the loop" (so need for KL divergence)
  - No concept of state, huge state space (50k actions)
  - **Summaries aligned with human preferences**

Summarizing application (Stiennon et al. 20)

- Some observations:
  - No warmup problem
  - No "human in the loop" (so need for KL divergence)
  - No concept of state, huge state space (50k actions)
  - **Summaries aligned with human preferences**
- Can you see what comes next?

InstructGPT (Ouyang et al. 22)

- *Training language models to follow instructions with human feedback* paper describes InstructGPT version of ChatGPT (text-davinci-002) we use nowadays
- It uses RLHF to align language model (GPT-3) to do what humans ask.
- A lot labeling of data done.
- Follow the three steps methodology: Supervised + Learn reward model + Train language model to generate reward

InstructGPT (Ouyang et al. 22)

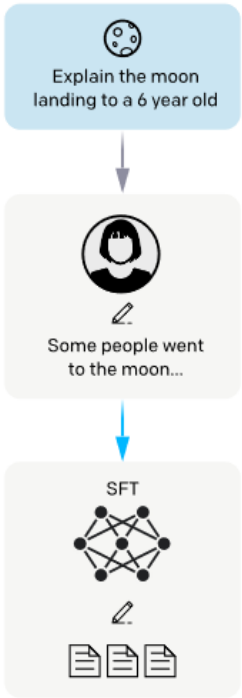
Step 1

Collect demonstration data, and train a supervised policy.

A prompt is sampled from our prompt dataset.

A labeler demonstrates the desired output behavior.

This data is used to fine-tune GPT-3 with supervised learning.



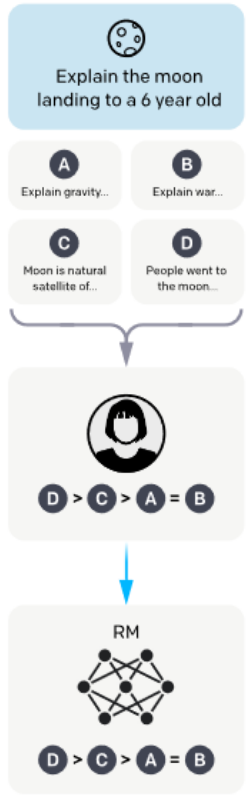
Step 2

Collect comparison data, and train a reward model.

A prompt and several model outputs are sampled.

A labeler ranks the outputs from best to worst.

This data is used to train our reward model.



Step 3

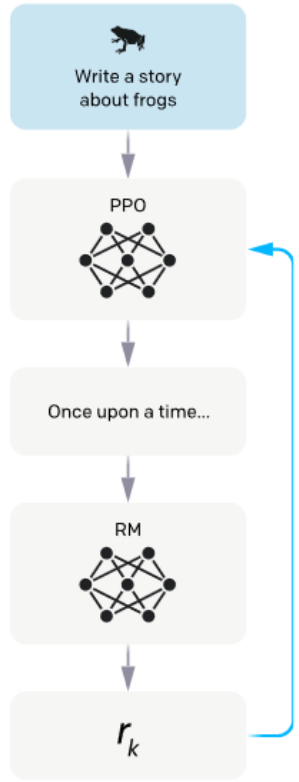
Optimize a policy against the reward model using reinforcement learning.

A new prompt is sampled from the dataset.

The policy generates an output.

The reward model calculates a reward for the output.

The reward is used to update the policy using PPO.



InstructGPT (Ouyang et al. 22)

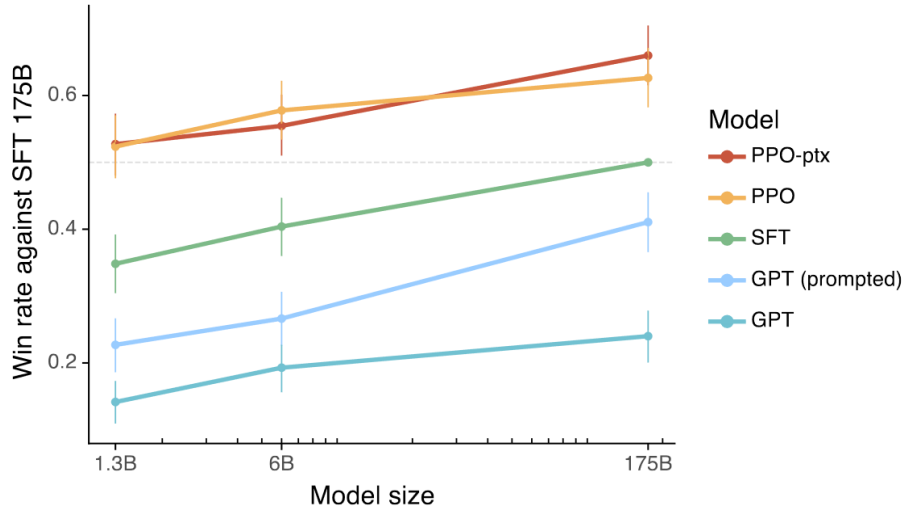
- Not a lot of details on how the RL is trained.
- Only some small differences:
  - Amount of answers for each prompt (more data for training)
  - In a variation of PPO (PPO-ptx), they add in the loss an extra term to not deviate further away from dataset used for training

$$\text{objective}(\phi) = \mathbb{E}_{(x,y)\sim D_{\pi_\phi^{\mathrm{RL}}}} \left[r_\theta(x,y) - \beta \log \left(\pi_\phi^{\mathrm{RL}}(y \mid x) / \pi^{\mathrm{SFT}}(y \mid x) \right) \right] + \gamma \mathbb{E}_{x\sim D_{\text{pretrain}}} \left[\log \left(\pi_\phi^{\mathrm{RL}}(x) \right) \right]$$

- They still do not iterate the RLHF

InstructGPT (Ouyang et al. 22)

- Results:



- Awesome results.
- Team of 40 labelers
- Small costs of finetuning compared to train GPT3
  - 3600 petaflops-day to train GPT3
  - 5 petaflops-day for SL + 60 petaflops-day to RL fine-tuning

Final words about RL on Language models

- ChatGPT and InstructGPT are not the only trained models using this approach (Claude, Mistral, Gemini).
- RL method is on-policy (why?)!
- People usually do not close the loop
- Not a lot of detail in paper! Papers start to enclose information about their research because of the possible economical impact

Some of latest developments

- Variations of RLHF:
  - **Safe RLHF** (Dai et al. 23): Combine alignment criteria that might conflict with each other sometimes
  - Reinforcement Learning from AI Feedback (**RLAIF**) (Lee et al. 23): Reduce human labelling using LLM to generate preferences. Also **Self-Rewarding Language Models** (Lee et al. 24)
  - **Constitutional AI**: (Bai et al 22)
- **Pairwise PPO**: (Tianhao Wu et al. 23)
- **DPO**: Direct Preference Optimization translates the RL problem to a classification problem (Rafailov et al. 23)
- **WARM**: Ensembles of reward functions to avoid *Reward Hacking* (Ramé et al. 24)

Subsection 1

Extended RL: towards AGI

AI and RL

- RL is well suited to the idea of AI agent
- Learns a behavior to fulfill own goals
- Is grounded in the environment
- Has the notion of optimality but with given resources (rationality)

AI and RL

- But still some problems for true AI agents
- Most important is that learning is only of one behavior defined by reward function
- When learning another task, learning has to start from scratch
- Too many interactions with env. for learning
- Not suited to the idea of long-live learning
- Some steps in solving these limitations
  - Transfer of learning
  - Multi-task learning
  - (Curriculum learning)
  - Hierarchical Learning
  - Meta-learning

Transfer RL

- Can we extend knowledge generated in one task to a different task?
- Changes in the task: different dynamics, different reward and/or different actions.
- Several ways to do that:
  - Learn one task and start with policy and values and finetune to next task
  - Randomization of the input to prepare for other scenarios or use of entropy in the policy
  - MultiTask and Meta-learning (see next slides)
  - Transfer of info from one task to the other (Q-values, policy, reward, samples, model, features, etc.)
- Example: Sharing of examples and IRL for task disentangled from actions (AIRL)
- See recent survey

Hierarchic RL

- Also helps in sparse rewards, but also useful for transfer learning.
- Natural way of learning.
- In some cases a complex task can be decomposed in simpler tasks.
- Learning is simplified when first these tasks are learnt.
- Several ways to find that:
  1. Using subrewards for subactions (reward shaping)
  2. **Discover them automatically**
- Actions can be reused to learn other tasks
- See references about the topic in course web page

Multi-task learning

- What is a task?

$$\text{A task:} \quad \mathcal{T}_i \triangleq \left\{\mathcal{S}_i, \mathcal{A}_i, p_i\left(s_1\right), p_i\left(s' \mid s, a\right), r_i(s, a)\right\}$$

- Changes in on item means a different task
- Agent does not only solve one task but several

Multi-task learning

- What is a task?

$$\text{A task:} \quad \mathcal{T}_i \triangleq \left\{\mathcal{S}_i, \mathcal{A}_i, p_i\left(s_1\right), p_i\left(s' \mid s, a\right), r_i(s, a)\right\}$$

- Changes in on item means a different task
- Agent does not only solve one task but several

Multi-task RL

- Given: a set of training tasks
- Goal: Learn a policy that can solve different tasks

Multi-task learning

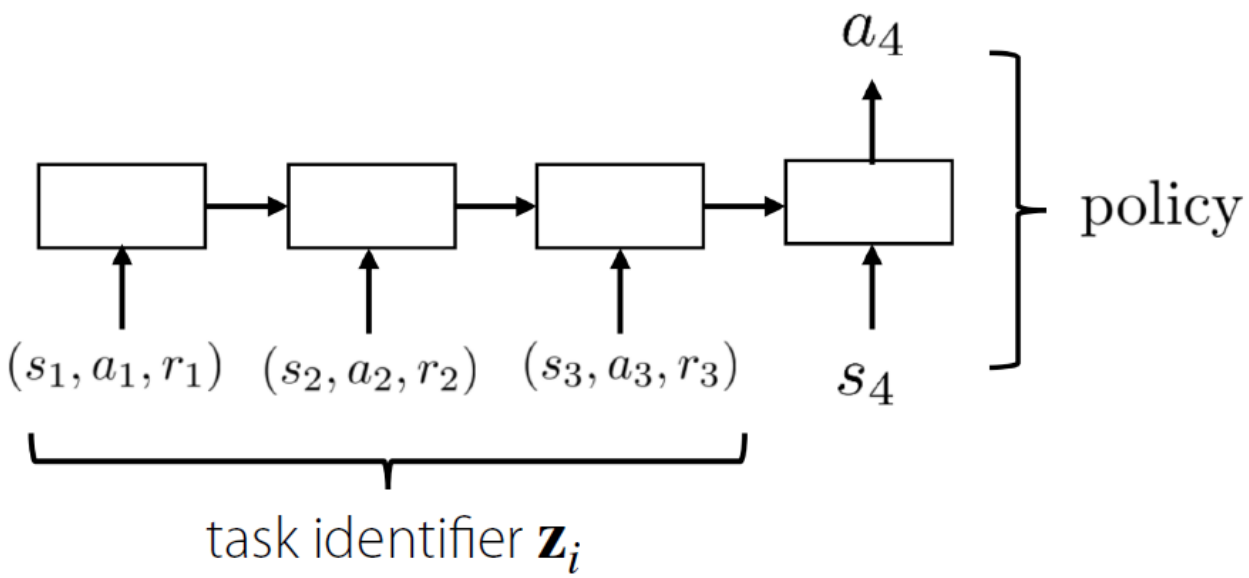
- Optimize learning/performance across all tasks through shared knowledge.
- Conditioned policies (see previous lecture) are a kind of Multi-task RL where each task is characterized by a goal state
- In Multi-task, we have conditioned policy again, but more general.

$$\pi(s, z)$$

where z is indicative of the task

Multi-task learning

- Hmmm. But indication of task is reward function or dynamics or starting state
- In some formulations the agent has to discover the scenario from the rewards he obtain
- This means use of memory:



Meta-Learning in RL

- Similar to multi-task learning but different focus and procedure

Meta RL

- Given: a set of training tasks
- Goal: Learn to solve those task and also can be learn *efficiently* new tasks

- Formulation: Given a set of training tasks, learn a policy that can also be applied successfully (directly or after small finetuning) to a set of testing tasks.

Meta-Learning in RL

- Procedure to learn is done as follows:
  1. Sample one task $\mathcal{T}_i$ from set of training tasks
  2. Generate N episodes for task $\mathcal{T}_i$ with policy
  3. Store data in ED for $\mathcal{T}_i$
  4. Update policy to maximize discounted return for all tasks.
- Focus on efficiently learn a set of different tasks.
- Learn-to-learn idea at the beginning, but extended also to generalization between tasks: Learning of each task has to be consistent and (hopefully) helpful for learning other tasks.
- Very popular in two last years, in ML and RL in particular (See course CS330 from Stanford here)
- See also a specific introduction and review of latest approaches for RL

Comparison meta and multi RL

Multi-Task Learning

Learn to solve a set of tasks.

learn tasks

perform tasks

Meta-Learning

Given i.i.d. task distribution,
learn a new task efficiently

learn to learn tasks

quickly learn
new task

Life long learning in RL

- Our agents may not be given a large batch of data/tasks right off the bat!
- Tasks are presented in sequence:

- Also called sequential learning, continual learning or incremental learning
- No forgetting part absent in Meta-RL

Links with neuro-physiology

- In Artificial Intelligence it is important to compare techniques with actual techniques used by humans and animals
- Reinforcement learning has its roots in intuitive idea of learning in animals.
- Lately, a lot of papers support that RL is implemented in the brain
  - Link of RL with actual learning in brain
  - Dopamine as reward
  - Dopamine implements TD error.
  - Support for dopamine acting as distributed value estimation

Some succesful aplications of RL

Applications

- Games: Backgammon, Go, Chess (competition), Star-Craft, Dota 2, Poker (Pluribus)
- Robotics: Walking, Manipulation (also here), etc.
- Medicine: Review. Example: Sepsis treatment, or ventilation
- Drug dessign: For instance here.
- Physics
- Recommender systems
- Finances
- Optimization in general, f.i control power, or for IoT,
- Mathematics: For instance, Logic profs
- Natural Language processing: Summarizing texts.

Conclusion

- Most of ML methods are based on data (supervised or unsupervised).
- Many models have already been trained with "all the internet".
- Data are being increasingly produced by AI. 2022 was probably the last year of the "human internet".
- RL only relies on a reward function: i.e., it could potentially learn from data itself produces.
- Could RL be a method to keep learning when AI will be constrained by "human data"?