Notes on Imitation Learning: Behavioral Cloning and DAgger

yihua

2025 年 10 月 17 日

0.1 Behavioral Cloning and DAgger

你好

Overview. Behavioral Cloning (BC) and Dataset Aggregation (DAgger) are two fundamental algorithms in imitation learning. BC formulates imitation as a supervised learning problem on expert demonstrations, while DAgger addresses the covariate shift that arises when the learned policy deviates from the expert’s state distribution.

Behavioral Cloning. Let $\pi_\theta(a|s)$ be a parameterized policy with parameters θ, and let $\mathcal{D}_E = \{(s_i, a_i)\}_{i=1}^N$ be a set of expert demonstrations sampled from the expert distribution $p_{\pi_E}(s)$. The BC objective minimizes the negative log-likelihood (or equivalently, a regression loss) between the expert actions and the policy outputs:

$$\min_\theta \ \mathbb{E}_{(s,a)\sim\mathcal{D}_E}\left[-\log \pi_\theta(a|s)\right] \quad \text{or} \quad \min_\theta \ \mathbb{E}_{(s,a)\sim\mathcal{D}_E}\left[\|a - \pi_\theta(s)\|^2\right].$$

The policy is trained purely on the expert's state distribution $p_{\pi_E}(s)$. However, during deployment, the learned policy induces its own state distribution $p_{\pi_\theta}(s)$, which gradually drifts away from $p_{\pi_E}(s)$. This discrepancy leads to a phenomenon known as **compounding error**: small action errors accumulate over time, pushing the policy into unseen states where its behavior is unreliable.

Covariate Shift. Formally, the expected performance degradation grows quadratically with the horizon T:

$$J(\pi_\theta) - J(\pi_E) = \mathcal{O}(T^2\epsilon),$$

where ϵ denotes the per-step imitation error on the expert distribution. This quadratic scaling explains why BC performs poorly in environments such as Walker2d, where small deviations quickly result in loss of balance or termination.

Empirical Observation. In our experiments, BC achieves high training performance but low evaluation returns:

- For example, on `Walker2d-v4`, the BC model reaches Train_AverageReturn $\approx$ 5400, while Eval_AverageReturn $\approx$ 2300.
- This reflects the distribution mismatch: BC fits well on expert-like states but fails to generalize when encountering out-of-distribution states during rollouts.

Dataset Aggregation (DAgger). To mitigate the covariate shift issue inherent in Behavioral Cloning, Ross and Bagnell (2011) proposed the **Dataset Aggregation (DAgger)** algorithm. The key idea is to iteratively collect data from the learner's policy and let the expert relabel these states with correct actions. This progressively aligns the training distribution with the states actually encountered by the learned policy.

Algorithm. At iteration k, given the current policy π_k, the algorithm performs:

1. Roll out the policy π_k in the environment to obtain a set of states:

$$s_t \sim p_{\pi_k}(s_t),$$

   where p_{π_k} denotes the state distribution induced by π_k.

2. Query the expert π_E for the corresponding actions:

$$a_t = \pi_E(s_t).$$

3. Aggregate these expert-labeled samples into the dataset:

$$\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, a_E(s_t))\}.$$

4. Retrain the policy by minimizing the supervised loss on the aggregated dataset:

$$\pi_{k+1} = \arg\min_{\pi} \ \mathbb{E}_{(s,a)\sim\mathcal{D}}[-\log \pi(a|s)].$$

Error Bound. DAgger reduces the compounding error from quadratic to linear growth in horizon:

$$J(\pi_\theta) - J(\pi_E) = \mathcal{O}(T\epsilon),$$

where ϵ is the imitation error per step. This ensures robustness against distributional drift and guarantees sublinear regret.

Empirical Behavior. In practice:

- Early iterations include many low-reward states, making training temporarily harder.
- Later iterations include more corrective actions, improving recovery and stability.
- This yields the typical *train-down / eval-up* pattern seen in experiments.

Comparison between BC and DAgger. Although both algorithms aim to imitate expert behavior, their learning dynamics and theoretical guarantees differ fundamentally.

State Distribution Dynamics. DAgger' s aggregated dataset approximates:

$$p_{\text{DAgger}}(s) = \frac{1}{K}\sum_{k=1}^{K} p_{\pi_k}(s),$$

thus covering both ideal and off-policy states. This coverage allows the learner to generalize to recovery scenarios unseen by BC.

Empirical Insights. On `Walker2d-v4`:

- Eval_AverageReturn rises from 2300 to 5200 after 10 DAgger iterations.
- Train_AverageReturn first drops (due to difficult samples) then recovers to near-expert level (≈ 5400).

Conclusion. **Behavioral Cloning** provides a simple supervised baseline but suffers from distributional mismatch. **DAgger** explicitly addresses this by querying the expert on states induced by the learner, thereby learning corrective behavior and achieving robust long-horizon performance. In continuous control tasks like Walker2d, DAgger demonstrates a clear advantage, transforming unstable imitation into stable, expert-level control through iterative data aggregation.

Implementation Notes on PyTorch Components

1. `expand_as()` — Tensor Shape Broadcasting. In the implementation of continuous-action Behavioral Cloning, the policy outputs both the mean $\mu_\theta(s)$ and the log standard deviation $\log\sigma_\theta$ of a Gaussian distribution. Since $\log\sigma_\theta$ is stored as a learnable vector of size (action_dim,), we must expand it to match the batch dimension of $\mu_\theta(s)$ before constructing the distribution:

$$\sigma_\theta(s) = \exp(\log\sigma_\theta) \in \mathbb{R}^{d_a}, \qquad \sigma_\theta(s) = \texttt{self.logstd.exp().expand\_as(mean)}$$

The method `expand_as(mean)` replicates the parameter along the batch dimension without allocating additional memory. This ensures that for a batch of N samples, both μ and σ have the same shape (N, d_a), allowing element-wise sampling and log-probability computation.

2. `torch.distributions.Independent()`. In continuous control, we model the policy as a multivariate Gaussian with diagonal covariance:

$$\pi_\theta(a|s) = \mathcal{N}(a; \mu_\theta(s), \mathrm{diag}(\sigma_\theta^2)).$$

To implement this, PyTorch provides:

```
dist = torch.distributions.Independent(
            torch.distributions.Normal(mean, std), 1)
```

The inner `Normal` defines a univariate Gaussian for each action dimension. The outer `Independent(..., 1)` tells PyTorch that the last dimension of the tensor should be treated as an *event dimension*, i.e. all action components are independent but belong to a single joint distribution. This allows the resulting distribution to support:

- `dist.sample()` — draws a full action vector a_t;
- `dist.log_prob(a)` — returns the sum of per-dimension log-probabilities, consistent with $\sum_i \log \mathcal{N}(a_i; \mu_i, \sigma_i^2)$.

Hence, `Independent` is crucial to make $\pi_\theta(a|s)$ a proper multivariate Gaussian, not just a collection of scalars.

3. Log-Standard-Deviation Parameterization. We optimize $\log \sigma$ rather than σ directly to ensure numerical stability and guarantee $\sigma > 0$. The parameter update proceeds as:

$$\sigma = \exp(\log \sigma), \quad \nabla_{\log \sigma} = \sigma \, \nabla_\sigma.$$

This exponential parameterization is standard in policy-gradient and imitation learning implementations, preventing invalid (negative or zero) variances.

4. Replay Buffer Sampling. The replay buffer stores arrays:

$$\mathcal{D} = \{(s_t, a_t, r_t, s_{t+1}, d_t)\}_{t=1}^{M}.$$

For imitation learning, we only need (s_t, a_t) pairs. Sampling a mini-batch proceeds as:

```
indices = np.random.permutation(len(replay_buffer.obs))[:batch_size]
ob_batch = replay_buffer.obs[indices]
ac_batch = replay_buffer.acs[indices]
```

This ensures temporal independence between samples and stabilizes gradient estimates.

5. Distribution-Based Forward Pass. In continuous BC, the forward pass returns a distribution rather than a deterministic tensor:

```
def forward(self, obs):
    mean = self.mean_net(obs)
    std = self.logstd.exp().expand_as(mean)
    return torch.distributions.Independent(
                torch.distributions.Normal(mean, std), 1)
```

At training time, we compute:

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

which can be implemented as:

```
loss = -dist.log_prob(actions).mean()
```

This encourages the policy to maximize the likelihood of expert actions under its own Gaussian distribution.

6. Interpretation of Log-Probability Summation. When computing `dist.log_prob(a)`, PyTorch automatically sums over action dimensions. This is equivalent to:

$$\log \pi_\theta(a|s) = \sum_{i=1}^{d_a} \log \mathcal{N}(a_i; \mu_i, \sigma_i^2),$$

ensuring the policy likelihood corresponds to the full action vector. This summation aligns with the continuous analog of the categorical cross-entropy used in discrete BC.