CS 224R: Deep Reinforcement Learning (Spring 2023)

Leni Aniva

2024-03-20

- Instructors: Chelsea Finn, Karol Hausman
- Quarter: Spring 2023
- Time: MW 16:30 - 17:50
- Location: Gates B1
- Website: `https://cs224r.stanford.edu/`
- Discussion: Ed via Canvas
- Email: cs224r-spr2223-staff@lists.stanford.edu
- Office Hours: See schedule
- Grading: 50% Homework (lowest will be 5%, others are 15% each) and 50% Project

  Assignments:

  1. Imitation Learning
  2. Online reinforcement learning
  3. Offline reinforcement learning
  4. Goal-conditioned and meta reinforcement learning
  5. Project

  Project:

  - Research-level project of your choice
  - Groups of 1-3 students
  - Can share with other classes with higher expectation
  - Same late day policy as homework
  - No late days for poster, 16:00 - 19:00, 7 June
  - A word of warning: Deep RL methods take a long time to learn behaviour
- 6 late days total across homeworks and projects. Max 2 late days per assignment.

Pre-requisites:

- Machine learning: CS229 or equivalent

  We will assume knowledgeo of SGD, cross-validation, calculus, probability theory, linear algebra
- Some familiarity with deep learning: Backpropagation, convolutional networks, recurrent networks
- Assignments will require training with PyTorch

- Some familiarity with reinforcement learning
  We'll quickly go over the basics

Topics:

1. Imitation Learning: Behaviour cloning, Inverse RL
2. Model-free deep RL: Policy gradients, Actor-critic methods, Q-learning
3. Model-based deep RL
4. Offline RL: Conservative methods, decision transformers
5. Multi-task and meta RL: Hindsight relabeling, learning to explore
6. Advanced topics: hierarchical RL, sim2real, reset-free RL
7. Case studies of interesting and timely applications

Contents

1 Introduction

Announcements

- 04-05: Homework 1 out
- 16:30 04-06. (Skilling Auditorium): PyTorch tutorial will be recorded
- 04-19: Homework 1 due

Reinforcement learning solves sequential decision-making problems: A system needs to make multiple decisions based on stream of information. The solutions

- Imitation learning
- Offline and online RL
- Model-free and model-based RL
- Multi-task and meta RL

Excursion: Supervised Learning

In supervised learning, we are given labeled data $\{(x, y)\}$ and want to learn $f(x) \simeq y$. We directly receive information about what to output, and the inputs are i.i.d.

In comparison, in **Reinforcement Learning**, we learn *behaviour* $\pi(a|s)$. The data is not i.i.d., since each action affects future observations.
Open problems:

- Can robots learn *generalisable behaviour*?
- Can robots learn behaviours *autonomously*?

Why study deep reinforcement learning? Sequential decision-making problems are everywhere, and the advancements of deep neural networks have achieved considerable success in recent years. Applications:

1. Controlling robots and autonomous vehicles
2. AI that interact with people
3. Systems with feedback loops
4. Some tasks' objectives are not accuracy or differentiable

2 Imitation Learning

- *Deep Imitation Learning for Complex Manipulation Tasks from Virtual Reality Teleoperation*. Zhang et al. (2017)

Announcements

- 04-05: Homework 1 out
- 16:30 04-06. (Skilling Auditorium): PyTorch tutorial
- 04-07: Fill out AWS form with account id
- Up to 2% extra credit for providing TA endorsed answers on Ed
- Project: Final groups are not locked in until project proposals are due, survey due 04-17

We start by a formalisation of behaviour. An intelligent agent observes some environment, takes action, and the action updates the environment.
A model which decides which action to take in response of an environment is the **policy**, denoted $\pi_\theta(a|s)$, where a is the **action** and s is the **state**. It is often useful to output a distribution of actions rather than a single action, and hence π_θ here is a distribution. The interaction can be described as a cycle:

1. Observe state s_t
2. Take action a_t e.g. by sampling $a_t \sim \pi_\theta(\cdot|s_t)$
3. Observe next state s_{t+1} sampled from unknown world **dynamics** $p(\cdot|s_t, a_t)$

The result is a trajectory $s_1, a_1, \ldots, s_T$, also called a **policy roll-out**. Since the next state is only a distribution parameterised by s_t, a_t, the trajectory is **Markovian**.

Question: Is the policy being changed as we go through the trajectory or we plot the entire trajectory and then update the policy?

Dependent on the algorithm. Some policies update in the middle of a roll-out.

Question: Does the state include the history of actions?

The state includes only the previous action but the history is encoded here.
In some cases (e.g. dialogue) it would be beneficial to include history in the state and policy.

Imitation learning refers to train a policy using suprevised learning to imitate some data provided.

- Data: The data, **demonstrations**, is the set of given trajectories collected by an expert $\mathcal{D} := \{(s_1, a_1, \ldots, s_T)\}$. This does not have to be optimal.
- Training: Train policy to mimic expert using a *cross-entropy loss*

$$\arg\max_\theta \mathbb{E}_{(s,a)\sim\mathcal{D}}[\log \pi_\theta(a|s)]$$

This is a differentiable function w.r.t. θ (given differentiable π_θ), so it could be trained with gradient descent.

Excursion: Cross-entropy loss

The cross-entropy loss here is also called ℓ^2-loss. This has to do with the Gaussian distribution

$$f(x) \propto \exp\left(-\frac{1}{2\sigma^2}\left\|\boldsymbol{x}-\boldsymbol{\mu}\right\|^2\right)$$

Taking logarithm of this, we have

$$\log f(x) = -\frac{1}{2\sigma^2}\left\|\boldsymbol{x}-\boldsymbol{\mu}\right\|^2 + C$$

When σ is fixed, the difference here is a ℓ^2 distance between $\boldsymbol{x}$ and $\boldsymbol{\mu}$.

2.1 Collecting Data

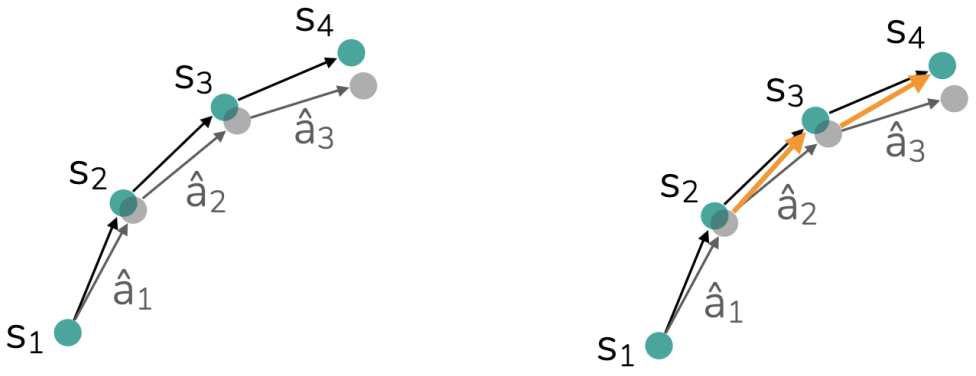
How can we collect demonstrations?

- (Robotics) Videos of expert demonstrations
- Execute a lot of simulations
- (Games) Records of human players
- (Robotics) Kinesthetic teaching: Forcing the robot arm to move and record the movement
  - \+ Easy interface
  - \- Human controller visible in scene
- (Robotics) Remote controllers:
  - ~ Interface is different
- (Robotics) Puppeteering: Use one robot to follow the other
  - \+ Easy interface
  - \- Requires double hardware

In some domains, demonstration data have already been collected. e.g. driving, writing text messages. In some domains it may not be viable to collect demonstrations. e.g. quadruped robot, or when we don't even know what is the best behaviour?

Question: Are GANs used for data generation in robotics?

There are some works that use generative models for data augmentation.



(a) Compounding errors (b) Corrective behaviour data

Figure 2.1: Compounding errors leading to state drifting out of data

Figure 2.2: Multiple paths from point A to B exist, but the optimal policy under ℓ^2-loss converges to the prohibited middle path. This is because the distribution is unimodal.

Can we directly use videos of people and animals? There is a **embodiment gap**: Differences in appearance, physical capabilities, degrees of freedom, make it challenging to learn from video data.

Question: Can we attach sensors to the expert to collect motion data?

Yes. This can be quite useful. Sometimes there is still a physical embodiment gap. See *Data Glove*.

2.2 What can go wrong?

Problems:

1. **Compounding errors**: Accumulation of errors can lead to the state drifting away from data distribution. This is **covariate shift**.

   Solutions:

   - Collect a lot of demo data and hope for the best.
   - Collect *corrective* behaviour data

2. **Multimodal demonstration data**: Agent may fail to capture multimodal data due to constrains in policy distribution.

   Solutions:

- Use a more expressive distribution

  Mixture density network: Outputs parameters for a (Gaussian) mixture distribution

- Use another loss function which matches a single mode. The cross-entropy minimises the KL-divergence $\mathrm{D}_{\mathsf{KL}}(p_{\text{expert}} \| \pi)$, but the reverse KL divergence $\mathrm{D}_{\mathsf{KL}}(\pi \| p_{\text{expert}})$ will optimise to fit a single peak of the expert distribution.

  The problem with this approach is the reverse KL distribution is often intractable.

The expressiveness of the distribution is separate from the expressiveness of the neural network. e.g. A huge network that outputs a Gaussian is still constrained by the limited expressiveness of the Gaussian.

Excursion: Generative Models

Some other classes of generative models represent distributions in other ways. For example a model could take in $(s, \boldsymbol{z})$ where $\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{1})$.
e.g. VAEs, Diffusion models
Naïve training of a model like this would make the model ignore the noise.

3. **Miamatch in observability between expert and agent**:

   To train a chatbot, we might have example demos are scraped from conversations, but knowledges external to the conversation may exist that are visible to the expert but not the agent.

   Solutions:

   - Try to get as much contextual information to the agent as possible
   - Collect demos in a way that gives expert same information as agent

Question: What if the action space is discrete but huge (e.g. words)?

- Tokenisation
- Factorisation e.g. representing words by morphemes or letters and this becomes an *autoregressive* distribution

2.3 Learning from Online Interventions

To collect corrective behaviour data, we may collect the errored trajectories from the agent and specifically train on expert trajectories correcting from such errors. **Dataset Aggregation** (**DAgger**) is the process of

1. Roll-out learned policy $\pi_\theta : s'_1 : \hat{a}_1, \ldots, s'_T$

2. Query expert action at visited states $a^* \sim \pi_{\text{expert}}(\cdot|s')$

   This can happen during roll-out e.g. safety driver

3. Aggregate corrections with existing data $\mathcal{D} \leftarrow \mathcal{D}\{(s', a^*)\}$

4. Update policy $\min_\theta \mathcal{L}(\pi_\theta, \mathcal{D})$

This makes the model more robust since it sees a wider distribution of states. When humans are used as experts, this is **Human Gated DAgger**.

+ Data-efficient way to learn from expert

– Can be challenging to query expert when agent has control

– Does not discourage model from deviating from expert, only teaches it how to fix problems.

3 Markov Decision Processes and Policy Gradients

- *Simple statistical gradient-following algorithms for connectionist reinforcement learning.* Williams (1992)

Announcements

- 04-17: Project Survey due
- 04-19: Homework 1 due
- 04-27: Project Proposal due

3.1 Reinforcement Learning Problem

Consider the problem of robot grasping an object. A robot arm has a list of motions it can take, and the action influences what the robot will observe in the future. We can capture them in the form of a policy. Two problems:

- How do we evaluate a policy?
- How do we optimise a policy for a desired outcome?

The agent gets sensory observation at time t, o_t, and performs action a_t using a policy $\pi_\theta(a_t|o_t)$. The action affects the state s_t. If the state is *fully observed*, the policy is $\pi_\theta(a_t|s_t)$. The state has s_t all the information needed for making a decision, but the agent only has access to the partial observation o_t. It may be possible that the observation does not have the information needed to provide an optimal decision.

Object Classification	Object manipulation
Supervised learning	Sequential decision making
iid data	action affects next state
large labeled, curated dataset	what are the dataset and labels?
well defined loss function	unclear loss function

Table 3.1: Comparison between object classification (supervised learning) and object manipulation (reinforcement learning)

Excursion: Plato's Allegory of the Cave

We can perceive real objects and describe them in an abstract manner based on our observations of them. In Plato's Allegory of the Cave, the gods hold the objects we observe in a cave and we can only observe their shadow.
For example, the state may be the precise location and shape of objects, and the observation is an image formed from visual perception.

Question: If the state contains all information needed, why not spend time improve the perception system rather than the reinforcement learning algorithm?

Often the notion of a state is unclear what is in the state. It is also costly or impossible to gather some information.

The value of actions can be determined using a **reward function** $r(s, a)$. e.g. crashing the car has low reward, driving to destination has high reward. $s, a, r(s, a), p(s'|s, a)$ together define a **Markov decision process** (**MDP**).
A supervised learning task can be described as

$$\mathcal{T} := \{ p(x), p(y|x), \mathcal{L} \}$$

Loss function; Data distribution; Label distribution

In comparison, a reinforcement learning task can be described as

$$\mathcal{T} := \{ \mathcal{S}, \mathcal{A}, p(s_1), p(s'|s, a), r(s, a) \}$$

State space; Dynamics; Reward; Action space; Initial state distribution

The most expensive component to building an RL algorithm is generating the samples.

3.2 Policy Gradients

Question: Why assume Markov property instead of using LSTM?

Using LSTM is still assuming the Markov property but replacing state with all history of states. Assuming Markov property makes modeling of RL models simpler.

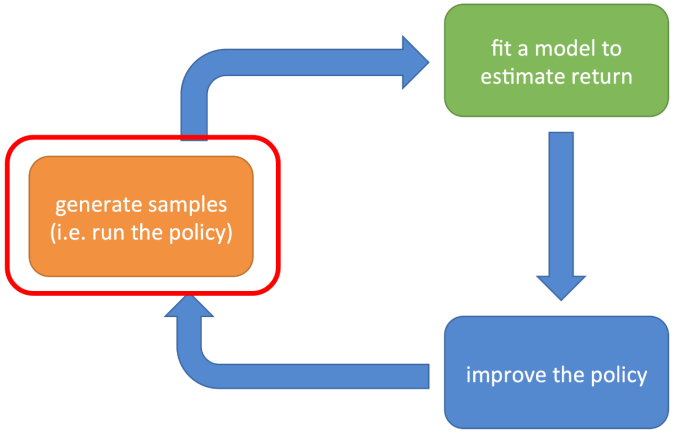


Figure 3.1: Anatomy of an RL algorithm

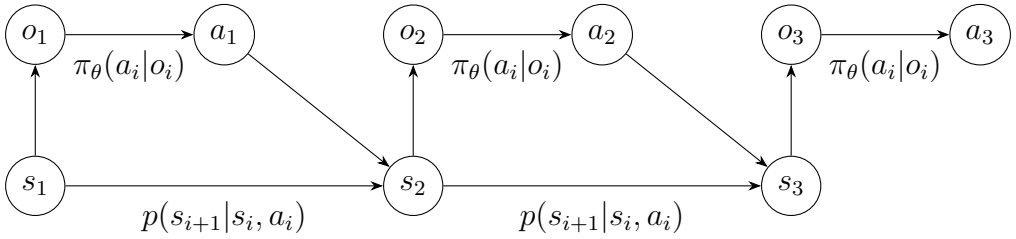


Figure 3.2: Markov chain formed by the sequences of o_i, s_i, a_i

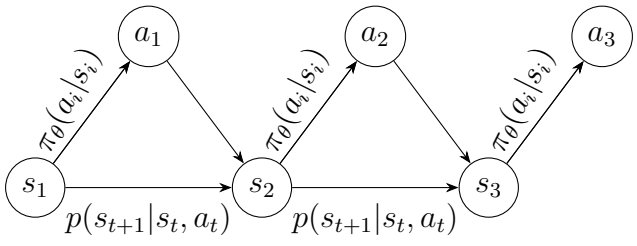


Figure 3.3: Markov chain with $o_i = s_i$

In reinforcement learning, we assume that the dynamics model $p(s_{t+1}|s_t, a_t)$, is unknown. Due to the Markov property, the trajectory distribution is

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

Question: Can we have observations coming in at different frequencies?

Yes. This is kind of an open problem. We will discuss this in hierarchical reinforcement learning where different levels receive observations at different frequencies.

The objective of reinforcement learning is to maximise reward over all trajectories:

$$\theta^* := \arg\max_\theta \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right]$$

Reward

Trajectory

To train the model, we roll-out the policy multiple times

$$J(\theta) := \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[r(\tau) \right] \simeq \frac{1}{N} \sum_i \sum_t r(s_{i,t}, a_{i,t})$$

$r(\tau) := \sum_t r(s_t, a_t)$

Number of samples

Question: How to decide how long to roll-out a policy for?

In this case we assume there is a fixed number of times to roll-out a policy, T. The infinite case requires a *discount factor* γ and will be discussed in a future lecture.

There are multiple ways to improve the policy, including Q-learning, actor-critic. Why are there so many RL algorithms? This is because of the different trade-offs in each algorithm:

- Continuous/Discrete actions
- Is it easier to learn the environment or the policy?
- Sample complexity.
- Off/On policy. **Off policy** refers to improve policy without generating new samples. **On policy** refers to generating new samples for each policy change

Policy gradient refers to improving the policy using gradient descent.

Theorem 3.1. *We have*

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

Proof. Observe that

$$J(\theta) := \mathbb{E}_{\tau \sim \pi_\theta}[r(\tau)] = \int \pi_\theta(\tau) r(\tau) \, \mathrm{d}\tau$$

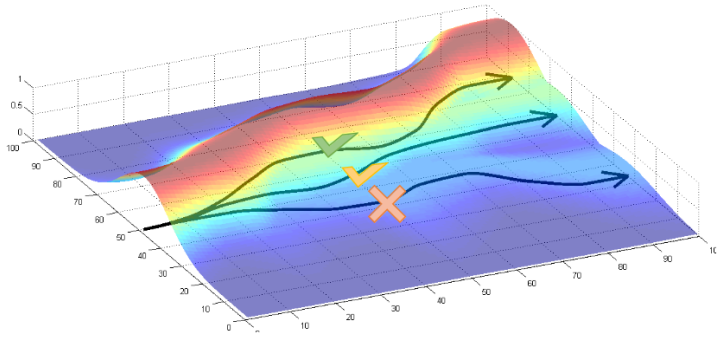


Figure 3.4: Example of policy gradient algorithm with colour/height corresponding to rewards

We can differentiate under mild conditions on π_θ and r:

$$\nabla_\theta J(\theta) = \int \nabla_\theta \pi_\theta(\tau) r(\tau) \, \mathrm{d}\tau$$

This integral is intractable and we would like it to be in a expectation form which can be approximated using Monte Carlo. To proceed further, we use the following fact

$$\frac{\mathrm{d}}{\mathrm{d}x} \log f(x) = \frac{f'(x)}{f(x)}$$

so $f(x) \frac{\mathrm{d}}{\mathrm{d}x} \log f(x) = f'(x)$.
Hence the integrand can be rewritten as

$$\nabla_\theta \pi_\theta(\tau) r(\tau) = \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) r(\tau)$$

so

$$\nabla_\theta J(\theta) = \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) r(\tau) \, \mathrm{d}\tau = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} [\nabla_\theta \log \pi_\theta(\tau) r(\tau)]$$

We can take logarithm of $\pi_\theta(\tau)$, to get

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

When this is differentiated w.r.t. θ, the terms other than π_θ become 0, therefore

$$\nabla_\theta \log \pi_\theta(\tau) r(\tau) = \left(\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) \right)$$

□

The expectation in $\nabla_\theta J(\theta)$ can be then approximated with sampling and mean to create a tractable expression. This is the **REINFORCE** algorithm:

Algorithm: REINFORCE

1. Sample τ from $\pi_\theta(a|s)$ from the policy
2. Calculate

$$\hat{\nabla}_\theta J(\theta) := \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\log\pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_{t=1}^{T} r(s_{i,t},a_{i,t})\right)$$

3. Update $\theta \leftarrow \theta + \alpha\hat{\nabla}_\theta J(\theta)$, where α is the step size.

In comparison to supervised imitation learning, whose gradient can be written as

$$\hat{\nabla}_\theta J(\theta) := \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\log\pi_\theta(a_{i,t}|s_{i,t})\right)$$

policy gradient takes into account the rewards.

Question: How to find out which action to credit when accounting for rewards?

This is the credit assignment problem. It can be mitigated by a large number of samples. We will see how to reduce the number of samples in the next few lectures.

Pros and cons of Policy Gradient:

\+ Simple

\+ Easy to combine with existing multi-task algorithms

\- Produces a *high-variance* gradient

Can be mitigated with baselines, trust regions

\- On-policy: Cannot reuse existing experience to update gradient.

Question: If only a tiny fraction of search space leads to good actions, how can we prevent gradient vanishing or a long training phase for the policy?

This is a problem with policy gradient. Usually people pre-train policy using e.g. mutation learning and then use policy gradient.

Question: During roll-out, do we take the action of highest probability?

We sample the action from $\pi_\theta(\cdot|s_t)$ instead of using $\arg\max_a = \pi_\theta(a|s_t)$.

Question: Is there guarantee about the policies we can get?

There are some guarantees in convergence.

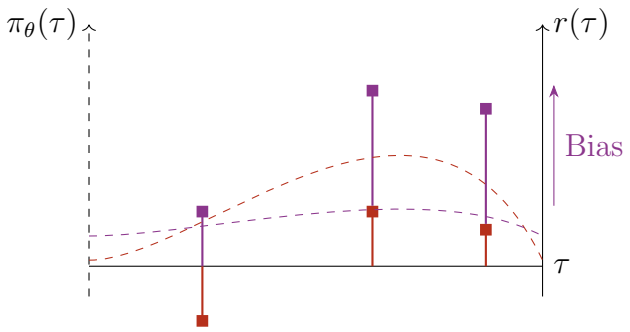


Figure 3.5: Biasing every reward $r(s, a)$ by a constant c does not bias the policy gradient estimator, but it results in a more flat distribution which needs more samples to approximate.

3.3 Variance Reduction

We have the policy gradient estimator $\hat{\nabla}_\theta J(\theta)$. We want a gradient that is as efficient as possible, i.e. has as small of a variance of a possible, since it reduces the number of samples needed to accurately estimate the gradient.
Consider if the rewards are inflated by a constant c, i.e. $r(\tau) \leftarrow r(\tau) + c$. Then we notice that the estimator for $\nabla_\theta J(\theta)$ is unbiased:

$$\mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau)(r(\tau) + c)] = \nabla_\theta J(\theta) + c\,\mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\underbrace{\nabla_\theta \log \pi_\theta(\tau)}_{\text{Score function}}] = \nabla_\theta J(\theta)$$

Since the expectation of the score function is 0. In practice, we set $c := -\frac{1}{N}\sum_{i=1}^{n} r(\tau_i)$ to minimise the variance. The intuitive explanation is that, if the critic always gives a high rating to the actions taken by the agent, we would need more samples to distinguish what the critic is expecting.

Question: Why would you add constants to the reward function?

We do not want to inflate the reward since it would increase the variance, but if we can evaluate the mean of rewards, we could subtract it from the rewards to reduce variance.

Another tool to reduce variance is the **causality trick**. We can slightly change the objective so the reward is only summed from the current step to the end to reduce variance:

$$\hat{\nabla}_\theta J(\theta) := \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T} \log \pi_\theta(a_{i,t}|s_{i,t}) \left(\underbrace{\sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'})}_{\text{Rewards "to go"}}\right)$$

Question: By doing causality trick, is the estimator still unbiased?

Yes.

4 Value-based RL and Actor-Critic Methods

- *Trust Region Policy Optimization*. Schulman et al. (2015)
- *Proximal Policy Optimization Algorithms*. Schulman et al. (2017)
- *Training language models to follow instructions with human feedback*. Ouyang et al. (2022) "Instruct GPT"
- *Solving Rubik's Cube with a Robot Hand*. Akkaya et al. (2019)

Announcements

- 04-17: Project Survey due
- 04-19: Homework 1 due
- 04-27: Project Proposal due

4.1 Value-based RL

In the causality trick, the reward to-go is the reward from a *single* trajectory of actions. We could also design the reward based on the set of *possible* future actions. We could substitute in the "true" rewards to go to get a better estimate:

$$\hat{\nabla}_\theta J(\theta) := \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\log \pi_\theta(a_{i,t}|s_{i,t})\hat{Q}(s_t,a_t)$$

$$Q^\pi(s_t,a_t) := \sum_{t'=t}^{T}\mathbb{E}_{\pi_\theta}[r(s_{t'},a_{t'})|s_t,a_t]$$

Q^π is the **Q-function**, representing the total reward from taking a_t in s_t. The **value function** is the total reward from s_t:

$$V^\pi(s_t) := \mathbb{E}_{a_t\sim\pi_\theta(a_t|s_t)}[Q^\pi(s_t,a_t)]$$

The **advantage function** measures the amount by which action a_t is better than the average reward from state s_t:

$$A^\pi(s_t,a_t) := Q^\pi(s_t,a_t) - V^\pi(s_t)$$

We fit a model to estimate Q^π, V^π, A^π, and use the estimate

$$\hat{\nabla}_\theta J(\theta) := \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\log \pi_\theta(a_{i,t}|s_{i,t})\hat{A}^\pi(s_{i,t},a_{i,t})$$

Only V^π needs to be fitted, since both Q^π and A^π can be expressed in terms of V^π:

- Q:

$$\begin{aligned} Q^\pi(s_t, a_t) &= r(s_t, a_t) + \sum_{t'=t+1}^{T} \mathbb{E}_{\pi_\theta}[r(s_{t'}, a_{t'})|s_t, a_t] \\ &= r(s_t, a_t) + \sum_{t'=t+1}^{T} \mathbb{E}_{\pi_\theta}[r(s_{t'}, a_{t'})|s_{t+1}, a_{t+1}] \\ &\simeq r(s_t, a_t) + V^\pi(s_{t+1}) \end{aligned}$$

  The last step is an approximation, since the expectation is evaluated on the stochastic distribution of $p(s_{t+1}|s_t, a_t)$, but we are replacing it by the observed next state s_{t+1}

- A:

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

The crux of this method is: If the value function changes at the next action, we can already determine the reward instead of having to roll-out the policy to the end.
We want to fit $\hat{V}^\pi(s_i)$ (with a L^2 loss, for example) to the ideal target

$$y_{i,t} := \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}[r(s_{t'}, a_{t'})|s_{i,t}]$$

We could use the **Monte Carlo target**:

$$\hat{y}_{i,t}^{\mathsf{MC}} := \sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'})$$

Or we can use the **bootstrap estimate**, which is a type of autoregressive model using the previous step's estimate:

$$\hat{y}_{i,t} := r(s_{i,t}, a_{i,t}) + V^\pi(s_{i,t+1}) \simeq r(s_{i,t}, a_{i,t}) + \hat{V}^\pi(s_{i,t+1})$$

Previous fitted value function

Question: If we don't roll-out to the very end, do we get a greedy policy that only considers near future rewards?

No since the value function considers all future rewards.

Question: Does the number of trajectories with positive rewards matter? If only few trajectories lead to positive rewards, would it skew the estimate?

If the trajectories with rewards are few, it becomes an exploration problem.

Question: Is the previous step policy π the same as the current step policy?

We use a slightly outdated version of the network for the previous step policy.

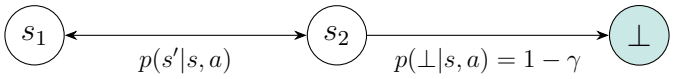


Figure 4.1: Discount factor adds a terminal state to the Markov chain with probability of $1-\gamma$ being reached in each step

If the T episode length is ∞, the agent can potentially accumulate infinite rewards. The solution is to have a **discount factor** $\gamma \in [0,1]$ (0.99 works well) and weight rewards higher sooner than later.

$$y_{i,t} \simeq r(s_{i,t}, a_{i,t}) + \gamma \hat{V}^\pi(s_{i,t+1})$$

Introducing the discount factor γ changes the MDP: It introduces a terminal state with no rewards.
The advantage function can be expressed in two ways. We can either use the value function based definition or the Monte Carlo definition:

- $\hat{A}^\pi_{\mathsf{C}}(s_t, a_t) := r(s_t, a_t) + \gamma \hat{V}^\pi(s_{t+1}) - \hat{V}^\pi(s_t)$
  - \+ Lower variance
  - \- Higher bias if value is wrong.
- $\hat{A}^\pi_{\mathsf{MC}}(s_t, a_t) := \sum_{t'=t}^{\infty} \gamma^{t'-t} r(s_{t'}, a_{t'}) - \hat{V}^\pi(s_t)$
  - \+ No bias
  - \- Higher variance due to single-sample estimate

We can combine the above two approaches by using the value function after n steps. This is **N-step rewards**:

$$\hat{A}^\pi_n(s_t, a_t) := \sum_{t'=t}^{t'+n} \gamma^{t'-t} r(s_{t'}, a_{t'}) - \hat{V}^\pi(s_t) + \gamma^n \hat{V}^\pi(s_{t+n})$$

Policy gradient can reuse old data via *importance sampling*.

Excursion: Importance Sampling

Suppose we want to evaluate

$$\mathbb{E}_{x\sim p(x)}[f(x)] = \int p(x) f(x)\, \mathrm{d}x$$

but we cannot sample from p and we can merely evaluate its density. We can sample from another distribution q and use the *importance* $p(x)/q(x)$:

$$\mathbb{E}_{x\sim p(x)}[f(x)] = \int q(x) \frac{p(x)}{q(x)} f(x)\, \mathrm{d}x = \mathbb{E}_{x\sim q(x)}\left[\frac{p(x)}{q(x)} f(x)\right]$$

The closer $p(x) \simeq q(x)$, the better the estimate.

Suppose the older iteration's parameter is θ and the new iteration parameter is θ'. We can perform importance sampling by substituting the expectation on $\pi_{\theta'}$ with one on π_θ on policy gradient

$$\begin{aligned} J(\theta') &= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\frac{\pi_{\theta'}(\tau)}{\pi_\theta(\tau)} r(\tau)\right] \\ \nabla_{\theta'} J(\theta') &= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\frac{\nabla_{\theta'}\pi_{\theta'}(\tau)}{\pi_\theta(\tau)} r(\tau)\right] = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\frac{\pi_{\theta'}(\tau)}{\pi_\theta(\tau)} \nabla_{\theta'}\log\pi_{\theta'}(\tau) r(t)\right] \\ &= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\prod_{t=1}^{T}\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \nabla_{\theta'}\log\pi_{\theta'}(\tau) r(t)\right] \end{aligned}$$

where we have used the identity

$$\frac{\pi_{\theta'}(\tau)}{\pi_\theta(\tau)} = \frac{\cancel{p(s_1)}\prod_{t=1}^{T}\pi_{\theta'}(a_t|s_t)\cancel{p(s_{t+1}|s_t,a)}}{\cancel{p(s_1)}\prod_{t=1}^{T}\pi_\theta(a_t|s_t)\cancel{p(s_{t+1}|s_t,a)}} = \prod_{t=1}^{T}\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}$$

The problem with importance distribution is that if the two distributions $\pi_\theta, \pi_{\theta'}$ are very different, the estimator becomes hard to evaluate.
The solutions of this:

- Penalise the gradient step so $\theta = \theta'$. e.g.

  $$\theta' \leftarrow \arg\max(\theta' - \theta)\nabla_\theta J(\theta), \|\theta' - \theta\|_2 \leq \epsilon$$

  Since we care about the distribution not shifting, we can instead use a divergence penaliser

  $$\theta' \leftarrow \arg\max(\theta' - \theta)\nabla_\theta J(\theta), \mathrm{D}_{\mathsf{KL}}(\pi_{\theta'}\|\pi_\theta) \leq \epsilon$$

- If the distribution changes too much, we can also perform roll-out again.

Trust Region Policy Optimisation (**TRPO**) applies all the tricks:

- Use advantage function to reduce the variance
- Use importance sampling to take multiple gradient steps
- Constrain the optimisation objective in the policy space

Proximal Policy Optimisation (**PPO**) applies all the tricks in TRPO, and penalises large policy updates using a loss function on $\pi_{\theta'}(a|s)/\pi_\theta(a|s)$. It is the optimisation used in now state-of-the-art RL models.

Example

RL with human feedback (**RLHF**) is an algorithm where the reward function is augmented by human feedback oracle.
In PPO for Large Language Models, we have a reward function $R(s;p)$ for output s to

prompt p. The reward is higher when a human prefers the output. When we train a network to approximate $R(s;p)$, it may fail to assign low scores for unnatural inputs s, so we penalise it with inputs that deviate from human language.

$$\hat{R}(s;p) := R(s;p) - \underbrace{\beta}_{\text{Hyperparameter}} \log \frac{\overbrace{p^{\mathsf{RL}}(s)}^{\text{Policy density}}}{\underbrace{p^{\mathsf{PT}}(s)}_{\text{Pre-trained model density}}}$$

The LLM is trained in 3 phases:

1. Collect human prompt and answer pairs and fine-tune GPT-3 with this data
2. A prompt and several models are sampled and a human labeler ranks the answers. The ranking is used to train a reward model.
3. Train the policy from (1) using reward function in (2).

Question: Do we encode logic and other aspects in the reward function $\hat{R}$?

No.

5 Q-Learning

- *Playing Atari with Deep Reinforcement Learning*. Mnih et al. (2013)

Announcements

- 04-17: Project Survey Due
- 04-19: Homework 1 Due, Homework 2 Out

5.1 Actor-Critic Methods

The value formulation gives rise to the **actor-critic** algorithm. The policy π_θ is the **actor** and the advantage function $\hat{A}^\pi$ is the **critic**.

Algorithm: Online Actor-Critic

1. Sample action $a \sim \pi_\theta(a|s)$, get (s, a, s', r)
2. Update $\hat{V}^\pi$ using target $r + \gamma \hat{V}^\pi(s')$
3. Evaluate $\hat{A}(s, a) = r(s, a) + \gamma \hat{V}^\pi(s') - \hat{V}^\pi(s)$
4. Evaluate $\hat{\nabla}_\theta J(\theta) := \nabla_\theta \log \pi_\theta(a|s) \hat{A}^\pi(s, a)$

5. Update $\theta \leftarrow \theta + \alpha \hat{\nabla}_\theta J(\theta)$

5.2 Policy and Value Iterations

The only reason we want the advantage, value, and Q functions is because we want a better estimate of the rewards to go. However, we could extract more information from them. The Q-function represents the *sum of estimated future rewards*, and we could engineer this function into a policy.
Suppose we have a policy $\pi(a|s)$. We can look at the advantage function $A^\pi(s_t, a_t)$ to find a better action a' than the average action given by π: $\arg\max_{a_t} A^\pi(s_t, a_t)$. Taking this argmax is *at least as good* as a random action drawn from the policy $\pi(a_t|s_t)$ no matter what the policy is. Hence the new policy can be

$$\pi'(a_t|s_t) := \mathbb{1}\{a_t = \arg\max_a A^\pi(s_t, a)\} \tag{5.1}$$

Algorithm: Policy Iteration

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

The $A^\pi(s, a)$ can be evaluated as

$$A^\pi(s, a) := r(s, a) + \gamma\, \mathbb{E}_{p(s'|s,a)}[V^\pi(s')] - V^\pi(s)$$

Since $V^\pi(s)$ is independent on a, it does not factor into the $\arg\max$, so we can replace A^π by Q^π

$$\arg\max_a A^\pi(s_t, a) = \arg\max_a Q^\pi(s_t, a)$$

with

$$Q^\pi(s, a) := r(s, a) + \gamma\, \mathbb{E}[V^\pi(s')]$$

With this we can skip policy update function entirely.

Algorithm: Policy Iteration

1. Evaluate $Q^\pi(s, a)$
2. Set $\pi \leftarrow \pi'$

Since we can just let policy be $\arg\max_a Q(s, a)$, we can skip policy entirely. This gives the **Value Iteration** algorithm

Algorithm: Value Iteration

1. $Q(s,a) \leftarrow r(s,a) + \gamma\, \mathbb{E}_{p(s'|s,a)}[V(s')]$
2. $V(s) \leftarrow \max_a Q(s,a)$

We can fit the policy function using some MSE loss:

$$L(\phi) := \|V_\phi(s) - y\|\,, \qquad y := \arg\max_a r(s,a) + \gamma\, \mathbb{E}_{p(s'|s,a)}[V_\phi(s')]$$

The evaluation of $\max_a Q(s,a)$ maybe intractable, since the dynamics model generating the state s' is unknown to us and evaluating this maximum requires evaluation of all counterfactual a's. We cannot predict the reward provided by taking the action $V_\phi(s)$.

5.3 Q-Learning

The solution to the intractable dynamics model is by fitting the $\mathsf{Q}_\phi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ function instead of the value function.

Algorithm: Q-Learning

1. Sample $\{(s_i, a_i, s_i', r_i)\}$ using some policy, where s_i' is the next state
2. $y_i \leftarrow r(s_i,a_i) + \gamma\, \mathbb{E}[V_\phi(s_i')] \simeq r(s_i,a_i) + \gamma \max_{a_i'} \mathsf{Q}_\phi(s_i', a_i')$
3. Fit $\mathsf{Q}_\phi(s_i,a_i)$ to y_i

Now the Q-function is dependent on the action a, so we do not need to simulate the dynamics model.

$$\mathsf{Q}_\phi(s,a) = r(s,a) + \gamma \max_{a'} \mathsf{Q}_\phi(s',a')$$

Question: What would happen if we have a continuous action space?

We can perform some optimisation. We will cover this in the next lecture.

Question: How do we get s'?

s' is the state the system will be in after taking action a (not a'). This fixes the state to the particular action a.

For an optimal policy $\pi^\star$ derived from Equation 5.1, it satisfies the **Bellman Equation**

$$Q^\star(s_t,a_t) = \mathbb{E}_{s_{t+1}\sim p(s_{t+1}|s_t,a_t)}\left[r(s_t,a_t) + \gamma \max_{a'} Q^*(s_{t+1},a')\right]$$

Q-Learning essentially learns a table of optimal actions to take for every single possible state.

Algorithm: Fitted Q-iteration

1. Collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy, where s'_i is the next state.
2. $y_i \leftarrow r(s_i, a_i) + \gamma \max_{a'_i} \mathsf{Q}_\phi(s'_i, a'_i)$ (Bellman Equation)
3. $\phi \leftarrow \arg\min_\phi \frac{1}{2} \sum_i \|\mathsf{Q}_\phi(s_i, a_i) - y_i\|^2$
4. Repeat (2) – (3)
5. $\pi(a|s) := \arg\max_a \mathsf{Q}_\phi(s, a)$

+ Q-Learning is an *off-policy* algorithm.
+ More efficient than on-policy methods
+ Can update the policy without seeing the reward. The algorithm does not have to experience the reward in any trajectory and decide by looking up the next step Q_ϕ
+ Easy to parallelise
- Lots of tricks to make it work
- Potentially harder to learn than just a policy

Question: Is there a way to structure the neural net such that its easy to sample the argmax?

If $\mathcal{A}$ is discrete, we could either use the action as a part of the input, or have multiple outputs as the logits of a categorical distribution. The latter usually works better.
If $\mathcal{A}$ is continuous, we could have the model output a scalar $\mathsf{f}(s, a)$ ...

6 Practical Deep RL Implementations

- *Deep Reinforcement Learning with Double Q-learning*. Hasselt et al. (2015)
- *QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation*. Kalashnikov et al. (2018)

Announcements

- 04-19: Homework 1 Due, Homework 2 Out
- 04-26: Project Proposal due

6.1 Q-Learning Tricks

Q-Learning is not gradient descent algorithm. Fitting the Q-function requires a gradient descent but the Q-iteration step, which evaluates y_i using Bellman Equation, does not use gradient descent. We can run Q-learning online:

Algorithm: Online Q-iteration

1. Rollout policy to get $\{(s_i, a_i, s_i', r_i)\}$
2. $y_i \leftarrow r(s_i, a_i) + \gamma \max_{a_i'} \mathsf{Q}_\phi(s_i', a_i')$
3. $\phi \leftarrow \arg\min_\phi \frac{1}{2} \sum_i \|\mathsf{Q}_\phi(s_i, a_i) - y_i\|^2$
4. Repeat (2) – (3)

Potential problems in Q-learning:

- Correlated samples in online Q-learning:

  If we take action a_i and observe (s_i, a_i, s_i', r_i), this is not iid data and sequential data are strongly correlated. Having strongly correlated data from *the same trajectory*, the fitted Q-function would only work for the tiny snippet of trajectory and not be able to generalise.

  One solution is **Replay Buffers**, a set $\mathcal{B}$ which stores historical trajectories. Usually $\mathcal{B}$ is large to decorrelate the trajectories

  Algorithm: Online Q-iteration with Replay Buffer

  1. Sample $\{(s_i, a_i, s_i', r_i)\}$ from dataset $\mathcal{B}$, a *replay buffer* initially populated by rolling out some policy.
  2. $y_i \leftarrow r(s_i, a_i) + \gamma \max_{a_i'} \mathsf{Q}_\phi(s_i', a_i')$
  3. $\phi \leftarrow \arg\min_\phi \frac{1}{2} \sum_i \|\mathsf{Q}_\phi(s_i, a_i) - y_i\|^2$
  4. Repeat (2) – (3) for k iterations (commonly $k = 1$)
  5. Rollout the Q-learning based policy and update $\mathcal{B}$

- The target of fitting is constantly moving

  The step $\phi \leftarrow \arg\min_\phi \frac{1}{2} \sum_i \|\mathsf{Q}_\phi(s_i, a_i) - y_i\|^2$ is a well-defined stable regression step. Step (2) moves the target of regression.

  One solution is **Target Networks**. We save an earlier version ϕ' of ϕ and fit the current step Q_ϕ function to the target y_i created from $\mathsf{Q}_{\phi'}$ With this change, the step of fitting ϕ becomes a supervised regression problem

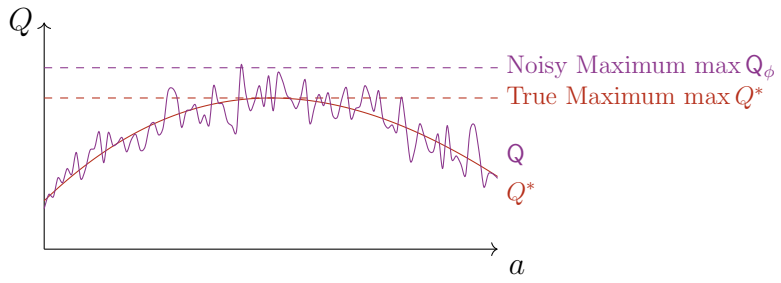


Figure 6.1: The maximum obtained from Q-learning overshoots the true maximum because it is evaluated from the imperfect noisy neural network output

Algorithm: Classic Deep Q-Learning (DQN)

1. Save target network parameters ϕ' (every N steps)
2. Sample $\{(s_i, a_i, s'_i, r_i)\}$ from dataset $\mathcal{B}$, a *replay buffer* initially populated by rolling out some policy.
3. $y_i \leftarrow r(s_i, a_i) + \gamma \max_{a'_i} \mathsf{Q}_{\phi'}(s'_i, a'_i)$
4. $\phi \leftarrow \arg\min_\phi \frac{1}{2} \sum_i \|\mathsf{Q}_\phi(s_i, a_i) - y_i\|^2$
5. Repeat (2) – (4) for k iterations (commonly $k = 1$)
6. Rollout the Q-learning based policy and update $\mathcal{B}$

Is Q-Learning accurate? The learned Q function Q_ϕ may overestimate the rewards since each step of Q_ϕ is a maximum of the noisy Q_ϕ of the next step.

Excursion:

Imagine if we have two random variables X_1, X_2, then we have

$$\mathbb{E}[\max(X_1, X_2)] \geq \max(\mathbb{E}[X_1], \mathbb{E}[X_2])$$

This is because the two separate expectations eliminate the dependent component between the two variables.

We can curb this effect using **Double Q-Learning**, which is a modification of target network with:

$$y \leftarrow r + \gamma \mathsf{Q}_{\phi'}(s', \arg\max_{a'} \mathsf{Q}_\phi(s', a'))$$

where the value is samples from one Q-function (target network) and action from another Q-function (current network).
In online Q-learning algorithm, the agent may get stuck in some state. In order to encourage exploration we may use the following policy instead. This is **ϵ-greedy**

exploration

$$\pi(a_t|s_t) := \begin{cases} 1-\epsilon & a_t = \arg\max_{a_t} \mathsf{Q}_\phi(s_t, a_t) \\ \epsilon/(|\mathcal{A}|-1) & \text{otherwise} \end{cases}$$

A common trick starts with higher ϵ and gradually drop to encourage exploration.

7 Model-Based RL

- *Deep Dynamics Models for Learning Dexterous Manipulation*. Nagabandi et al. (2020)
- *When to Trust Your Model: Model-Based Policy Optimization*. Janner et al. (2019)

Announcements

- 04-26: Project Proposal due; (Graded Lightly)
- 05-03: Homework 2 due

7.1 Gradient-based and Sampling-based Optimisation

Two common classes of optimisation methods exist.

- **Gradient-based optimisation** or **first-order optimisation** relies on local gradient to guide the next sample
  - \+ Scalable to high dimensions
  - \+ Works well especially in overparameterised regimes
  - \- Requires nice optimisation landscape
- **Sampling-based optimisation** or **zeroth-order optimisation** relies on sampling heuristic regions
  - \+ Parallelisable
  - \+ Requires no gradient information
  - \- Scales poorly to high dimensions

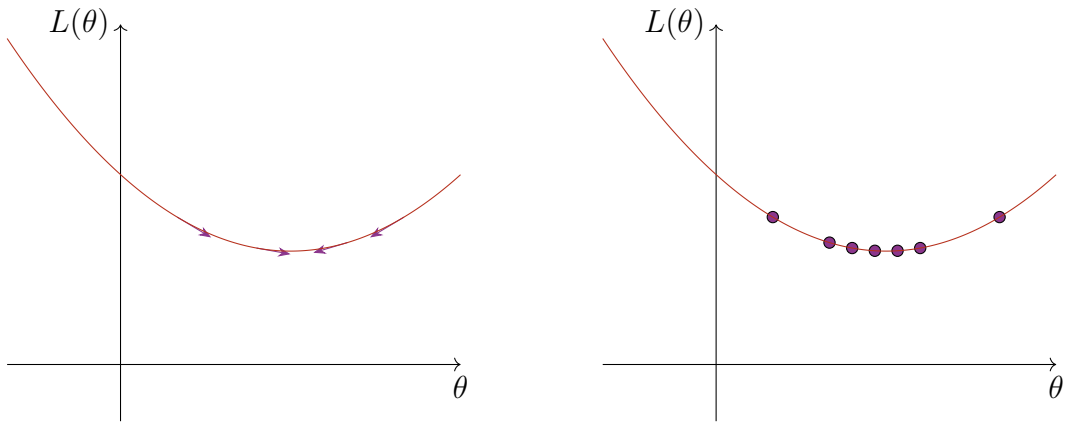
Question: Is sampling-based optimisation more prone to getting stuck in local minima?

In general it is not more prone.

Question: Do people use hybrid methods?

Yes

A sample-based method is the **Cross-Entropy Method** (**CEM**).



(a) Gradient-based optimisation (b) Sample-based optimisation

Figure 7.1: Gradient and Sampling based optimisations

Algorithm: Cross-Entropy Method

1. Sample from distribution $p_i(\theta)$
2. Rank samples according to loss $\theta_1, \ldots, \theta_K$
3. Fit Gaussian distribution p_i to "elite" samples $\theta_1, \ldots, \theta_K$

Question: Why is this called the Cross-Entropy Method

Step 3 fits the theoretical distribution to the empirical distribution using cross entropy.

7.2 Learning a Dynamics Model

It would be useful if we could simulate the real world in reinforcement learning. Given samples s_i, s_i', a_i, we can fit a dynamics model $\mathsf{p}_\phi(s, a) \mapsto s'$:

$$\min_\phi \sum_i \|\mathsf{p}_\phi(s_I, a_i) - s_i'\|^2$$

and use such a learned dynamics model to train the reinforcement learning algorithm. When such a model exists, we could optimise the policy using

$$\hat{\theta} := \arg\max_\theta \max \sum_{t=t_0}^{t_0+H} r(s_t, a_t) + V(s_{t_0+H+1})$$

Terminal function

where the $\{(s_t, a_t)\}_{t=t_0}^{H}$ sequence is obtained by rolling out the policy π_θ on the dynamics model p_ϕ. H is the planning horizon. The terminal function is there to partially mitigate the short-sightedness problem, but it introduces the problem of having to train V.

Algorithm: Model-Based Reinforcement Learning

1. Collect data $\mathcal{D}$ by rolling out π
2. Train model $\mathsf{p}_\phi(s'|s,a)$ using $\mathcal{D}$
3. Optimise the objective function $\sum_t r(s_t, a_t)$ using gradient (backpropagation through time) or sample based optimisation
4. Execute planned actions and append the visiting tuples (s, a, s') to $\mathcal{D}$. This is to discourage the dynamics model from diverging from reality.

Question: Is the horizon H much shorter than the actual horizon?

Yes. This is a drawback and could cause the model to be greedy.

This method could fail due to inaccuracy in p_ϕ. The policy may try to exploit the inaccuracy in the model.

7.3 Using the Learned Dynamics Model

Another approach is to *plan and re-plan* using the model. We optimise the action sequence $a_{t:t+H}$ using p_ϕ and execute the *first planned action* a_t. To reduce cost, the re-planning can be done every 5 or 10 steps instead of just 1. At this point it is important to note the distinction between open and closed loop algorithms

- **Open-Loop**: Agent sends a complete action sequence into the world without regards of the updated state.
- **Closed-Loop**: Agent sends action into world and world feeds action back into agent

Algorithm: Model-Predict Control

1. Rollout base policy $\pi_0(a_t|s_t)$ (e.g. random) to collect data $\mathcal{D} = \{(s, a, s')\}$
2. Learn dynamics model p_ϕ
3. Use model p_ϕ to optimise action sequence $a_{t:t+H}$
4. Execute the first planned action and observe state s'
5. Append (s, a, s') to dataset $\mathcal{D}$

\+ Re-plan to correct for model errors

\- Computationally intensive

\+ Simple

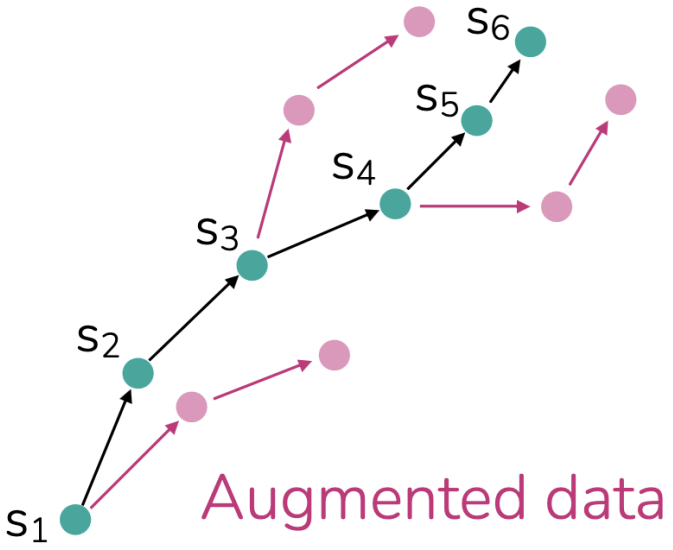


Figure 7.2: Augmenting existing data

+ Easy to plug in different reward functions

– Only practical for short-horizon problems or very shaped reward functions

Short horizon is because its both too computationally intensive and inaccurate to use the model for long horizons.

Question: Does MPC always give the most optimal solution

No. It could happen when the horizon is too short, or the optimiser is not powerful enough, or the model is inaccurate.

Can we train a policy using a learned model? We can distill planner's action into a policy

+ No longer computationally intensive at test time

– Still limited to short-horizon problems unless a terminal function is used.

To solve the short-horizon problem, we can also augment model-free RL methods using data from the model. With a given trajectory (s_i), we can augment data using the learned model p_ϕ by branching from s_1. To mitigate this issue, we can create augmented trajectories by starting from later states s_i, which improves coverage without having to do long rollouts.

Question: How do we know the reward for this?

We either know the functional form of the reward and evaluate it on augmented data, or we can have the model predict both the reward and the state $\mathsf{p}_\phi(r, s'|s, a)$.

Algorithm: Model-Based Policy Optimisation

1. Rollout policy π_θ and add data (s, a, s') to dataset $\mathcal{D}$
2. Update dynamics model $\mathsf{p}_\phi(s'|s, a)$ (or $\mathsf{p}_\phi(r, s'|s, a)$ if the reward function is unknown) from $\mathcal{D}$
3. Collect synthetic rollouts using π_θ in model p_ϕ from states in $\mathcal{D}$ and add to $\mathcal{D}_{\mathsf{model}}$
4. Update policy π_θ and critic Q using $\mathcal{D}_{\mathsf{model}} \cup \mathcal{D}$

Usually in implementations, $\mathcal{D}_{\mathsf{model}}$ and $\mathcal{D}$ have fixed sizes and are first-in-first-out. Sometimes older data are kept to prevent regression. In practice the policy would be stochastic to encourage exploration

+ Models are immensely useful if easy to learn
+ Models can be trained without reward labels (self-supervised)
+ Models are somewhat task-agnostic (can sometimes be transferred across rewards)
- Models don't optimise for task performance
- Sometimes harder to learn than a policy

8 Reward Learning

- *Variational Inverse Control with Events: A General Framework for Data-Driven Reward Definition*. Fu et al. (2018)
- *Deep reinforcement learning from human preferences*. Christiano et al. (2017)

Announcements

- 04-26: Project Proposal due
- 05-03: Homework 2 due

Rewards in RL sometimes come from straightforward sources such as scores in computer games. In real world scenarios it could be unclear what the reward function may be. Often we use a proxy. In **Direct Imitation Learning**, the policy directly mimics the actions of an expert.

- There is no reasoning about outcomes or dynamics in this method
- The expert might have different degrees of freedom
- Might not be possible to provide demonstrations

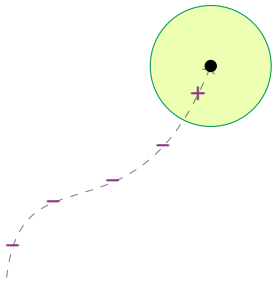


Figure 8.1: The goal classifier assigns a score of 1 to states close to the item and a reward of 0 for states farther away from the item

8.1 Rewards from Behaviour

A **Goal Classifier** is a model r that discerns goal states from other states. e.g. If the agent is trying to find an item, it could be rewarded when it gets to a position (state) that is close to the item. The goal classifier could be trained to classify successful and unsuccessful states and use this goal classifier as the reward function.

- The agent may learn artefacts related to the goal classifier and not focus on the task itself.

Can we prevent the RL algorithm from exploiting the classifier's weaknesses? During the RL training process, policy roll-outs generate fresh new samples. We can add these sample states to the datasets $\mathcal{D}_{\pm}$ used to train the goal classifier.

Algorithm: Goal Classifier

1. Collect initial successful states $\mathcal{D}_+$ and unsuccessful states $\mathcal{D}_-$
2. Update classifier r using $\mathcal{D}_+, \mathcal{D}_-$
3. Roll-out policy π and obtain trajectories $s_t, a_t, \ldots$
4. Update policy π using r based reward.
5. Add visited states s_t to $\mathcal{D}_-$
6. Repeat from (2)

where to ensure the classifier receives balanced samples,

- Keep $\mathcal{D}_+$ and $\mathcal{D}_-$ the same size using first-in-first-out on $\mathcal{D}_-$
- Or, when sampling batches from $\mathcal{D}_{\pm}$ we sample half from $\mathcal{D}_+$ and half from $\mathcal{D}_-$ (oversampling $\mathcal{D}_+$ undersampling $\mathcal{D}_-$)
- Large enough $\mathcal{D}_+$ is needed to not overfit into the small $\mathcal{D}_+$ set
- Or, we could regularise the classifier

Excursion: Generative Adversarial Networks

Generative Adversarial Networks are a class of models where two networks, a *generator* and a *discriminator*, play a two-player zero-sum game. The generator attempts to generate new samples which mimics a data distribution $p(x)$ (from which we have samples from) and the discriminator attempts to tell apart the synthesised sample from real samples.
The discriminator may pickup subtle details from the generator artefacts. Some techniques to regularise the discriminator in GAN: Gradient clipping, Spectral normalisation, instance noise

Alternatively, in the case of not trying to reach a goal, we could use **Generative Adversarial Imitation Learning**:

Algorithm: Generative Adversarial Imitation Learning

1. Collect expert demonstration trajectories $s_t, a_t, \dots$ from expert policy π_{expert} and add (s, a) pairs to $\mathcal{D}_+$.
2. Update classifier r using $\mathcal{D}_+, \mathcal{D}_-$
3. Roll-out policy π and obtain trajectories $s_t, a_t, \dots$
4. Update policy π using r based reward.
5. Add visited states (s, a) to $\mathcal{D}_-$ and return to (2)

\+ Practical framework for task specification

~ Adversarial training can be unstable

\- Requires examples of desired behavior or outcomes

8.2 Rewards from Human Preferences

Suppose we have trajectories (full or partial roll-outs) τ_+, τ_-, where $\tau_+ \succ \tau_-$, i.e. humans feedback indicates τ_+ is better than τ_-. We want a reward that gives $r(\tau_+) > r(\tau_-)$. We can define the estimated probability of $\tau_+ \succ \tau_-$ to be $\sigma(\mathsf{r}_\theta(\tau_+) - \mathsf{r}_\theta(\tau_-))$. Then we can maximise the log probability

$$\max_\theta \mathbb{E}_{\tau_+ \succ \tau_-} \left[\log \sigma(\mathsf{r}_\theta(\tau_+) - \mathsf{r}_\theta(\tau_-))\right]$$

The states that are compared must have the same initial state. In LLM, the states are instead constrained to have the same prompt. This is to ensure comparisons are across the same category.

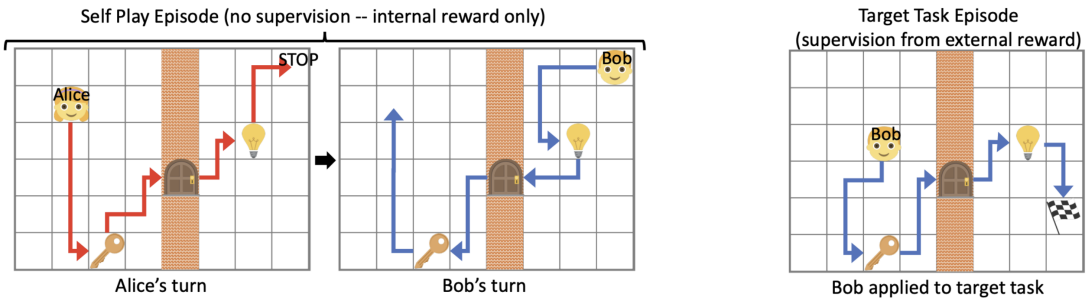


Figure 8.2: Sukhbaatar et al. *Intrinsic Motivation and Automatic Curricula via Asymmetric Self-Play.* ICLR 2018

Algorithm: Learn a Reward Function from Human Preferences

1. Given dataset, sample batches of trajectories $\{\tau_i\}$ and ask humans to rank.
2. Compute their rewards $\mathsf{r}_\theta(\tau_i)$ under the current reward model r_θ
3. For all $\binom{k}{2}$ pairs per batch, compute

$$\nabla_\theta \, \mathbb{E}_{\tau_+ \succ \tau_-} \left[\log \sigma(\mathsf{r}_\theta(\tau_+) - \mathsf{r}_\theta(\tau_-)) \right]$$

4. Update θ using the gradient from (3) and return to (2)

This can be done in the loop of online RL. This is done to train ChatGPT:

1. Train f to do next-token prediction on massive dataset
2. Supervised fine-tuning to follow instructions
3. Train reward model r_θ
4. Use PPO to maximise $\sum \mathsf{r}_\theta - \mathrm{D}_{\mathsf{KL}}(\hat{f} \| f)$ where $\hat{f}$ is a model trained on natural language to encourage f to generate natural outputs.

The feedback can also come from another AI model since critique is easier than generation. Learning rewards from human preferences:

+ Pairwise preferences are easy to provide
+ Has been deployed at scale
- May require supervision in the loop of RL

Can RL agents provide their own goals using "unsupervised" RL? One example is that we can formulate goals using a two-player game with a goal setter and goal reacher.

9 Offline RL

- *Conservative Q-Learning for Offline Reinforcement Learning*. Kumar et al. (2020)
- *MT-Opt: Continuous Multi-Task Robotic Reinforcement Learning at Scale*. Yu et al. (2021)

Announcements

- 05-03: Homework 2 due
- 05-03: Homework 3 out

Why would we do offline RL? **Offline RL** refers to RL on a given static dataset. Offline RL is useful when collection of new data or rolling out of policies is expensive. It can also utilise existing datasets.
In offline RL, we assume we have a dataset $\{(s, a, s', r)\}$ sampled from some unknown **behaviour policy** π_β, which can be a mixture of policies. In this policy,

$$s \dim d^{\pi_\beta}(\cdot), a \sim \pi_\beta(\cdot|s), s' \sim p(\cdot|s, a), r = r(s, a)$$

The objective is to learn

$$\max_\theta \sum_t \mathbb{E}_{s_t \sim d^{\pi_\theta}(\cdot), a_t \sim \pi_\theta(\cdot|s_t)}[r(s_t, a_t)]$$

This objective is online and cannot be evaluated purely from an offline dataset.
Can we just use off-policy algorithms? e.g. the standard Q-learning algorithm

$$\min_Q \min \sum_{(s,a,s') \in \mathcal{D}} Q(s, a) - (r(s, a) + \gamma \max_{a'} Q(s', a'))^2$$

In most cases this does not work. The reason why this doesn't work is that when Q is evaluated on an action a' not in the dataset:

- Q-function will be unreliable on OOD actions
- max will seek out actions where Q-function is over-optimistic
- After values propagate, Q-values will become substantially overestimated.

Question: Could you do dataset attribution in offline RL to determine which part of the dataset lead to an action?

Haven't seen a previous work like this but it would be interesting to look at.

Instead of using a maximum, we could replace $\max_{a'}$ with

$$\mathbb{E}_{a \sim \pi_{\text{new}}(\cdot|s)}[Q(s, a)], \qquad \pi_{\text{new}} := \arg\max_{\pi_{\text{new}}} \mathbb{E}_{a \sim \pi_{\text{new}}}[Q(s, a)]$$

and π_{new} should be close to π_β. There are two ways to encourage closedness:

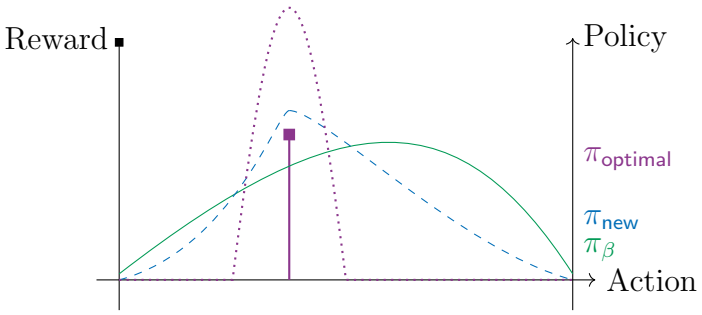


Figure 9.1: KL Divergence policy may be closer to the behaviour policy than the optimal policy.

9.1 Data Constraint Methods

- Absolute continuity: Allow $\pi_{\text{new}}(a|s) > 0$ only if $\pi_\beta(a|s) \geq \epsilon$
  - \+ Close to what we want
  - – Challenging to implement in practice
- Divergence: minimise $\mathrm{D}_{\mathsf{KL}}(\pi_{\text{new}} \| \pi_\beta)$
  - \+ May not be what we want
  - – Easy to implement in practice.

Both of these cannot be evaluated exactly and models use some imitation learning to fit $\hat{\pi}_\beta$ to π_β.
This leads to the data constraint methods:

Algorithm: Data Constraint Methods

Different ways of implementing data constraint methods:

1. Change actor update via Lagrange Multiplier λ

$$\begin{aligned}\theta &\leftarrow \arg\max_\theta \mathbb{E}_{s \dim \mathcal{D}, a \sim \pi_\theta(\cdot|s)}[Q(s,a)] - \lambda \, \mathrm{D}_{\mathsf{KL}}(\pi_\theta \| \pi_\beta) \\ &= \arg\max_\theta \mathbb{E}_{s \dim \mathcal{D}, a \sim \pi_\theta(\cdot|s)}[Q(s,a)] + \lambda \log \pi_\beta(a|s) + \lambda \, \mathrm{H}(\pi_\theta(\cdot|s))\end{aligned}$$

2. Modify reward function

$$\bar{r}(s,a) := r(s,a) - \mathrm{D}_{\mathsf{KL}}(\pi_\theta \| \pi_\beta)$$

Constraints by modeling π_β:

\+ Intuitive

- Often this is too conservative in practice. Relaxing λ can mitigate this somewhat but risks running off the distribution support.

9.2 Conservative Methods

We could also overestimation in offline RL. Without explicitly modeling behaviour policy, we can push down on large Q-values.

Standard Critic Update (Bellman)

$$\hat{Q} \leftarrow \arg\min_{Q} \max_{\mu'} \mathbb{E}_{(s,a,s')\sim\mathcal{D}} \left[(Q(s,a) - (r(s,a) + \gamma\, \mathbb{E}_{\pi}[Q(s',a')])) \right]$$

$$+ \alpha\, \mathbb{E}_{s\sim\mathcal{D}, a\sim\mu(\cdot|s)}[Q(s,a)] - \alpha\, \mathbb{E}_{(s,a)\sim\mathcal{D}}[Q(s,a)]$$

Push down on large Q values

Push up on data Q values

Without the push down term, we have $\hat{Q}^{\pi} \leq Q^{\pi}$ for all (s,a). With the push up term, this we no longer guarantee $\hat{Q}^{\pi} \leq Q^{\pi}$ for all (s,a) but we still have $\mathbb{E}_{\pi(a|s)}[\hat{Q}^{\pi}(s,a)] \leq \mathbb{E}_{\pi(a|s)}[Q^{\pi}(s,a)]$. This is known as **Conservative Q-Learning** (**CQL**). Often we want to add a regulariser term that maximises the entropy $\mathrm{H}(\mu)$ e.g. $R(\mu) := \mathbb{E}_{s\sim\mathcal{D}}[\mathrm{H}(\mu(\cdot|s))]$. The optimal $\mu(a|s) \propto \exp(Q(s,a))$. Thus

$$\mathbb{E}_{s\sim\mathcal{D}, a\sim\mu(\cdot|s)}[Q(s,a)] = \log \sum_{a} \exp Q(s,a)$$

Algorithm: Conservative Q-Learning

1. Update $\hat{Q}$ using L_{CQL} using dataset $\mathcal{D}$
2. Update policy π:
   - If actions are discrete, $\pi(a|s) = \mathbb{1}\{a = \arg\max_a \hat{Q}(s,a)\}$
   - If actions are continuous, $\theta \leftarrow \theta + \eta \nabla_{\theta}\, \mathbb{E}_{s\sim\mathcal{D}, a\sim\pi_{\theta}(\cdot|s)}[\hat{Q}^{\pi}(s,a)]$

In **Model-Based Offline RL**, we use the following *model-generated* state-action pairs instead of μ:

$$\mathbb{E}_{(s,a)\sim\rho}[Q(s,a)]$$

Intuition: If model produces data that look clearly different from the real data, it's easy for Q to make it look bad. Here a policy is needed to learn ρ. The difference between this and μ is that it also attempts to push down on model generated states, whereas μ only pushes on the actions.
Implicitly constrain data by penalising Q:

+ Simple
+ Works fairly well in practice
- Need to tune α

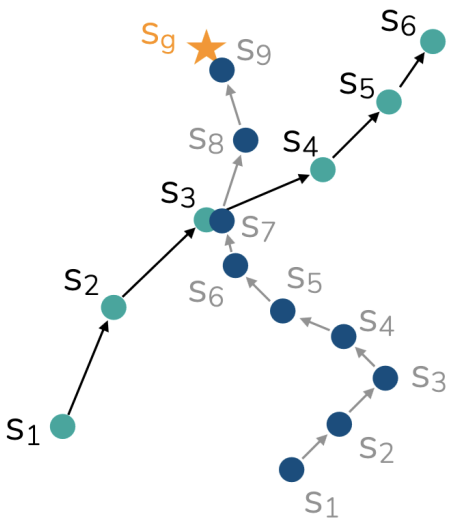


Figure 9.2: Offline RL can stitch together trajectories $s_1 \to s_3, s_7 \to s_9$ and learn a policy that goes from s_1 to s_9.

9.3 Data Stitching

Why Offline RL instead of imitation learning? Offline learning can stitch together different parts of the data with optimal behaviour by leveraging reward even when the data is sub-optimal. *Imitation learning cannot outperform the expert.*

10 Offline RL cont.

- *Offline Reinforcement Learning with Implicit Q-Learning*. Kostrikov et al. (2021)

Announcements

- 05-03: Homework 2 due
- 05-03: Homework 3 out (covers weighted and conditional imitation learning)
- Two office hours moved from in-person to hybrid

The key challenge in offline RL is the over-estimation of Q-values because of the shift between π_β and π_θ. We can

- Explicitly constrain to the data by modeling π_β
- Implicitly constrain to the data by under-estimating Q

There are other ways to leverage reward in imitation learning. Pure imitation learning does not use reward information. If we have reward labels, can we imitate only good trajectories? One way is to filter the data:

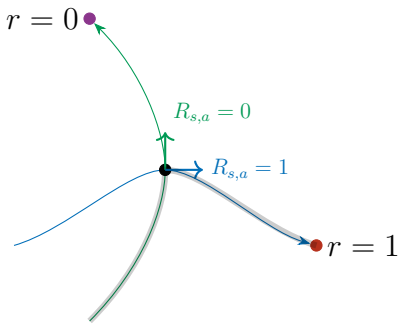


Figure 10.1: The advantage at the intersection prefers the action towards the direction of higher reward. This allows the two trajectories to be stitched together.

Algorithm: Filtered Behaviour Cloning (%BC)

1. Collect data $\mathcal{D} := \{(s, a, s', r)\}$:
2. Filter data $\tilde{\mathcal{D}} := \{\tau : r(\tau) > \eta\}$ for some η such that $\left|\tilde{\mathcal{D}}\right| = k\% \, |\mathcal{D}|$
3. Imitation learning $\max_\theta \mathbb{E}_{s,a\sim\tilde{\mathcal{D}}} \log \pi(a|s)$

A policy trained like this is a very good baseline to test against.

10.1 Weighted Imitation Learning

Could we weight each transition depending on how good the action is? We can measure the goodness of an action using the *advantage function* $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$ to measure the advantage of taking action a relative to the average action on π.
We can perform **weighted imitation learning** using A as the weight

$$\max_\theta \mathbb{E}_{(s,a)\sim\mathcal{D}}[\log \pi(a|s) \ \exp(A(s,a))] \tag{10.1}$$

Standard Imitation Learning ↑ Advantage weights ↑

Theorem 10.1. *The advantage-weighted objective (Equation 10.1) approximates KL-constrained objective*

$$\pi_{\text{new}} := \operatorname*{arg\,max}_{\pi:\mathrm{D}_{\mathsf{KL}}(\pi\|\pi_\beta)\le\epsilon} \mathbb{E}_{a\sim\pi(\cdot|s)} Q(s, a)$$

Proof. See *Relative Entropy Policy Search*. Peters et al. and *On Stochastic Optimal Control and Reinforcement Learning by Approximate Inference*. Rawlik et al. ("Ψ-Learning"). □

For now we will use the advantage of the behaviour policy A^{π_β}. How can we estimate the advantage function? One way is to estimate the value function V^{π_β} with Monte Carlo methods

$$\hat{V}^{\pi_\beta} := \min_V \mathbb{E}_{(s,a)\sim\mathcal{D}}[(V(s) - R_{s,a})^2]$$

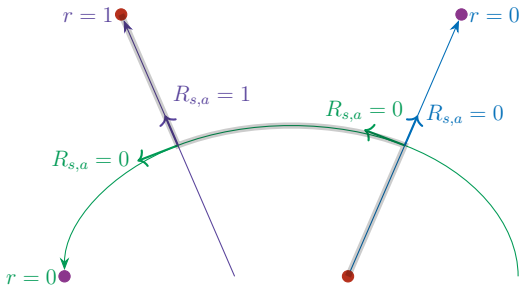


Figure 10.2: The advantage function would not be able to distinguish the two actions at s_1 when trained with Monte Carlo. Therefore it would not be able to stitch together the trajectory between the two red dots.

where $R_{s,a}$ is the **empirical reward** or reward to-go in the trajectory $R_{s_t,a} = \sum_{t'=t}^{T} r(s_{t'})$. then we can approximate $\hat{A}^{\pi_\beta}(s,a) = R(s,a) - \hat{V}^{\pi_\beta}(s)$. This gives the **Advantage-Weighted Regression** algorithm.

Algorithm: Advantage-Weighted Regression (AWR) with Monte Carlo Updates

1. Fit value function

$$\hat{V}^{\pi_\beta} \leftarrow \arg\min_V \mathbb{E}_{(s,a)\sim\mathcal{D}}[(V(s) - R_{s,a})^2]$$

2. Train policy

$$\hat{\pi} \leftarrow \arg\max_\pi \mathbb{E}_{(s,a)\sim\mathcal{D}}\left[\log \pi(a|s) \exp\left(\frac{1}{\alpha}(R_{s,a} - \hat{V}^{\pi_\beta}(s)\right)\right]$$

+ Simple
+ Avoid querying or training on any OOD actions
- Monte Carlo estimation is noisy
- $\hat{A}^{\pi_\beta}$ assumes *weaker* policy than $\hat{A}^{\pi_\theta}$

AWR can fail when too many trajectories are needed to be stitched together, as shown in example Figure 10.2. In order to mitigate this issue we can train $\hat{Q}$ using an iterative process:

$$\hat{Q}^\pi \leftarrow \min_Q \mathbb{E}_{(s,a)\sim\mathcal{D}}\left[(Q(s,a) - (r + \gamma\, \mathbb{E}_{a'\sim\pi_\theta}[Q(s',a')]))^2\right]$$

then we would have

$$\hat{A}^\pi(s,a) = \hat{Q}^\pi(s,a) - \mathbb{E}_{a'\sim\pi_\theta(\cdot|s)}[\hat{Q}^\pi(s,a')]$$

Algorithm: Advantage-Weighted Regression with Temporal Difference (TD) Updates

1. Fit Q-function

$$\hat{Q}^\pi \leftarrow \min_Q \mathbb{E}_{(s,a)\sim\mathcal{D}}\left[\left(Q(s,a) - (r + \gamma\, \mathbb{E}_{a'\sim\pi_\theta(\cdot|s)}[Q(s',a')])\right)^2\right]$$

2. Estimate advantage

$$\hat{A}^\pi(s,a) = \hat{Q}^\pi(s,a) - \mathbb{E}_{a'\sim\pi_\theta(\cdot|s)}[\hat{Q}^\pi(s,a')]$$

3. Train policy

$$\hat{\pi} \leftarrow \arg\max_\pi \mathbb{E}_{(s,a)\sim\mathcal{D}}\left[\log\pi(a|s)\,\mathbb{E}\left(\frac{1}{\alpha}\hat{A}^\pi(s,a)\right)\right]$$

+ Policy still only trained on action data
+ Temporal difference updates instead of Monte Carlo
– The evaluation of $a \sim \pi(\cdot|s)$ is particularly problematic because it possibly queries OOD actions!

In *AWAC: Accelerating Online Reinforcement Learning with Offline Datasets*. Nair et al. (2020), the evaluation of $a \sim \pi(\cdot|s)$ is replaced by $a' \sim \mathcal{D}$ to avoid sampling out of distribution. This leads to the **SARSA** (State-Action-Reward-State-Action) algorithm:

Algorithm: SARSA

Update the $\hat{Q}$ function with

$$\hat{Q}^\pi \leftarrow \min_Q \mathbb{E}_{(s,a,s',a')\sim\mathcal{D}}\left[\left(Q(s,a) - (r(s,a) + \gamma\, \underbrace{Q(s',a')}_{\text{A sample of } V^{\pi_\beta}(s')}\right)^2\right]$$

where the tuple (s,a,s',a') represents *two consecutive states* in the trajectory.

This ties Q with π_β. We can train a policy better than π_β using an *asymmetric loss function*. The L^2 loss function targets the mean of a random variable. We can target a higher or lower percentile using **expectile regression**, with a loss function

$$\ell_2^\tau(x) := \begin{cases} (1-\tau)x^2 & x > 0 \\ \tau x & x \le 0 \end{cases}$$

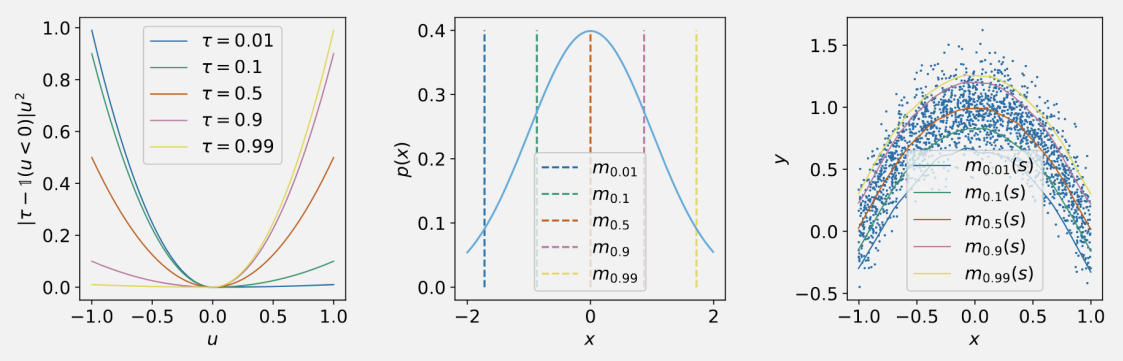


Figure 10.3: The expectile loss function ℓ_2^τ and its use cases in regression from Kostrikov et al. (2021)

Algorithm: Implicit Q-learning (IQL)

1. Fit V-function with expectile-loss (with large $\tau > 0.5$):

$$\hat{V}(s) \leftarrow \arg\min_V \mathbb{E}_{(s,a)\sim\mathcal{D}}\left[\ell_2^\tau(V(s) - \hat{Q}(s,a))\right]$$

2. Fit Q-function with MSE loss:

$$\hat{Q}^\pi \leftarrow \min_Q \mathbb{E}_{(s,a,s')\sim\mathcal{D}}\left[\left(Q(s,a) - (r + \gamma\hat{V}(s'))\right)^2\right]$$

At the end, extract policy:

$$\hat{\pi} \leftarrow \arg\max_\pi \mathbb{E}_{(s,a)\sim\mathcal{D}}\left[\log\pi(a|s)\exp\left(\frac{1}{\alpha}(\hat{Q}(s,a) - \hat{V}(s))\right)\right]$$

+ Never need to query OOD actions
+ Policy is only trained on actions in data
+ Decoupling actor and critic training, so computationally fast.

Question: Do we still have to do it if we have access of the underlying policy π_β?

Yes.

- It could be used to augment the data in case a state is never associated with some action in $\mathcal{D}$.
- In Temporal Difference updates, the Q-function's estimation of the remainder can be directly taken on the distribution $a' \sim \pi_\beta(\cdot|s)$ if π_β is available.

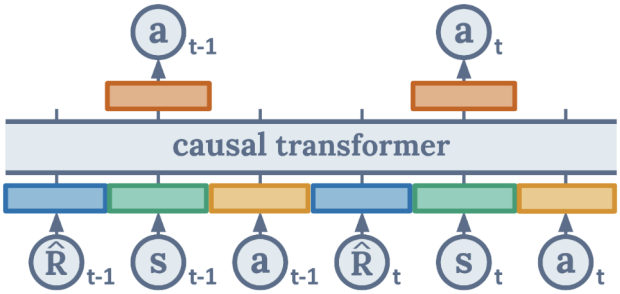


Figure 10.4: Causal transformer used for conditional imitation learning

10.2 Conditional Imitation Learning

For some datasets, filtered BC can work really well. The drawback with filtered BC is it discards data. Instead of discarding such data, we can condition the policy on the return achieved on trajectory τ on the state s (return to go). i.e.

$$\hat{\theta} \leftarrow \arg\max_{\theta} \mathbb{E}_{(a,s)\sim\mathcal{D}}[\log \pi(a|s, R_{s,a})]$$

A policy trained in this manner can mimic both good and bad behaviours and the policy can be tuned to mimic both good and bad data. This is referred to as **Upside-down RL**, **Reward-conditioned policies**, or **Decision transformers**. The model can be sequential:

- A policy trained like this cannot be expected to stitch trajectories.

10.3 Offline Evaluation and Hyper-parameter Tuning

Can we evaluate a policy or compare the performance of two trained policies $\pi_{\theta_1}, \pi_{\theta_2}$ without rolling out the policy? There is no reliable way to evaluate off-line.

- Roll-out policy and evaluate:
  - \+ Accurate
  - − Can be risky or expensive
  - ~ Can be risky or expensive

  This makes the algorithm not fully off-line, at which point it may also be possible to train the policy on-line.
- Evaluate in high-fidelity simulator or model
  - \+ May be good enough in some situations
  - − Developing simulator is hard
- Some algorithms have heuristics

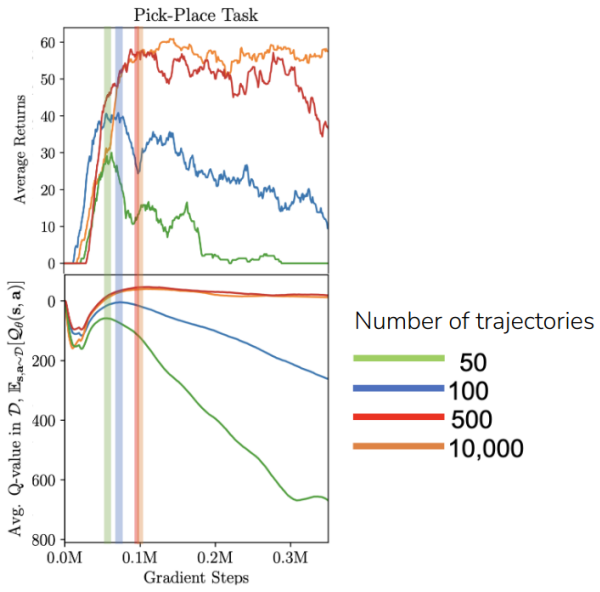


Figure 10.5: Early stopping with CQL

In CQL (Conservative Q-Learning), the peak and average Q-values during training can be used to evaluate the policy. By looking at the history of training, this heuristic can be used to early stop CQL before it overfits. See *A Workflow for Offline Model-Free Robotic Reinforcement Learning*. CoRL '21

11 Multi-Tasked RL and Goal-Conditioned RL

Announcements

- 05-17: Homework 3 due

The biggest challenge of the reinforcement learning methods so far is *sample complexity*. The algorithms take a long time and a large amount of data to train.

11.1 Multi-Task Imitation and Policy Gradients

The classic supervised learning loss for a single task imitation learning is

$$\min_\theta L(\theta, \mathcal{D}) = \min_\theta -\mathbb{E}_{(s,a)\sim\mathcal{D}}[\log \pi_\theta(a|s)]$$

In a multi-task imitation learning, the loss can be aggregated over several task datasets $\min_\theta \sum_i L(\theta, \mathcal{D})$.
A reinforcement learning task is given by

State space — Dynamics — Reward

$$\mathcal{T} := \{\mathcal{S}, \mathcal{A}, p(s_1), p(s'|s,a), r(s,a)\}$$

Action space — Initial state distribution

Question: Does the dynamics model have to be stochastic

Not necessarily.

A set of tasks $\{\mathcal{T}_i\}$ can be cast as a standard Markov decision process, with states being augmented with a task identifier z_i: $s := (\bar{s}, z_i)$. The pros and cons of multi-task learning are

+ Cross-task Generalisation
+ Easier exploration
+ Sequencing for long horizon tasks
+ Reset-free learning: If two tasks are complementary, they can reset each other's terminal state and be trained in conjunction.
+ Per-task sample-efficiency gains
- Poor generalisation cf. individual RL models

  This is exhibited in Metaworld (CoRL 2019), a benchmark about multi-task reinforcement learning algorithms. Each task is solvable by a single-task RL model.

Excursion: Gradient projection

When training a multi-task model for task $1, 2$, it may happen that the gradient $\boldsymbol{g}_1 := \nabla_\theta L_1$ and $\boldsymbol{g}_2 := \nabla_\theta L_2$ point in different directions. If the gradients $\boldsymbol{g}_1 \cdot \boldsymbol{g}_2 < 0$, it may be difficult to train the model. A solution is to project conflicting gradients onto the normal planes of each other

$$\hat{\boldsymbol{g}}_2 := \begin{cases} \boldsymbol{g}_2 - \text{proj}_{\boldsymbol{g}_1} \boldsymbol{g}_2 & \boldsymbol{g}_1 \cdot \boldsymbol{g}_2 < 0 \\ \boldsymbol{g}_2 & \boldsymbol{g}_1 \cdot \boldsymbol{g}_2 \geq 0 \end{cases}$$

With the projection order randomised. This mitigates generalisation issues.

The difference between reinforced and supervised learning algorithms is that the agent also decides the data distribution, and so far we have not leveraged this.

11.2 Multi-Task Q-Learning

Consider an agent which is trained to play hockey. What if an agent performs a good pass while attempting to shoot a goal? The data does not have to go to waste and it can be labeled in hindsight as an example of passing. This is **Hindsight Experience Replay** (**HER**)

Algorithm: Multi-Task RL with Re-labeling

1. Collect data $\mathcal{D}_k := \{(s_{1:T}, a_{1:T}, z_i, r_{1:T})\}$
2. Store data in replay buffer $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_k$
3. Perform *hindsight relabeling*: Relabel experience in $\mathcal{D}_k$ for task $\mathcal{T}_i$: $\mathcal{D}'_k := \{(s_{1:T}, a_{1:T}, z_j, r'_{1:T})\}$ where $r'_t = r_j(s_t)$
4. Store relabeled data in replay buffer $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}'$

Methods of choosing $\mathcal{T}_i$ in step (3):

- Randomly
- Tasks in which the trajectory get high reward
- Other

The mechanism of re-labeling is highly non-trivial and there has been a number of works in this (Eysenbach et al. Rewriting History with Inverse RL, Li et al. Generalized Hindsight for RL, Kalashnikov et al. MT-Opt, Kalashnikov et al. MT-Opt). We can apply re-labeling when

- Reward function form is known, relatable
- Dynamics consistent across tasks/goals
- Using an off-policy algorithm

For example, suppose two tasks are closing and opening a drawer. Can we use episodes from drawer opening task for drawer closing task? The answer is yes since the drawer closing task can be used as negative examples in Q-Learning, but not in policy gradient.

11.3 Goal-Conditioned RL

Algorithm: Goal-Conditioned RL with Re-labeling

1. Collect data $\mathcal{D}_k := \{(s_{1:T}, a_{1:T}, z_i, r_{1:T})\}$
2. Store data in replay buffer $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_k$
3. Perform *hindsight relabeling*: Relabel experience in $\mathcal{D}_k$ for task $\mathcal{T}_i$: $\mathcal{D}'_k := \{(s_{1:T}, a_{1:T}, z_j, r'_{1:T})\}$ where $r'_t = -d(s_t, s_T)$
4. Store relabeled data in replay buffer $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}'$

Alternatively, any state in the trajectory can be used as relabeling.

Goal-Conditioned RL alleviates the exploration problem since any state can be used as a positive example. The reward is measured only by the distance to the target state $d(s_t, s_T)$.

Example

In Lynch, Khansari, Xiao, Kumar, Tompson, Levine, Sermanet. Learning Latent Plans from Play. '19, random human play data (with no task or goal) is fed into the policy as data.

12 Transfer Learning in RL

Guest Lecture: Jie Tan
Previous works using reinforcement learning on legged locomotion:

1. *Learning to Walk via Deep Reinforcement Learning* (Haarmoja et al. RSS 2019): The robot learns to walk on its own after 2 ours of training
2. *Sim-to-Real* (RSS 2018): Using domain randomisation to train a robot
3. *Learning to Walk in Minutes* (Rudin, 2021): Training thousands of robot clones in simulation at the same time reduces the overall training time.
4. *Rapid Motor Adaptation* (RSS 2021): Adapt to new domains in seconds.
5. *Learned Gait Transitions* (CoRL 2021): Using energy efficiency to automatically emerge natural gaits.
6. *Learning Agile*
7. *Learning to Navigate Sidewalks* (ICRA 2022): Combining perception and locomotion together to navigate sidewalks in urban area.
8. *Learning Robust Perceptive Locomotion* (Science Robotics 2022)

Model Predictive Control (MPC) has made a lot of progress in the past years, but DRL has caught up and produced state-of-the-art results.

12.1 Learning Locomotion

There are two approaches to learn locomotion: In the real world, or in the simulation.
The problem about real world learning

- data efficiency, 60 million steps are needed to learn human-like locomotion behaviour which will take years.
- Human supervision is required to reset the robot's state.
- Safety: Robotics can damage things in the real world or itself

The problem setup is *Learning to Walk in Real World with Minimum Human Efforts* (CoRL 2020):

- Observations: 8 motor angles, roll, pitch, previous action in the *last 6 time-steps*
- Actions: 8 target motor angles
- Reward function: A combination of velocity along a direction, turning rate, and a penalisation term for acceleration.

Once we have this formulation, we have the **Soft Actor-Critic** (**SAC**)

$$\max_{\pi} \mathbb{E}_{\tau\sim\rho_\pi}\left[\sum_{t=0}^{T} r(s_t,a_t)\right] \quad \text{s.t.} \begin{cases} \mathbb{E}_\pi[-\log(\pi_t(\cdot|s_t))] & \geq 0 \\ \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}\left[\sum f_s(s_t,a_t)\right] & \geq 0 \\ \mathbb{E}_\pi[-\log(\pi_t(\cdot|s_t))] & \geq \mathcal{H} \end{cases}$$

Policy rollout of π

where

$$f_s(s_t,a_t) := \min(\hat{\phi} - |\phi_t|, \hat{\rho} - |\rho_t|)$$

Max safety pitch $\hat{\phi}$; Max safety roll $\hat{\rho}$

Using Lagrangian multipliers, the goal is equivalent to

$$\max_{\pi}\min_{\lambda\geq 0} \mathbb{E}_{\tau\sim\rho_\pi}\left[\sum_{t=0}^{T} r(s_t,a_t) + \lambda f_s(s_t,a_t)\right]$$

This zero-sum two-player game forces $f_s \geq 0$.

Algorithm: Dual Gradient Descent

1. Randomly initialise π, set $\lambda = 0$
2. Rollout policy π
3. Calculate policy gradient $\frac{\partial \mathcal{L}}{\partial \theta}$
4. $\pi \leftarrow \pi + \alpha \frac{\partial \mathcal{L}}{\partial \pi}$
5. Calculate gradient $\frac{\partial L}{\partial \lambda}$
6. $\lambda := \max(0, \lambda - \beta \frac{\partial \mathcal{L}}{\partial \lambda})$
7. Return to step (2)

The **Sim-to-Real Gap** is the difference between simulation and real world

- Dynamics: The embodiment gap makes the motor control different
- Perception: The visual sensors perceive differently in simulation and real world.

What are the causes of the sim-to-real gap?

- Unmodeled dynamics: Inaccurate contact models
- Wrong simulation parameters
- Latency
- Actuator dynamics
- Stochastic real environment
- Noise, Numerical accuracy

In the past few years, there has been an exponential growth on the papers for sim-to-real methods.

12.2 Sim-to-Real Gap

There are three methods of bridging the sim-to-real gap. **System Identification** refers to determining the precise physical parameters to plug in to the simulation. This involves measuring:

- Mass
- Centre of mass
- Motor resistance (viscous resistance)

Once a model is built, we can test the model by comparing simulations and real world. System Identification can also be done automatically.
Domain Randomisation is another method that improves the robustness of the policy.

- Trains efficiency for robustness
- Need careful tuning for the range of domain randomisation

Can we use a small amount of data to mitigate these limitations of DR?
Domain Adaptation: Create a set of physical parameters $\boldsymbol{\mu}$, which would be randomised in DR, and encode it in $\boldsymbol{z}$, a latent encoding of $\boldsymbol{\mu}$. Then condition the policy $\pi(a|s, \boldsymbol{z})$ on $\boldsymbol{z}$. When the policy is deployed to the real world, we freeze the policy weights π and search $\boldsymbol{z}$ to find the optimal policy. $\boldsymbol{z}$ is a low-dimensional vector so it can be optimised by simple optimisation algorithms e.g. Bayesian optimisation.
Limitations:

- The latent space may not contain the optimal version of the real world
- Policy is not updated; Performance does not necessarily improve
- Adaption is slow (requires a few episodes)

Rapid Motor Adaptation (**RMA**) uses an adaptation network which is called 10 times per second to assess the environment. This allows the robot to adapt to a variety of environments previously unseen in simulation.
Does sim-to-real solve everything?

- Some physical phenomena are difficult to model
- Impossible to capture the diversity of real-world scenarios

The complete picture of sim-to-real is that simulation and real-world are complementary methods.

13 Meta RL

Announcements

- 05-17: HW3 due, HW4 out
- 05-24: Project Milestone due

Multi-Task Learning is to solve multiple tasks and minimise a loss function of the form

$$\min_{\theta} \sum_{i=1}^{T} L_i(\theta, \mathcal{D}_i)$$

Transfer Learning refers to solving target task $\mathcal{T}_2$ after solving source task $\mathcal{T}_1$ by transferring knowledge from $\mathcal{T}_1$.
Meta-Learning refers to based on data from $\mathcal{T}_1, \ldots, \mathcal{T}_n$ to solve a new task $\mathcal{T}_{\text{new}}$. In all settings, the tasks must share the same structure. Examples of meta-learning tasks:

- A manipulator which can move objects according to only a few examples shown
- A chat bot which adapts to user preferences

In **Meta supervised learning**, the model is fed in training data $(\mathcal{D}_i, \boldsymbol{x}_i)$ and tested with a new set of training data $(\mathcal{D}, \boldsymbol{x})$. In meta reinforcement learning, the model is fed $(\mathcal{D}, s)$ pairs and outputs action a.
The caveat of meta reinforcement learning is that an additional component of data collection is involved. For example, a maze navigation model could be trained to quickly adapt to a new maze.
There are two variants of Meta RL:

- Episodic Variant: The data $\mathcal{D}$ are complete trajectories
- Online Variant: The data $(\mathcal{D}, s, a)$ represent the next step action the model needs to take.

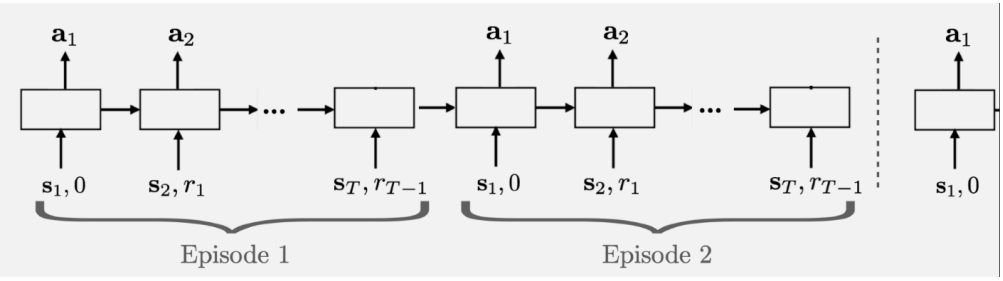


Figure 13.1: Black-Box Meta-RL Model

13.1 Black-Box Meta-RL Methods

In Black-Box methods, the model $f_\theta(\mathcal{D}, s)$ is a LSTM, NTM, or CNN. The training/support set is fed in as the beginning of the sequence and s, the query, is attached at the end.
The actions a_{t-1} does not need to be passed in since it is encoded in the state.
This method is different from RL with a recurrent policy, since

1. Reward is passed as input
2. Hidden state maintained

Algorithm: Training a Black-Box Meta-RL Model

Training

1. Sample task $\mathcal{T}_i$
2. Roll-out $\pi_\theta(a|s, \mathcal{D}_i^{\text{tr}})$ for N episodes, under dynamics $p_i(s'|s, a)$ and reward $r_i(s, a)$.
3. Add to training buffer $\mathcal{D}$
4. Optimise

$$\max_\theta \mathbb{E}_{\mathcal{T}_i} \sum_t \mathbb{E}_{(s,a)\sim\mathcal{T}_i,\pi_\theta}[r(s_t, a_t)]$$

Test

1. Sample new task $\mathcal{T}$
2. Roll-out policy $\pi(a|s, \mathcal{D}_j^{\text{tr}})$ for at most N episodes

\+ General and Expressive

\+ A variety of design choices in architecture

\- Hard to optimise

~ Inherits sample efficiency from outer RL optimiser

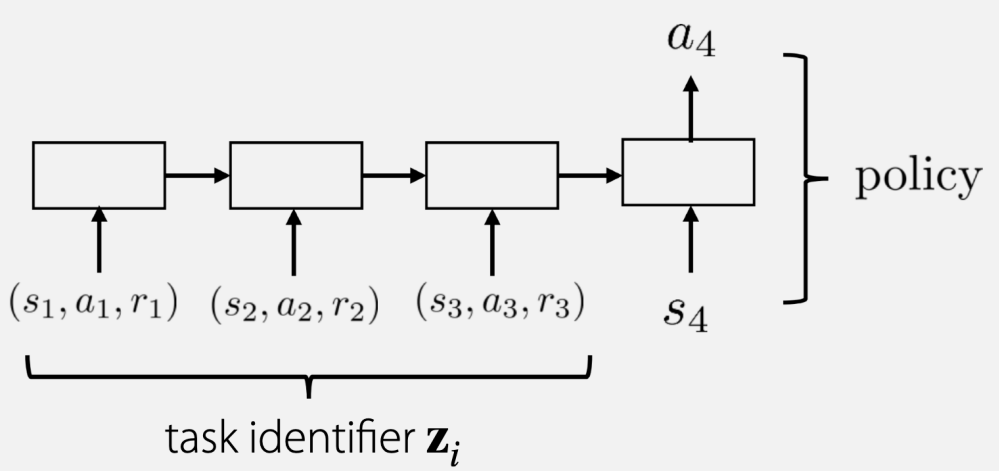


Figure 13.2: Multi-task Policy with experience as task classifier

13.2 Optimisation-Based Meta-RL Methods

Fine-Tuning refers to learning from pre-trained parameters by repeating

$$\phi \leftarrow \underbrace{\theta}_{\text{Pre-trained parameters}} - \alpha \nabla_\theta L(\theta, \mathcal{D}^{\text{tr}})$$

In meta-training, optimisations can be nested in each other to adapt to different tasks

$$\min_\theta \sum_t L(\theta - \alpha \nabla_\theta L(\theta, \mathcal{D}_t^{\text{tr}}), \mathcal{D}_i^{\text{tr}})$$

There are two disadvantages to this method. One is that the gradient descent process is costly, so the inner gradient descent may be executed with only a single step. The other is that this requires evaluation of second order derivatives. This method is **Model-Agnostic Meta-Learning** (**MAML**).
In optimisation-based meta RL, there are two loops of optimisation which MAML can go in.

Algorithm: MAML

Training

1. Sample task $\mathcal{T}_i$
2. Collect $\mathcal{D}_i^{\text{tr}}$ by rolling out π_θ
3. Inner loop adaptation: $\phi_i \leftarrow \theta + \alpha \nabla_\theta J_i(\theta)$
4. Collect $\mathcal{D}_i^{\text{ts}}$ by rolling out π_{ϕ_i}
5. Outer loop update: $\theta \leftarrow \theta + \beta \sum_{\text{task } i} \nabla_\theta J_i(\theta)$

Testing

1. Sample task $\mathcal{T}$
2. Collect $\mathcal{D}^{\text{tr}}$ by rolling out π_θ
3. Inner loop adaptation: $\phi_i \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

14 Meta RL: Learning to Explore

Announcements

- 05-17: HW3 due, HW4 out
- 05-24: Project Milestone due

14.1 Learning to Explore

Suppose we wish to train an agent which learns to explore. We can directly optimise for end-to-end exploration and exploitation w.r.t. reward:

\+ Simple

\+ Leads to optimal strategy in principle

\- Challenging optimisation when exploration is hard.

Example episodes during meta-training:

- The agent could go to the goal by pure random chance. Training on this episode would reinforce this opportunistic behaviour and discourage generalisation.
  - Gets positive reward for current task but $\mathcal{D}_i^{\text{train}}$ would not be different than for any other task
- The agent goes to the wrong path and then the goal.
  - ~ Provides signal on suboptimal exploration/exploitation strategy
- The agent looks at the instruction and then follows the instruction
  - \+ Optimal behaviour
  - \- Agent will not get extra rewards from this behaviour

End-to-end approach for training is hard due to a chicken and egg problem: Execution (learning to cook) and Exploration (learning to find ingredients) must be simultaneously learned and learning either of them have low reward on their own.
To mitigate these issues, we can use alternative exploration strategies

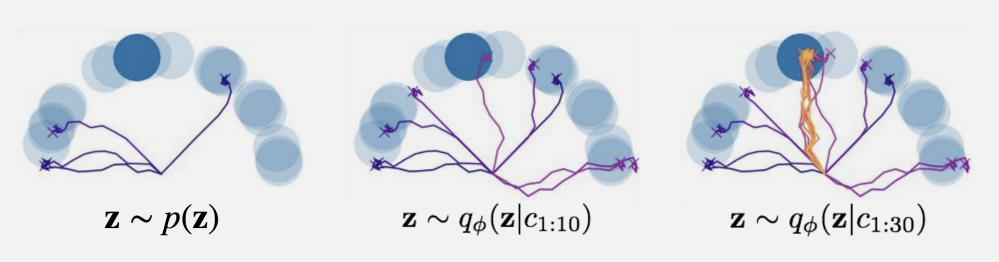


Figure 14.1: Multi-task Policy with experience as task classifier

- Posterior sampling/Thompson sampling (PEARL. Rakelly, Zhou, Quillen, Finn, Levine. ICML '19)

  1. Learn distribution over latent task variable $p(z), q(z|\mathcal{D}^{\text{train}})$ and corresponding task policies $\pi(a|s,z)$
  2. Sample z from current posterior and sample from policy $\pi(a|s,z)$

  - Can learn suboptimal exploration strategies
  - If each task takes a long time to complete the procedure will be extremely slow

- Use intrinsic rewards (MAME. Gurumurthy, Kumar, Sycara. CoRL '19)

- Task dynamics and reward prediction (MetaCURE. Zhang, Wang, Hu, Chen, Fan, Zhang. '20)

  Explicitly seek out states which *inform* about the dynamics.

  1. Train dynamics-reward model $f(s,r|s,a,\mathcal{D}^{\text{train}})$
  2. Collect $\mathcal{D}^{\text{train}}$ so model is accurate.

  - Lots of distractors of complex high-dimensional state dynamics

Overall, alternative exploration strategies are

+ Easy to optimise

+ Many based on principle strategies

- Suboptimal by arbitrarily large amount in some environments

14.2 Decoupled Exploration and Exploitation

Another solution is to extend the MetaCURE method. We label each training task with a unique id μ and trains an exploration policy to identify this id.
Meta-train:

- Execution policy $\pi^{\mathsf{exec}}(a|s,\mu)$

- Exploration policy π^{exp} and **task-identification model** $q(\mu_i|\mathcal{D}^{\text{train}})$

  The exploration policy generates $\mathcal{D}^{\text{train}}$ such that q maps this trajectory to the task identifier μ.

Meta-test:

1. Rollout exploration policy π^{exp} $\mathcal{D}$
2. Predict task id $\mu \leftarrow q(\cdot|\mathcal{D})$
3. Rollout $\pi^{\mathsf{exec}}(a|s,\mu)$

Question: How is the exploration policy supervised?

We train it using the goodness of deriving μ from $\mathcal{D} \sim \pi^{\mathsf{exp}}$ as the reward for π^{exp}. The reward is a type of information gain. An informal way to state the reward is

$$r_t := \text{accuracy of } q(\mu|\tau_{1:t}) - \text{accuracy of } q(\mu|\tau_{1:t-1})$$

There is one more ingredient to the algorithm. We can add a **information bottleneck** which maps from μ (task id) to a latent embedding z. Then we learn to explore by recovering this information

Excursion: Information Bottleneck

One way to add a bottleneck is to add noise in the representation $\bar{z} := z+\epsilon, \epsilon \sim N(0,1)$. We can either add this at train time or test time:

+ Will discard information
- If done at test time, may discard useful information
- If done at train time, the model can scale up the magnitude of z to bypass the bottleneck.

We can:

1. Add gaussian noise during training
2. Prevent the model from increasing magnitude

In **Variational Information Bottleneck**, we add noise before passing representation to the next layer: $\bar{z} := z+\epsilon$. Then we add a loss term $L^{\text{train}} + \|z\|^2$. This is equivalent to $\mathrm{D}_{\mathsf{KL}}(F(z|\mu_i)\|N(0,1))$

We train the execution policy and encoder $f(z|\mu)$ using

$$\mathbb{E}_{\pi^{\mathsf{exec}}}[r] - \mathrm{D}_{\mathsf{KL}}(f(z|\mu)\|N(0,1))$$

Here the execution policy is modified as $\pi^{\mathsf{exec}}(a|s,z)$.

[Fish]

This is the **Decoupled Reward-free ExplorAtion and Execution in Meta-Reinforcement Learning** method (**DREAM**)
(Informal) Theoretical Analysis shows

1. DREAM objective is consistent with end-to-end optimisation (under mild assumptions), and hence it can theoretically achieve optimum
2. Consider a bandit like setting with $|\mathcal{A}|$ arms

   In Task-i, arm i yields reward. In all Tasks, arm 0 reveals the rewarding arm.

   - RL2 requires $O(|\mathcal{A}|^2 \log |\mathcal{A}|)$ samples for meta-optimisation
   - DREAM requires $O(|\mathcal{A}| \log |\mathcal{A}|)$ samples for meta-optimisation

Decoupled Exploration and Exploitation:

+ Leads to optimal strategy in principle
+ Easy to optimise in practice
- Requires task identifier

15 Reset-Free RL

Guest Lecture by Archit Sharma
Consider an RL problem with data $s_0, a_0, \ldots, s_H$. At some point the agent needs to start from the beginning s_0'.
In simulated environment such as `gym`, we can use the handy `reset` function to reset the environment to the initial state, whereas a robot trained in the real world will have to have its state reset by a human operator. **Autonomous RL** refers to agents operating or training with minimal human supervision.
We can evaluate autonomous RL using **Deployed Policy Evaluation** and **Continuous Policy Evaluation**.

15.1 Forward-Backward RL and MEDAL

What happens when the episode length is increased? The deployed policy evaluation degrades a lot.
In **episodic learning**, we always start from the initial state distribution and always reset to the initial state. In **non-episodic learning**, a few challenges arise

1. Exploration can cause the agent to drift far away.
2. The state distribution collapses since the agent learns to exploit one particular trajectory.

We can learn a policy which resets the distribution back to the initial state distribution. This is not universally applicable due to irreversible processes in the real world. This is **Forward-Backward RL** (not a particular name in the literature).
Assume we have a forward and backward reward functions $r_{\mathsf{f}}(s,a), r_{\mathsf{b}}(s,a)$ where one reaches the goal state distribution ρ_g and one reaches the initial state distribution ρ_0. We learn a forward policy π_{f} and a backward policy π_{b}. During rollout, we rollout H steps for π_{f} and H steps for π_{b}.

- Requires an additional $r_{\mathsf{b}}(s,a)$

+ Simple to train.

Can we do better than this? From the perspective of π_{f}, it trains from the initial state ρ_0 every time. What if the forward policy can practice from the easier states? This idea is closely related to curriculum learning.

+ Success from easier states can make it faster to learn from harder states.

In comparison to episodic learning where the initial state distribution is fixed, the backward policy π_{b} controls the initial state distribution.
The key insight is to initialise forward policy π_{f} at states an *optimal policy* would visit ρ_*.
The key insight is to train π_{b} to match expert states ρ^* by minimising $\mathrm{D}_{\mathsf{JS}}(\rho_{\mathsf{b}}(s)\|\rho_*(s))$.
The problem is that we don't have either distributions. Rolling out π_{b} is approximately sampling ρ_{b}. We train π_{b} by first training a classifier as a reward function

$$c(s) \to \begin{cases} +1 & s \in \text{demos} \\ -1 & s \sim \rho_{\mathsf{b}}(\cdot) \end{cases}$$

Now we can train policies $\pi_{\mathsf{f}}, \pi_{\mathsf{b}}$

$$\pi_{\mathsf{f}} = \arg\max \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_{\mathsf{f}}(s_t, a_t)\right]$$
$$\pi_{\mathsf{b}} = \arg\max -\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \log(1 - c(s_{t+1}))\right]$$

The classifier $c(s)$ is

$$c(s) = \frac{\rho_*(s)}{\rho_{\mathsf{b}}(s) + \rho_*(s)}$$
$$\frac{\rho_{\mathsf{b}}(s)}{\rho_*(s)} = \frac{1 - c(s)}{c(s)}$$

which can be used as the training objective:

$$\mathrm{D}_{\mathsf{KL}}(\rho_{\mathsf{b}}(s)\|\rho_*(s)) = \int \rho_{\mathsf{b}}(s) \log \frac{\rho_{\mathsf{b}}(s)}{\rho_*(s)} \,\mathrm{d}s = \int \rho_{\mathsf{b}}(s) \log \frac{1 - c(s)}{c(s)} \,\mathrm{d}s$$

Question: What if the initial classifier is suboptimal?

The classifier in real cases pick up very quickly in comparison of the policy. The problem is this it he policy cannot catch up and reach the expert distribution.

This is the **Matching Expert Distributions for Autonomous Learning** (**MEDAL**) algorithm

- Requires expert demos
- Adversarial training can be tricky
+ Prevents the agent from drifting away

Challenge: We don't have a reward function $r_{\mathsf{f}}(s, a)$ for the forward policy. The solution is to use classifier-based rewards again.

15.2 QWALE/single-life RL

In **Single-Life RL**, the agent has *one life* to complete some task given previous training data. The task has hurdles that have never been presented to the agent before.
Deploying a policy and even fine-tuning online does not help the agent recover from a bad state. This problem is because of bias towards prior data.
In **Q-Weighted Adversarial Learning** (**QWALE**), we train the agent to stay close to states visited in previous data. A technique to reach a state distribution is to use a reward classifier.

$$c(s) \to \begin{cases} +1 & s \in \text{prior data} \\ -1 & s \in \text{online data} \end{cases}$$

Then we weigh all prior states when training the classifier by $\exp Q(s, a)$. QWALE trains policy π with the reward $r(s') - \log(1 - c(s'))$.
Note: The slides aren't available at the time of writing this and I can't keep up with the speed of slide changing!

Question: What if the state is continuous?

The trained classifier $c(s)$ will smooth out the states.

Question: Why not build the environment in simulation and train in such an environment?

It would be difficult to collect enough data to create a simulator which has high fidelity dynamics.

We have not covered

- How to handle irreversible states

[RL So far]

- Autonomous agents are taking over the web

Question: Can we have parallel reward functions? Can we have multiple goals/rewards?

Combine reward functions?

16 Hierarchical RL and Skill Discovery

Announcements

- 05-24: Project Milestone due
- 05-31: HW4 due

so far, we have seen online, off-policy, and offline reinforcement learning. What if we want to have the robot autonomously discover interesting behaviours? Some research from biology shows that autonomous skill discovery is possible. A reason for skill discovery is that coming up with tasks is tricky.

Example

Suppose there is a robot in front of a table and it has a gripper. What are some of the tasks we can come up? e.g. lifting an object, placing an object, pushing an object

Hierarchical RL refers to performing tasks at various levels of abstractions.

Excursion: Entropy

Consider a distribution $p(x)$. The **entropy** is defined as $\mathrm{H}\, p(x) := -\mathbb{E}_{x\sim p(x)}[\log p(x)]$. For one x, $-\log p(x)$ is higher if $p(x)$ is lower. In one sense the entropy tells us how broad $p(x)$ is.
The **KL Divergence** is a type of "distance" (not really a metric since it's asymmetric) between two distributions:

$$\mathrm{D}_{\mathsf{KL}}(q\|p) = \mathbb{E}_q\left[\frac{\log q(x)}{\log p(x)}\right] = \mathbb{E}_q \log q(x) - \mathbb{E}_q \log p(x) = -\mathbb{E}_q \log p(x) - \mathrm{H}\, q(x)$$

The **Mutual Information** $\mathrm{I}(x;y)$ of a joint distribution is

$$\mathrm{I}(x;y) = \mathrm{D}_{\mathsf{KL}}(p(x,y)\|p(x)p(y)) = \mathbb{E}_{(x,y)\sim p(x,y)}\left[\log \frac{p(x,y)}{p(x)p(y)}\right]$$

Mutual information represents the amount of information one distribution has for the other. We have

$$\mathrm{I}(x,y) = \mathrm{H}(p(y)) - \mathrm{H}(p(y|x))$$

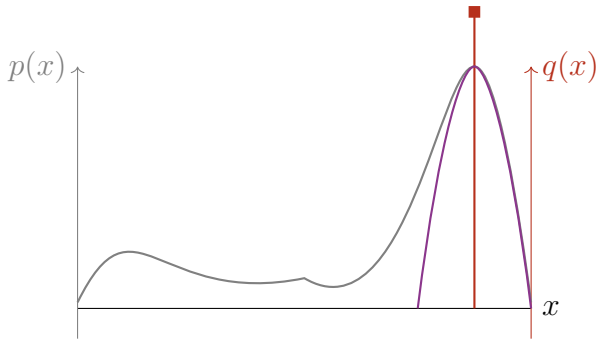


Figure 16.1: The minimiser of $-\mathbb{E}_q \log p(x)$ compared to the minimiser of $\mathrm{D}_{\mathsf{KL}}(q\|p)$

16.1 Skill Discovery

The **empowerment** (defined by Polani et al.) is

$$\mathrm{I}(s_{t+1}; a_t) := \mathrm{H}(s_{t+1}) - \mathrm{H}(s_{t+1}|a_t)$$

which represents the amount of control a_t has over the environment of the next state.
In **Soft Q-Learning**, the objective is replaced by

$$\sum_t \mathbb{E}_{(s_t,a_t)\sim Q}[r(s_t, a_t) + \mathrm{H}(a_t|s_t)]$$

and the policy by softmax

Algorithm: Fitted Q-iteration

1. Collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy, where s'_i is the next state.
2. $y_i \leftarrow r(s_i, a_i) + \gamma \operatorname{softmax}_{a'_i} \mathsf{Q}_\phi(s'_i, a'_i)$ (Bellman Equation)
3. $\phi \leftarrow \arg\min_\phi \frac{1}{2} \sum_i \|\mathsf{Q}_\phi(s_i, a_i) - y_i\|^2$
4. Repeat (2) – (3)
5. $\pi(a|s) \propto \exp A(s_t, a)$

In a multi-task setting with policy $\pi(a|s, z)$ where z is the task index, we cannot just use MaxEnt RL for two reasons:

1. Action entropy is not the same as state entropy

   Agent can take different actions but land in similar states

2. MaxEnt policies are stochastic but not always controllable.

   Intuitively, we want low density for a fixed z and high diversity across z's.

We would like a diversity-promoting reward function.

$$\pi(a|s,z) := \arg\max_{\pi} \sum_{z} \mathbb{E}_{s\sim\pi(s|z)}[r(s,z)]$$

we can define $r(s,z) := \log p(z|s)$
This has a connection to mutual information

$$\mathrm{I}(z,s) = H(z) - H(z|s)$$

Maximised by using uniform prior $p(z)$ — Minimised by maximising $\log p(z|s)$

16.2 Slightly Different Mutual Information

The problem with the above algorithm is that the learned skill may not be very useful or easy to use.

$$\max I(s',z|s) = \max(\mathrm{H}(s'|s) - \mathrm{H}(s'|s,z))$$

We are learning a skill-dynamics model $q(s'|s,z)$ compared to conventional global dynamics $p(s'|s,a)$. Skills are optimized specifically to make skill-dynamics easier to model. This is **Dynamics-Aware Discovery of Skills**.
We can use skill-dynamics for model-based planning. Based on the z fed into the state-dynamics model q, we can predict which state the model will be in. Tasks can be learned zero-shot.

16.3 Hierarchical RL

17 RL in the Real World: From Chip Design to LLMs

RL for Chip Design and LLMs (presented by Anna Goldie)

Announcements

- 05-31: HW4 due
- 06-07: Poster due

18 Connecting the dots

Announcements

- 06-07: Poster due
- 06-12: Project report due

Recurring themes:

- Efficient learning requires controlling distribution shift
- Learned functions can be exploited when optimised against
- Trade-off between computational and data efficiency

Challenges:

- Core algorithms
  - Data/Computational Efficiency
  - Stability
  - Offline workflow
- Assumptions
  - Formulating the problems in the context of MDPs
  - Is MDP even the right problem formulation?

18.1 Research Lightning Talks

1. Joey Hejna: Distance weighted supervised learning
2. Annie Xie: In-Context decision making (DPT)

   Question: Does the context get the entire history as input? If so how does it handle very long horizon tasks?

3. *Q-Transformer: Scalable Offline Reinforcement Learning*:
4. Paul Wohlhart: DeepRL@Scale
5. *AlpacaFarm*