NO UNIVERSITY, CS 000

RL Notes: Theory to Coding

Author: Dr. Huang

Huang Liping

2024 Spring, Singapore

Contents

1 BNN

1.1 Introduction

Bayesian neural network (BNN) combines neural network with Baysian inference. Simply speaking, in BNN, we treat the weights and outputs as the variables and we are finding their marginal distribution that best fit the data as in 1.1. **The ultimate goal of BNN is to quantify the uncertainty introduced by the models in terms of outputs and weights so as to explain the trustworthiness of the prediction**. source at web

Bayesian neural network can be considered as the extension of the standard neural network. The differences as in Fig. 1.2 are:

- **Goal**: SNN focuses on **optimization** while BNN focuses on **marginalization**. Optimization would find one optimal point to represent a weight while marginalization would treat each weight as a variable and find its distribution.
- **Estimation**: the parameters estimation of SNN would be maximum likelihood estimator (MLE) while for BNN, the estimation would be rather maximum a posteriori (MAP) or predictive distribution
- **Method**: basically, SNN would use differentiation to find the optimal value such as gradient descent. In BNN, since the sophisticated integrals are hard to determine, researchers would always rely on *Markov Chain Monte Carlos (MCMC), Variational Inference, and Normalizing Flows techniques.*

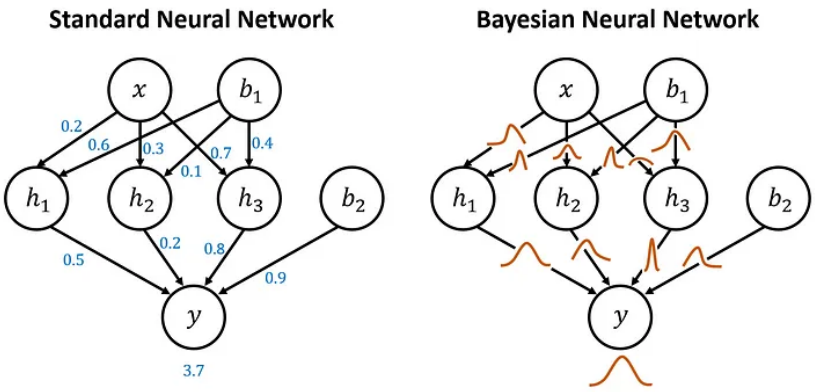

Figure 1.1: Comparing SNN and BNN.

	Standard Neural Network	Bayesian Neural Network
Goal	*Optimization*	*Marginalization*
Weight	*A Single Set*	*Probabilistic Distribution*
Method	*Differentiation (Gradient Descent)*	*Markov Chain Monte Carlos Variational Inference Normalizing Flows*
Estimate	*Maximum Likelihood Estimators*	*Maximum A Posteriori Full / Approximate Predictive Distribution*

Figure 1.2: Comparing SNN and BNN.

Uncertainties:

- **Aleatory Uncertainty (Statistical Uncertainty)**: In statistics, it's representative of unknowns that differ each time we run the same experiment (train the model). In Deep Learning, it refers to the **the uncertainty of the model outputs**.
- **Epistemic Uncertainty (systmetic uncertainty)**: In Deep Learning, it refers to the **uncertainty of the model weights**.

Pros:

- getting a prediction interval. It allows to automatically calculate the uncertainties associated with the prediction when dealing with unknown targets.
- training a robust model. Instead of taking into account just a single set of weights, BNN would find the distributions of the weights. By catering to the probability distributions, it can avoid the over-fitting problem by addressing the regularization properties.

Cons: demanding maths and stats knowledge. longer training epoches to converge.

1.2 What is BNN

The Bayes's theorem

$$P(H|D) = \frac{P(D|H)P(H)}{P(D)} \tag{1.1}$$

where, $P(H|D)$ is the posterior, $P(H)$ is the prior, $P(D|H)$ is the likelihood, $P(D)$ is the evidence.

Stochastic neural networks are a type of ANN built by introducing stochastic components into the network as in Fig. 1.3. This is performed by giving the network either a stochastic activation or stochastic weights. BNN can be considered a special case of ensemble learning. The main goal of using a stochastic nn architecture is to obtain a better idea of the uncertainty associated with the underlying processes.

This is accomplished by comparing the predictions of multiple sampled model parametrization θ. If the different models agree, then the uncertainty is low. If they disagree, then the uncertainty is high. This process can be summarized as

$$\begin{aligned} \theta &\sim p(\theta) \\ \mathbf{y} &= \mathbf{\Phi}_\theta(\mathbf{x}) + \epsilon \end{aligned} \tag{1.2}$$

where ϵ represents random noise to account for the fact that the function Φ is only an approximation. A BNN can then defined as any stochastic artificial neural network trained using Bayesian Inference.

The BNN with stochastic weights for **regression** could represent the following data generation processes:

$$\begin{aligned} \theta &\sim p(\theta) = \mathcal{N}(\mu, \Sigma) \\ \mathbf{y} &\sim p(\mathbf{y}|\mathbf{x}, \theta) = \mathcal{N}(\mathbf{\Phi}_\theta(\mathbf{x}), \Sigma) \end{aligned} \tag{1.3}$$

The choice of using normal laws $\mathcal{N}(\mu, \Sigma)$, with mean μ and covariance $\pm$ is arbitrary but is common in practice because of its good mathematical properties.

The BNN with stochastic weights for **classification** could represent the following generation processes:

$$\begin{aligned} \theta &\sim p(\theta) = \mathcal{N}(\mu, \Sigma) \\ \mathbf{y} &\sim p(\mathbf{y}|\mathbf{x}, \theta) = Cat(\mathbf{\Phi}_\theta(\mathbf{x})) \end{aligned} \tag{1.4}$$

In the case of stochastic activation, refer to Hands-on Bayesian Neural Networks – A Tutorial for Deep Learning Users for details. BNN with stochastic weight is more common in practice. The alternative architecture of using **stochastic activation** is sometimes used as it *allows compressing the number of variational parameters when using variational inference.*

The work flow of BNN is shown in Fig. 1.4.

The concepts comparison between SNN and BNN is shown in Fig. 1.5

1.3 BNN with Pyro

BNN with Pyro Tutorial

```
import pyro
import pyro.distributions as dist
```

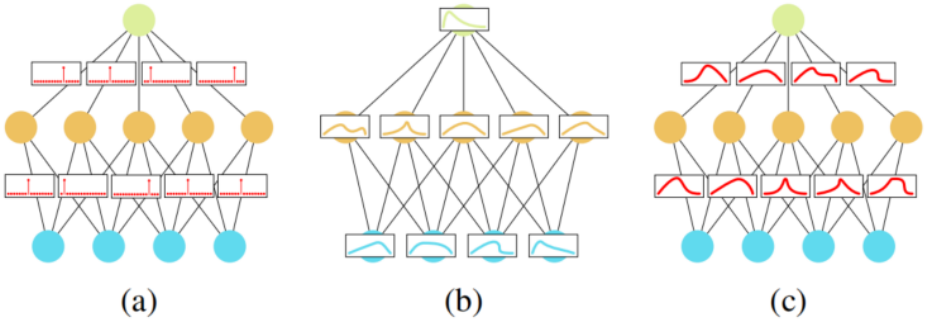

Figure 1.3: (a) point estimate neural network, (b) stochastic neural network with probability distribution for the activation, (c) stochastic neural network with a probability distribution over the weights.

```
from pyro.nn import PyroModule, PyroSample
import torch.nn as nn

from pyro.infer import MCMC, NUTS


from pyro.infer import SVI, Trace_ELBO
from pyro.infer.autoguide import AutoDiagonalNormal
```

1.4 MC Dropout

1.5 Variational Inference

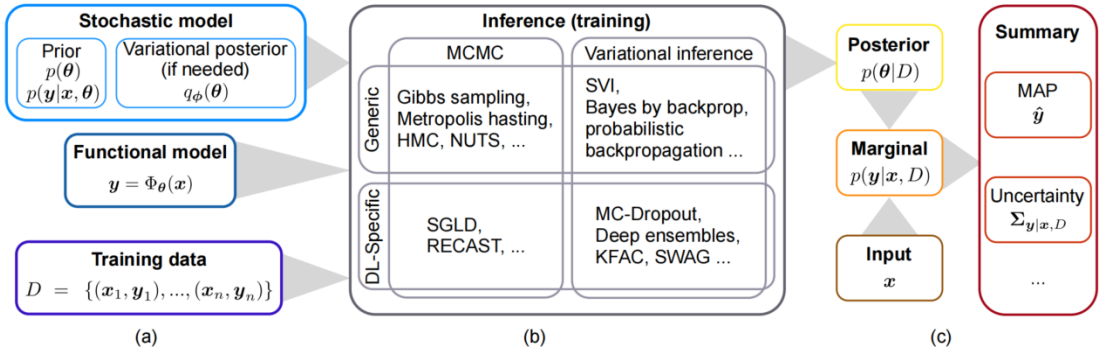

Figure 1.4: Work flow in BNN.

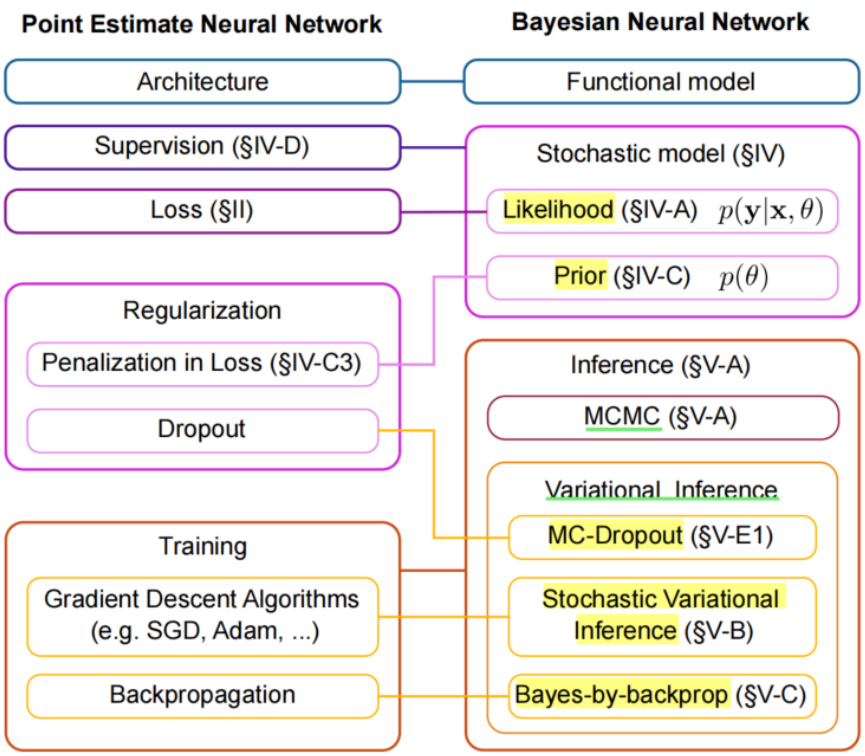

Figure 1.5: Concepts used in SNN and BNN.

2 SAC

2.1 Introduction

- on-policy models: TRPO, PPO, A3C, A2C sample inefficiency.
- off-policy deterministic models: DDPG, TD3, ACER.
- off-policy stochastic models|: Soft Q-learning, SAC

Ref: https://lilianweng.github.io/posts/2018-04-08-policy-gradient/

Actor-critic methods alternate between computing Q^π by iterating the Bellman operator $\mathcal{B}^\pi Q = r + \gamma P^\pi Q$ and improving the policy $\pi(\mathbf{a}|\mathbf{s})$ by updating it towards actions that maximize the expected Q-value.

2.2 Theory: stochastic off-policy RL

Three key ingredients of soft actor-critic:

- an actor-critic architecture with separate policy and value function networks.
- an off-policy formulation that enables reuse of previously collected data for efficiency *.
- entropy maximization to enable stability and exploration.

* For off policy algorithms, agent action policy and target policy are separated. off-policy algorithms do not require the agent to follow the same policy that they generated the data (action). Each time the agent takes an action, it generates a new data, on-policy algorithms learn immediately on this new data. However, off-policy algorithms accumulate the new generated data in the buffer, and resample a batch to estimate the target Q value, where the target Q value is updated by the resamples. Finally, the policy (actor) loss = mse(*target Q value* - policy prediction of the *resamples*). Hence, the policy is learned on the previously collected data. **Coding:** compared to on-policy, off-policy requires replay buffer and a separate target Q model.

For general RL, the policy objective is to maximize the accumulated reward. However, for SAC,

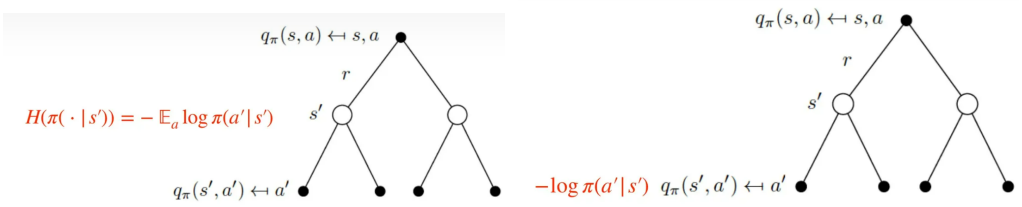

Figure 2.1: Soft Bellman Backup

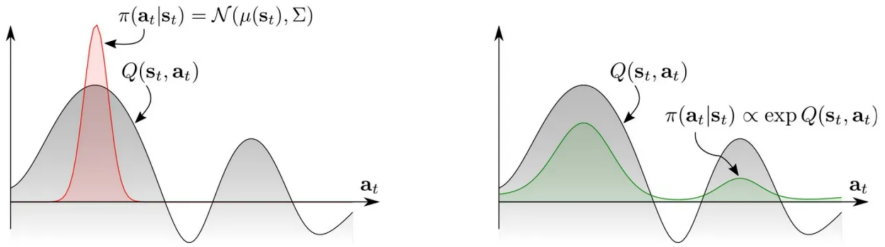

Figure 2.2: Action Distributions: Deterministic RL vs Stochastic RL

The reward is entropy augmented reward instead. The policy objective is

$$\mathcal{J}(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\rho_\pi} \left[r(\mathbf{s}_t, \mathbf{a}_t) + \alpha \mathcal{H}(\pi(\cdot|\mathbf{s}_t)) \right] \tag{2.1}$$

which further requires that the entropy of an action is maximized to make the policy stochastic, i.e., the probability of each action $\pi(\mathbf{a}_i|\mathbf{s}_t)$ is decentralized. As shown inf Fig. 2.1 $\mathcal{H}(\pi(\cdot|\mathbf{s}_t)) = -\mathbb{E}log\pi(a|\mathbf{s})$. Maximum entropy makes it consider any useful actions (trajectories) as shown in Fig. 2.2. The trade-off between expected return and entropy has a close connection to the exploration-exploitation trade-off: increasing entropy results in more exploration, which can accelerate learning later on. It can also prevent the policy from prematurely converging to a bad local optimum.

The policy update of SAC is:

$$\pi_{new} = \underset{\pi'\in\Pi}{\operatorname{argmin}}\, D_{KL}\left(\pi'(\cdot|\mathbf{s}_t) || \frac{exp\left(Q^{\pi_{old}}_{netCrtic}(\mathbf{s}_t, \cdot)\right)}{Z^{\pi_{old}}(\mathbf{s}_t)} \right) \tag{2.2}$$

Theorem 2.1: THEOREM NAME

This is a theorm. Below are equations.

Lemma 2.2: LEMMA NAME

This is a lemma

Proof 2.2: LEMMA NAME

This is a proof. □

2.3 Coding

Actor inputs: state **s**, outputs: action **a** and the log probabllity $log\_prob(\pi(\mathbf{a}|\mathbf{s}))$

Critic input: (state, action), output double Q values, i.e., q_1 and q_2.

2.3.1 Actor

- **actor modeling:** For continuous actions, the actor models the action distribution represented as μ and $\log\sigma$, which are vectors with size of $|\mathcal{A}|$ ($\mathcal{A}$ is the action space), and are parameterized by a neural network (*sharing the same structure between μ and $\log\sigma$*). Here, it models the $\log\sigma$ instead of σ to ensure $\sigma = exp(log\sigma) > 0$. (We can also use other methods, such as utilizing a sigmoid activation function at the last layer of $\log\sigma$). The actor takes the state as its input, and outputs the action vector **a**, $|\mathbf{a}| = |\mathcal{A}|$ and the corresponding log probability value of this sample action **a** $log\_prob(\pi(\mathbf{a}|\mathbf{s}))$.
- **usage of actor outputs:** the actor outputs are used to (1) select action when interacting with env, e.g., return **a** in the select action function; (2) get $\mathbf{a}'$, $log\_prob(\pi(\mathbf{a}'|\mathbf{s}'))$ given the next state $\mathbf{s}'$. They are further fed into calculating the target Q value for updating the Critic. We get the target-network prediction $Q_{netTarget}(\mathbf{s}', \mathbf{a}')$, and calculate the target Q value as $Q_{target} = r + \gamma \times (Q_{netTarget}(\mathbf{s}', \mathbf{a}') - \alpha \times log\_prob(\pi(\mathbf{a}'|\mathbf{s}')))$. (3) get $log\_prob(\pi(\mathbf{a}|\mathbf{s}))$ given the current state **s**, which is used in calculating the actor loss as $\alpha \times log\_prob(\pi(\mathbf{a}|\mathbf{s})) - Q_{netCritic}(\mathbf{s}, \mathbf{a})$
- **coding hints**. (1) How to get the action vectors, and the log probability in Actor: the Actor modeling the action distribution represented as μ and $\log\sigma$. We use the μ, σ to generate a distribution, e.g., $dist \triangleq$ **torch.distributions.Normal**(μ, σ) in Python. x=dist.rsample(), the output action is torch.tanh(x) and the log probability vector can obtained from dist.log_prob(u).sum(). Note here, the output action is tanh(x), so the log probability should be the log probability of tanh(x) instead of x as in Eq. 2.3. Check Sources (2, 3) for implementation.

$$log\pi(\mathbf{a}|\mathbf{s}) = log\mu(\mu|\mathbf{s}) - \sum_{i=1}^{D} log(1 - tanh^2(\mu_i)) \tag{2.3}$$

where μ_i is the i^{th} element of μ.

The $log\_prob(\pi(\cdot|\cdot))$ are fed into (i) calculating target Q value for updating Critic, and (ii) calculating the loss of the Actor (policy model) for updating Actor.

2.3.2 Double-Q Critic

SAC is off-policy, hence we have a general critic and a target critic.

where state $\mathbf{s}$ and action $\mathbf{a}$ are concatenated to construct the network input.

2.3.3 Temperature α

The target entropy of action set $\mathcal{A}$ is $\dim(\mathcal{A})$.

- **Jesen's Inequality**. For concave function g, we have the inequality:

$$\alpha g(x) + (1-\alpha)g(y) \leq g(\alpha x + (1-\alpha y)) \tag{2.4}$$

- maximum value: $\mathcal{H}(S) = \sum_x p(x)(-logp(x)) = \sum_x p(x) log\frac{1}{p(x)} \leq log \sum_x p(x)\frac{1}{p(x)} = log|S|$.

Based on the deduction in openai lilian log, the optimal temporature parameter in every step by minimizing the objective function:

$$\mathcal{J}(\alpha) = \mathbb{E}_{\alpha_t \sim \pi_t}(-\alpha log\pi_t(a_t|s_t) - \alpha\mathcal{H}_0) \tag{2.5}$$

where $\mathcal{H}_0$ is the predefined minimum policy entropy threshold.

2.3.4 Learning

- **update Actor**. The Actor loss is $a\_loss = mean\left(\alpha \times log\_prob(\pi(\mathbf{a}|\mathbf{s})) - Q_{netTarget}(\mathbf{s},\mathbf{a})\right)$. $\mathbf{a}$ and $log\_prob(\pi(\mathbf{a}|\mathbf{s}))$ (log probability of an action $\mathbf{a}_i$) are obtained from the Actor. $Q_{netTarget}(\mathbf{s},\mathbf{a})$ is obtained from the Critic.

  *Note: $V(\mathbf{s}) = \mathbb{E}_{\mathbf{a}\sim\pi}[Q_{netTarget}(\mathbf{s},\mathbf{a}) - log\pi(\mathbf{a}|\mathbf{s})]$ is the soft state value. Maximizing the soft state value is to minimizing the negative of the soft state value.

  The entropy of policy π is the Expectation of $-log\_prob(\pi)$. Assume that π is drawn from Normal distribution, then the entropy of policy π is calculated as $mean(-\alpha \times log\_prob(\pi(\mathbf{a}|\mathbf{s})))$. Hence *a_loss* aligns with the objective function of the soft policy in Eq. 2.1.

  $D_{KL}(P||Q) = \sum_{x\in\mathcal{X}}\left(P(x)log\left(\frac{P(x)}{Q(x)}\right)\right)$ denotes the logarithmic difference between the probabilities P and Q. By replacing $P = \pi'$ and $Q = exp\left(Q^{\pi_{old}}_{netTarget}(\mathbf{s}_t,\cdot) - logZ^{\pi_{old}}(\mathbf{s}_t)\right)$ from

Eq. 2.2, we can find that minimizing D_{KL} aligns with minimizing a_loss, which means updating the Actor is the same way to get π_{new}. The Lemma 2 denotes $Q^{\pi_{new}}(\mathbf{s}_t, \mathbf{a}_t) \geq Q^{\pi_{old}}(\mathbf{s}_t, \mathbf{a}_t)$, which means the Q value is improved by updating the soft policy network (Actor).

- **update Critic**. The Critic loss is mse between the current Q value (from $Q_{netCritic}$) and the target Q value, where the target Q value is the sum of current reward r and the discounted Q prediction for the next state s', which is calculated based on Eq. 2.1, i.e., $Q_{target} = r + \gamma \times (Q_{netTarget}(\mathbf{s}', \mathbf{a}') - \alpha \times log\_prob(\pi(\mathbf{a}'|\mathbf{s}')))$. Note: the Critic is a double-Q network, hence both current Q values are used in the loss function.

  Understanding: definition of Q (or reward) aligns with the Actor (policy) objective (loss) function and its updating equation (π_{new}).

- **update** α. $\mathcal{H}_0$ is set as $-|\mathcal{A}|$, which is the target entropy. α loss is calculated as

$$\alpha_{loss} = -(log\alpha * (log\pi + \mathcal{H}_0)) \tag{2.6}$$

2.3.5 Procedures

- **interaction between agent and env:**

  agent.select_action -> env.step() -> agent.replay_buffer.push() -> agent.train().

- **agent training**:

2.3.6 Sources

- 0: spinningup (github openai)
- 1: github rlkit
- 2: github (XinjiangHao)
- 3: garage
- 4: github stable baseline)
- 5. sac understanding at zhihu
- 6. openai LiLian Log

2.4 Summary

2.4.1 coding the theory

- stochastic property in theory and coding:

- off-policy

2.4.2 Variants

What we can do for define the Actor:

- stochastic to deterministic
- continuous and discrete
- variate Actor networks: e.g., involving CNN.
- using other distributions
- context conditioned policy: we may define the conditional probability of the policy with the given updated context. Check sources 3.

(1) change the network (2) change the distribution

3 CQL

3.1 Introduction

For conservative off-policy evaluation is to prevent overestimating the policy value. Hence, a penalty is involved in the Q value function, minimizing the expected Q value under a particular distribution of state-action pairs, $\mu(\mathbf{s}, \mathbf{a})$.

Note that standard Q-function training does not query the Q-function value at unobserved states, but queries the Q-function at unseen actions.

Concept to clear: (0) data sampling and procedure for updating policy and Q function, what are the problems involved in the data sampling and procedure,e.g., OOD problem, **bias sampling**, theories involved, etc.

Naive Q-learning in RL is defined as $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{P}(s'|s, a)$ is the transition distribution, $\mathcal{R}(s, a)$ is the reward function, and $\gamma \in (0, 1)$ is the discount factor. The goal in RL is to find a policy $\pi(a|s)$ that maximizes the expected cumulative discounted rewards. In Q-learning, it learns the optimal state-action value function $Q^*(s, a)$, which represents the expected cumulative discounted rewards starting in s taking action a, and then acting optimally thereafter by iterating the Bellman optimality operator $\mathcal{B}$, defined as 3.1. The optimal policy $\pi*$ can be recovered from $Q*$ by choosing the maximizing action a'.

$$\mathcal{B}\hat{Q}(s, a) = \mathcal{R}(s, a) + \gamma \mathbb{E}_{s' \sim \mathcal{P}(s'|s,a)} [\max_{a'} \hat{Q}(s', a')] \tag{3.1}$$

How to conduct Bellman optimality operator?

- when the state space $\mathcal{S}$ is large, we represent $\hat{Q}$ as a hypothesis from the set of function approximators $\mathcal{Q}$ (e.g., neural networks). In theory, the estimate of the Q-function is updated by projecting 3.1 into $\mathcal{Q}$, i.e., minimizing the mean squared Bellman error, which is calculated as

  $$\mathcal{L}_Q = \mathbb{E}_{\mathcal{V}} [(Q - \mathcal{B}\hat{Q})^2] \tag{3.2}$$

  where $\mathcal{V}$ is the state occupancy measure under the behavior policy.

- optimal Q value, the approximator $\mathcal{Q}$ returns the maximum value of $\mathcal{Q}(\mathbf{s}', \mathbf{a}'_k)$, where $\mathbf{a}'_{best}$ is the action that gets the maximum Q value for the input $\mathbf{s}'$.

Problems 1 (solved by learning a policy π_θ)

- in large action spaces (e.g., continuous), the maximization in Eq. 3.1 is generally intractable.
- **Actor-Critic** solution: learning a policy π_θ that maximizes the Q-function.

Problem 2

- **out-of-distribution (OOD) actions**: a'_{best} (i.e., $\mathbf{a}'_k$ for discrete action) is derived from the current Q-function estimator Q, which may lie far outside of the training distribution. However, Q is only reliable on inputs from the same distribution as its training dataset.

  Naively maximizing the value on the OOD actions, i.e., a' is OOD for $\max_{a'} \hat{Q}(s', a')$, may results in pathological values that incur large error.
- Batch-Constrained Q-learning (**BCQ**) solution: implicitly constrains the distribution of the learned policy to be close to the behavior policy, similarly to behavioral cloning (**BC**). But overly restrictive. For example, if the behavior policy is close to uniform, the learned policy will behave randomly, resulting in poor performance, even when the data is sufficient to learn a strong policy (check this in Fig. 2 of the BEAR paper).
- Bootstrapping Error Accumulation Reduction (**BEAR**) solution: the learned policy $\pi(a|s)$ has positive density only where the density of the behaviour policy $\pi_\beta(a|s)$ is more than a threshold, i.e., $\forall a, \pi(a|s) \leq \epsilon \Rightarrow \pi(a|s) = 0$, instead of a closeness constraint on the value of the density $\pi(a|s)$ and $\pi_\beta(a|s)$. It restricts the support of the learned policy, but not the probabilities of the actions lying with the support.

3.2 CQL Theory

3.2.1 OOD Actions in Q-Learning

Let $\zeta_k(s,a) = |Q_k(s,a) - Q^*(s,a)|$ denotes the total error at iteration k of Q-learning, and let $\delta_k(s,a) = |Q_k(s,a) - \mathcal{T}Q_{k-1}(s,a)|$ denote the current Bellman error. Then we have

$$\zeta_k(s,a) \leq \delta_k(s,a) + \gamma \max_{a'} \mathbb{E}_{s'}[\zeta_{k-1}(s',a')] \tag{3.3}$$

We expect $\delta_k(s,a)$ to be high on OOD states and actions, as errors at these state-actions are never directly minimized while training.

3.2.2 CQL Framework

- lower bounds the Q-value
- incorporate into the policy learning procedure

* Further understand SAC from these two aspects.

In practice, CQL augments the standard Bellman error objective with a simple Q-value regularizer, which is straightforward to implement on top of existing deep Q-learning and actor-critic implementations. Its performance improvement is significant, especially when learning from complex and multi-modal data distributions.

Actor-critic methods alternate between computing Q^π by iterating the Bellman operator $\mathcal{B}^\pi Q = r + \gamma P^\pi Q$ and improving the policy $\pi(\mathbf{a}|\mathbf{s})$ by updating it towards actions that maximize the expected Q-value.

For offline RL, let $\pi_\beta(\mathbf{a}|\mathbf{s})$ be the behaviour policy that generates the dataset $\mathbf{D}$, $d^{\pi_\beta}(\mathbf{s})$ be the discounted marginal state distribution of $\pi_\beta(\mathbf{a}|\mathbf{s})$. Hence, the dataset $\mathcal{D}$ is sampled from $d^{\pi_\beta}(\mathbf{s})\pi_\beta(\mathbf{a}|\mathbf{s})$. On all states $\mathbf{s} \in \mathcal{D}$, let

$$\hat{\pi}_\beta(\mathbf{a}|\mathbf{s}) := \frac{\sum_{\mathbf{s},\mathbf{a}\in\mathcal{D}} \mathbf{1}[\mathbf{s}=\mathbf{s},\mathbf{a}=\mathbf{a}]}{\sum_{\mathbf{s}\in\mathbf{D}} \mathbf{1}[\mathbf{s}=\mathbf{s}]} \tag{3.4}$$

denote the empirical behavior policy at that state.

Since $\mathcal{D}$ typically does not contain all possible transitions $(\mathbf{s}, \mathbf{a}, \mathbf{s}')$, the policy evaluation step actually uses an empirical Bellman operator that only backs up a single sample. However, the policy may suffer from state distribution shift at test time.

For offline RL, the garget values for Bejllman backups in policy evaluation use actions sampled from the learned policy, but, the Q-function is trained only on actions sampled from the behavior policy that produced the dataset $\mathcal{D}$, π_β. For offline RL, the OOD problem exists in:

- Since π is trained to maximize Q-values, it may be biased towards OOD actions with erroneously high Q-values. Typical offline RL methods mitigate this problem by constraining the learned policy away from OOD actions.
- Note that Q-function training in offline RL does not suffer from state distribution shift, as the Bellman backup never queries the Q-Function on OOD states. However, the policy may suffer from state distribution shift at test time.

Since standard Q-function training does not query the Q-function value at unobserved states, but queries the Q-function at unseen actions, CQL restricts state-action pair $\mu(\mathbf{s}, \mathbf{a})$ to match the state-marginal in the data set, such that $\mu(\mathbf{s}, \mathbf{a}) = \mu(\mathbf{s}, \mathbf{a})$. The iterative update fro training the Q-function is:

$$\hat{Q}^{k+1} \leftarrow \arg\min_Q \alpha \mathbb{E}_{\mathbf{s}\sim\mathcal{D},\mathbf{a}\sim\mu(\mathbf{a}|\mathbf{s})}[Q(\mathbf{s},\mathbf{a})] + \frac{1}{2}\mathbb{E}_{\mathbf{s},\mathbf{a}\sim\mathcal{D}}\left[Q(\mathbf{s},\mathbf{a}) - \hat{\mathcal{B}}^\pi \hat{Q}^k(\mathbf{s},\mathbf{a})\right] \tag{3.5}$$

where $\alpha \geq 1$ is a trade-off factor. The resulting Q-function $\hat{Q}^\pi := lim_{n\to\infty}\hat{Q}^k$ lower-bounds $\hat{Q}^\pi$ at all $\mathbf{s} \in \mathcal{D}, \mathbf{a} \in \mathcal{A}$l, as shown in Theorem 1.

If we only require that the expect value of the $\hat{Q}^\pi$ under $\pi(\dashv|\int)$ lower-bound V^π, the iterative update is changed to:

$$\hat{Q}^{k+1} \leftarrow \arg\min_{Q} \alpha(\mathbb{E}_{\mathbf{s}\sim\mathcal{D},\mathbf{a}\sim\mu(\mathbf{a}|\mathbf{s})}[Q(\mathbf{s},\mathbf{a})] - \mathbb{E}_{\mathbf{s}\in\mathcal{D},\mathbf{a}\in\hat{\pi}_\beta(\mathbf{a}|\mathbf{s})}[Q(\mathbf{s},\mathbf{a}]) + \frac{1}{2}\mathbb{E}_{\mathbf{s},\mathbf{a}\sim\mathcal{D}}\left[Q(\mathbf{s},\mathbf{a}) - \hat{\mathcal{B}}^\pi\hat{Q}^k(\mathbf{s},\mathbf{a})\right] \tag{3.6}$$

where the red part is an additional Q-value maximization term under the data distribution.

Theorem 3.1: THEOREM NAME

This is a theorm. Below are equations.

Theorem 3.2: THEOREM NAME

This is a theorm. Below are equations.

3.3 Coding

3.3.1 Sources

SAC and Conservative SAC at Git

3.4 BEAR

3.4.1 BEAR-QL

The policy improvement is in Eq. 3.7, and the algorithm of BEAR is in Fig. 3.1. Below are the proof for the policy convergence.

$$\pi_\phi := \max_{\pi\in\Delta_{|S|}} \mathbb{E}_{s\sim\mathcal{D}}\mathbb{E}_{a\sim\pi(\cdot|s)}\left[\min_{j=1,2,..,K}\hat{Q}_j(s,a)\right], s.t.\mathbb{E}_{s\sim\mathcal{D}}\left[MMD(\mathcal{D}(s),\pi(\cdot|s))\right] \leq \epsilon \tag{3.7}$$

It uses the maximum mean discrepancy (MMD) to approximately constrain π to Π, where the MMD estimate can be solely on samples from the distributions. Note that the expression for MMD does not involve the density of either distribution and it can be optimized directly through samples. MMD estimates serve as constraining distributions to a given support set. The sampled version of MMD between the unknown behaviour policy β and the actor π is used. Given samples $x_1, \cdots, x_n \sim P$ and $y_1, \cdots, y_m \sim Q$, the sampled MMD between P and Q is given by

$$MMD^2(x_1,\cdots,x_n,y_1,\cdots,y_m) = \frac{1}{n^2}\sum_{i,i'}k(x_i,x_{i'}) - \frac{2}{nm}\sum_{i,j}k(x_i,y_j) + \frac{1}{m^2}\sum_{j,j'}k(y_j,y_{j'}) \tag{3.8}$$

Algorithm 1 BEAR Q-Learning (BEAR-QL)

input : Dataset $\mathcal{D}$, target network update rate τ, mini-batch size N, sampled actions for MMD n, minimum λ

1: Initialize Q-ensemble $\{Q_{\theta_i}\}_{i=1}^{K}$, actor π_ϕ, Lagrange multiplier α, target networks $\{Q_{\theta'_i}\}_{i=1}^{K}$, and a target actor $\pi_{\phi'}$, with $\phi' \leftarrow \phi, \theta'_i \leftarrow \theta_i$
2: **for** t in $\{1, \ldots, \mathrm{N}\}$ **do**
3: Sample mini-batch of transitions $(s, a, r, s') \sim \mathcal{D}$
Q-update:
4: Sample p action samples, $\{a_i \sim \pi_{\phi'}(\cdot|s')\}_{i=1}^{p}$
5: Define $y(s,a) := \max_{a_i}[\lambda \min_{j=1,..,K} Q_{\theta'_j}(s', a_i) + (1-\lambda)\max_{j=1,..,K} Q_{\theta'_j}(s', a_i)]$
6: $\forall i, \theta_i \leftarrow \arg\min_{\theta_i}(Q_{\theta_i}(s,a) - (r + \gamma y(s,a)))^2$
Policy-update:
7: Sample actions $\{\hat{a}_i \sim \pi_\phi(\cdot|s)\}_{i=1}^{m}$ and $\{a_j \sim \mathcal{D}(s)\}_{j=1}^{n}$, n preferably an intermediate integer(1-10)
8: Update ϕ, α by minimizing Equation 1 by using dual gradient descent with Lagrange multiplier α
9: **Update Target Networks:** $\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$; $\phi' \leftarrow \tau\phi + (1-\tau)\phi'$
10: **end for**

Figure 3.1: Algorithm of BEAR. Note: (1) the target Q value $y(s,a)$ calculation is a trade-off between the min and max value. (2) the policy update is based on Eq. 3.7. (3) the Lagrange multiplier α is also updated. (4) It also has a target actor, and hence a target actor network update.

Here, $k(\cdot,\cdot)$ is any universal kernel. In the experiments, Laplacian and Gaussian kernels work well.

3.4.2 BEAR Theory

This section describes the theory and corresponding proof of BEAR as it is in the original paper.

Definition 3.3

Given a set of policies Π, the **distribution-constrained backup operator** is defined as:

$$\mathcal{T}^{\Pi}Q(s,a) \stackrel{\Delta}{=} \mathbb{E}\left[R(s,a) + \gamma \max_{\pi\in\Pi} \mathbb{E}_{P(s'|s,a)}\left[\max_{\pi\in\Pi} \mathbb{E}_{\pi}[Q(s',a')]\right]\right] \tag{3.9}$$

Definition 3.4

The **suboptimality constant** is defined as:

$$\alpha(\Pi) = \max_{s,a} |\mathcal{T}^{\Pi}Q^*(s,a) - \mathcal{T}Q^*(s,a)| \tag{3.10}$$

Definition 3.5

Let ρ_0 denote the initial state distribution, and $\mu(s,a)$ denote the distribution of the trianing data over $\mathcal{S} \times \mathcal{A}$, with marginal $\mu(s)$ over $\mathcal{S}$. Suppose there exist coefficients $c(k)$ such that for any $\pi_1, ..., \pi_k \in \Pi$ and $s \in \mathcal{S}$:

$$\rho_0 P^{\pi_1} P^{\pi_2} ... P^{\pi_k} \leq c(k)\mu(s) \tag{3.11}$$

where P^{π_k} is the transition operator on states induced by π_i. Then, define the **concentrability coefficient** $C(\Pi)$ as

$$C(\Pi) \triangleq (1-\gamma)^2 \sum_{k=1}^{\infty} k\gamma^{k-1} c(k) \tag{3.12}$$

It quantifies how far the visitation distribution generated by policies from Π is from the training data distribution.

Theorem 3.6: Bellman Error Bound for Constrained Backup $\mathcal{T}^{\Pi}$

Supporse we run approximate distribution-constrained value iteration with a set constrained backup $\mathcal{T}^{\Pi}$. Assume that $\delta(s,a) \geq \max_k |Q_k(s,a) - \mathcal{T}^{\Pi} Q_{k-1}(s,a)|$ bounds the Bellman error. Then,

$$\lim_{k \to \infty} \mathbb{E}_{\rho_0} \left[|V^{\pi_k}(s) - V*(s)| \right] \leq \frac{\gamma}{(1-\gamma)^2} \left[C(\Pi) \mathbb{E}_{\mu} [\max_{\pi \in \Pi} \mathbb{E}_{\pi} [\delta(s,a)]] + \frac{1-\gamma}{\gamma} \alpha(\Pi) \right] \tag{3.13}$$

Theorem 3.7: concentrability coeeficient bound

$$C(\Pi_\epsilon) \leq C(\beta) \left(1 + \frac{\gamma}{(1-\gamma) f(\epsilon)} (1-\epsilon) \right) \tag{3.14}$$

where $f(\epsilon) \triangleq \min_{s \in \mathcal{S}, \mu_{\Pi_\epsilon} > 0} [\mu(s)] > 0$,

(1) OOD in RL, boosttrapping error, how to stabilizing, BEAR
(2) how to bound Q-function, what's the lower-bound and upper-bound, what's the conservative Q-learning (CQL)
(3) what's behaviour cloning
(4) what's batch-constrained Q-learning (BCQ)
(5) BRAC

Instructor: Ins Tructor1

4 Introduction to Easy Class

4.1 Introduction

This is a test

4.2 Table

Right	Left	Longlonglonglonglonglonglonglong longlonglonglonglonglong-longlonglonglonglonglonglong longlonglonglonglonglong
Right	Left	Longlonglonglonglonglonglong longlonglonglonglonglonglong-long longlonglonglong longlonglonglonglonglonglonglong

Table 4.1: This is a caption

4.3 List

This is a List:

- **Bullet 1**: Bullet 1 is bullet 1.
- **Bullet 2**: Bullet 2 is bullet 2.

4.4 Definition

Definition 1. ***DEFINITION NAME****: This is a definition.*

4.5 Theorem

Theorem 4.1: THEOREM NAME

This is a theorm. Below are equations.

$$\psi(\mathbf{a}) = A \cdot \mathbf{a} + \mathbf{t}. \tag{4.1}$$

$$R_x = \begin{bmatrix} 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \\ 1 & 0 & 0 \end{bmatrix}, R_y = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ \sin(\theta) & 0 & \cos(\theta) \\ 0 & 1 & 0 \end{bmatrix}, R_z = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

Lemma 4.2: LEMMA NAME

This is a lemma

Proof 4.2: LEMMA NAME

This is a proof. □

4.6 Tikz Pictures

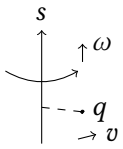

Figure 4.1: This is a caption.

Instructor: Ins Tructor1