

A Variable ranges

Table 1 presents a comprehensive list of physiological and laboratory variables used in the experiments. It includes their respective measurement ranges, units, and corresponding Logical Observation Identifiers Names and Codes (LOINC). The table covers vital signs (e.g., blood pressure, heart rate), respiratory parameters (e.g., tidal volume, inspiratory airway pressure), blood gas values (e.g., pH, PaO₂, PaCO₂), hematological markers (e.g., hemoglobin, WBC count), and fluid balance indicators (e.g., intravenous fluid intake, urine output). Additionally, demographic variables such as age, sex, weight, and height are included. The ranges reflect observed values within the dataset, ensuring consistency in the experimental setup

Table 1: Variable ranges, units and LOINC codes used for the experiments

Variable	Ranges	Unit	LOINC
Mean Arterial Pressure	30-200	mmHg	8478-0
Diastolic Pressure	20-120	mmHg	8500-1
Systolic Pressure	50-260	mmHg	8480-6
Inspiratory Airway Pressure	10-60	10-60	75942-3
Tidal Volume (observed)	80-2040	ml	75958-9
Haemoglobin	2-20	mmol/l	718-7
White Blood Cell Count	0-30	Gpt/l	26464-8
SaO ₂	40-100	%	2708-6
SpO ₂	30-100	%	59408-5
PaO ₂	20-600	mmHg	19255-9
PaCO ₂	20-100	mmHg	32771-8
Base excess	-20-30	mmol/l	11555-0
pH	6-8	-	97536-7
Intravenous Fluid Intake	0-20000	ml/4h	(multiple)
Urine Output	0-2000	ml/4h	(multiple)
Vasopressors	0-5	NE	(multiple)
Potassium	2-10	mmol/l	75940-7
Chloride	80-150	mmol/l	2069-3
Sodium	120-180	mmol/l	2947-0
INR	0.9-15	-	34714-6
Heart Rate	20-200	1/min	8889-8
Age	18-120	years	30525-0
Sex	0-1	-	72143-1
Weight	40-140	kg	29463-7
Height	155-200	cm	8302-2

B Dataset statistics

Table 2 provides an overview of the datasets used in the experiments, including MIMIC-IV, eICU, and HiRID. The table presents key statistics for each dataset, including the number of patients, the number of ventilation episodes, and the total hours of mechanical ventilation (MV) recorded.

Table 2: Statistics from the individual state vectors from each database after pre-processing before combination

Dataset	No. Patients	No. Episodes	Hours of MV
MIMIC IV	1,538	1,616	154,296
eICU	5,678	5,804	678,740
HiRID	5,356	5,531	419,469

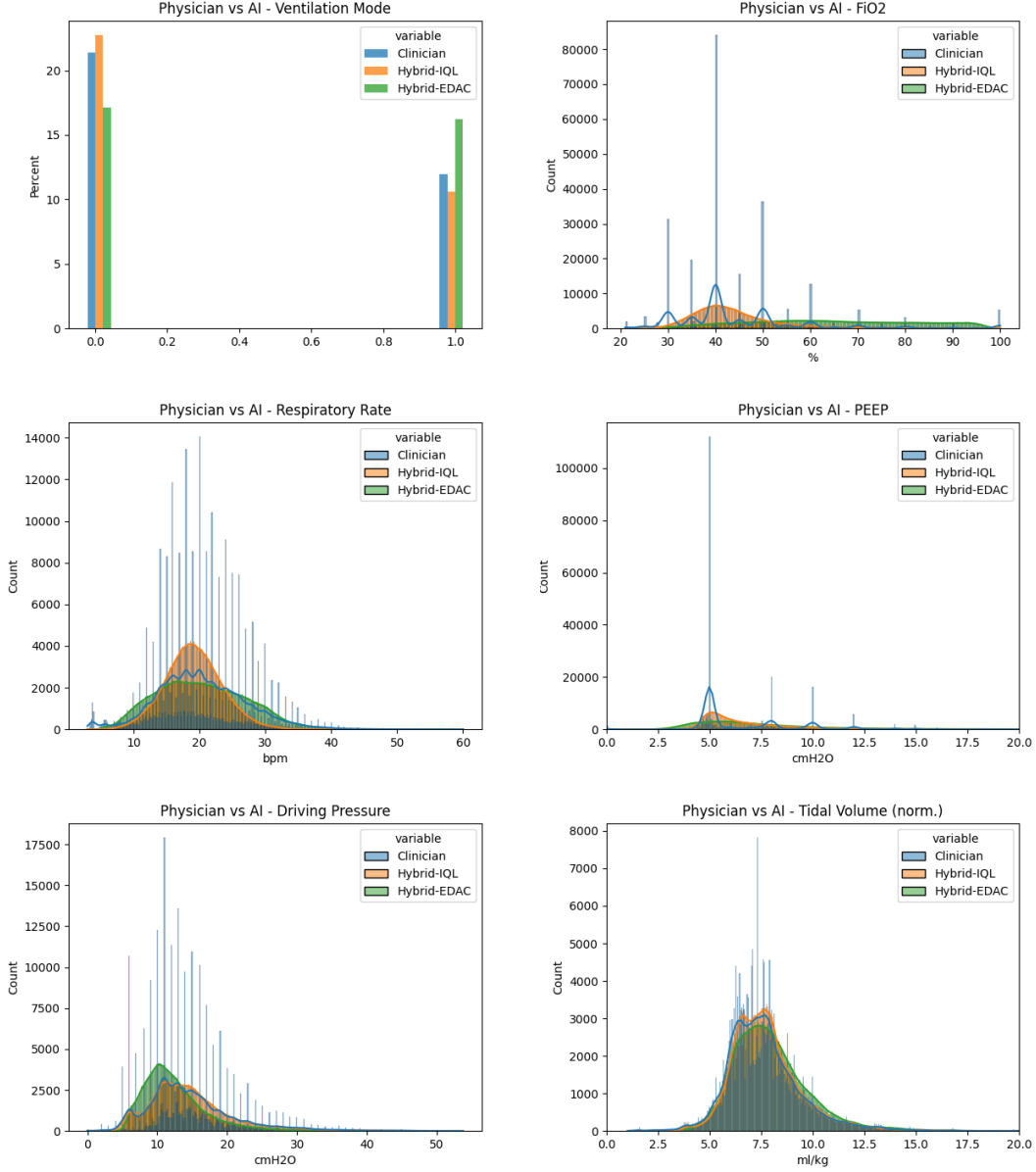
C Action bins

The bins in Table 4 have been defined by clinicians for action discretization based on clinical experience and practical use. Bin thresholds were defined as left-inclusive. For the ventilation mode dependent actions Driving Pressure and PEEP, the last bin it is not available in the given mode.

Table 3: Bin indices and their ranges for action discretization

Action	1	2	3	4	5	6	7	8	9
Ventilation control mode	0	1							
Respiratory rate	5-10	10-15	15-20	20-25	25-30	30-35	35-60		
Tidal Volume	3-4	4-5	5-6	6-7	7-8	8-9	9-10	10-11	11-12
Driving Pressure	0-6	6-10	10-14	14-18	18-22	22-26	26-40	n.a.	
PEEP	0-4	4-8	8-12	12-16	16-20	20-50	n.a.		
FiO2	21-40	40-60	60-80	80-100					

D Action distributions

**Figure 1:** Action distribution of normalized tidal volume for trained and dataset policies

Distribution comparison of different actions for clinician and the trained policies (using HybridEDAC & HybridIQL) shown in Figure 1.

E Action d^π

Table 4 shows d^π for trained action distribution compared to clinician distribution.

Table 4: d^π for trained action distribution compared to clinician distribution.

Algorithm	Tidal Volume	Respiratory Rate	PEEP	FiO ₂	Driving Pressure
EDAC	-167.77	2.49	-8.76	3.74	-4.99
IQL	-11.77	4.37	3.00	4.27	3.74
CQL	-70.74	0.75	-29.89	4.27	-18.04

F Code appendix

Code snippets are provided for review exclusively and full code is planned to released in 2026 due to the project restrictions. For both IQL and EDAC the code from CORL repo¹ was used as base. We only show the core updated parts of the code.

F.1 HybridIQL

IQL Policy Forward

```

1 LOG_STD_MIN = -20.0
2 LOG_STD_MAX = 2.0
3
4
5 def forward(self, obs: torch.Tensor):
6     base_out = self.policy_base(obs)
7
8     # Get continuous action distribution
9     mean = self.cont_mean_out(base_out)
10    std = torch.exp(self.log_std.clamp(LOG_STD_MIN, LOG_STD_MAX))
11    continuous_action_dist = Normal(mean, std)
12
13    # Get discrete actions distribution
14    logits = self.discrete_out(base_out)
15
16    discrete_action_dist = OneHotCategorical(logits=logits)
17    return continuous_action_dist, discrete_action_dist

```

IQL Policy Update

```

1 EXP_ADV_MAX = 100.0
2
3 def _update_policy(
4     self,
5     adv: torch.Tensor,
6     observations: torch.Tensor,
7     discrete_actions: torch.Tensor,
8     continuous_actions: torch.Tensor,
9     log_dict: Dict,
10 ):
11     exp_adv = torch.exp(self.beta * adv.detach()).clamp(max=EXP_ADV_MAX)
12     cont_dist, disc_dist = self.actor(observations)
13
14     log_prob_cont = cont_dist.log_prob(continuous_actions)
15     log_prob_disc = disc_dist.log_prob(discrete_actions)
16     log_prob_disc = log_prob_disc.unsqueeze(-1) if log_prob_disc.dim() == 1 else
17     ↪ log_prob_disc
18
19     log_prob = torch.cat([log_prob_cont, log_prob_disc], dim=-1)
20     bc_losses = -(log_prob.sum(-1, keepdim=False))
21
22     policy_loss = torch.mean(exp_adv * bc_losses)
23     log_dict["actor_loss"] = policy_loss.item()
24     self.actor_optimizer.zero_grad()
25     policy_loss.backward()
26     self.actor_optimizer.step()
27     self.actor_lr_schedule.step()

```

¹ <https://github.com/tinkoff-ai/CORL>

F.2 HybridEDAC

We modified the SOFT Actor Critic using the implementation from repo². Except the critic for the EDAC accepts both discrete and continuous actions on the input side.

EDAC Policy Forward

```
1  def forward(  
2      self,  
3      state: torch.Tensor,  
4      deterministic: bool = False,  
5  ):  
6      hidden = self.trunk(state)  
7      mu, log_sigma = self.mu(hidden), self.log_sigma(hidden)  
8      # Clip log_sigma as between -3 and 1  
9      log_sigma = torch.clip(log_sigma, self.log_std_min, self.log_std_max)  
10     cont_policy_dist = Normal(mu, torch.exp(log_sigma))  
11  
12     # Compute logits for the discrete branch  
13     logits = self.discrete_out(hidden)  
14     # Also compute the softmax probabilities (used for log-probs and alpha updates)  
15     prob_disc = softmax(logits, dim=-1)  
16     log_prob_disc = torch.log(prob_disc + 1e-8)  
17  
18  
19     if deterministic:  
20         cont_action = cont_policy_dist.mean  
21         disc_action = OneHotCategorical(logits=logits).mode  
22     else:  
23         cont_action = cont_policy_dist.rsample()  
24         disc_action = OneHotCategorical(logits=logits).sample()  
25  
26     # Apply tanh squashing to continuous actions  
27     tanh_action = torch.tanh(cont_action)  
28  
29     # Compute log probability for continuous actions with tanh correction.  
30     all_log_prob_c = cont_policy_dist.log_prob(cont_action)  
31     all_log_prob_c -= torch.log(1.0 - tanh_action.pow(2) + 1e-6)  
32     log_prob_cont = all_log_prob_c.sum(dim=-1, keepdim=True)  
33  
34  
35     actions = TensorDict(  
36         {  
37             'discrete_actions': disc_action,  
38             'continuous_actions': tanh_action  
39         },  
40         batch_size=state.shape[0]  
41     )  
42  
43     return actions, log_prob_cont, log_prob_disc, prob_disc
```

Alpha Loss Function

```
1  def _alpha_loss(self, state: torch.Tensor):  
2      with torch.no_grad():  
3          _, action_log_prob_cont, action_log_prob_disc, prob_disc = self.actor(state)  
4  
5          loss_cont = (-self.log_alpha_cont * prob_disc * (  
6              prob_disc * action_log_prob_cont + self.target_entropy_cont)).sum(-1).mean()  
7          loss_disc = (-self.log_alpha_disc * prob_disc * (action_log_prob_disc +  
8              ↪ self.target_entropy_disc)).sum(-1).mean()  
9          return loss_cont, loss_disc
```

² <https://github.com/nisheeth-golakiya/hybrid-sac>

```

1                                     Actor Loss Function
2
3  def _actor_loss(self, state: torch.Tensor):
4      action, action_log_prob_cont, action_log_prob_disc, prob_d = self.actor(state)
5      cont_action = action['continuous_actions']
6      disc_action = action['discrete_actions']
7      q_value_dist = self.critic(state, cont_action, disc_action)
8      q_value_min = q_value_dist.min(0).values.unsqueeze(-1)
9
10     # Note: here the loss is weighted by the discrete probability.
11     loss_disc = (prob_d * (self.alpha_disc * action_log_prob_disc -
12     ↪ q_value_min)).sum(-1).mean()
13     loss_cont = (prob_d * (self.alpha_cont * prob_d * action_log_prob_cont -
14     ↪ q_value_min)).sum(-1).mean()
15     loss = loss_cont + loss_disc
16     return loss

```

```

1                                     Critic loss
2
3  def _critic_loss(
4      self,
5      state: torch.Tensor,
6      discrete_action: torch.Tensor,
7      cont_action: torch.Tensor,
8      reward: torch.Tensor,
9      next_state: torch.Tensor,
10     done: torch.Tensor,
11 ) -> torch.Tensor:
12     with torch.no_grad():
13         next_action, next_action_log_prob_cont, next_action_log_prob_disc,
14         ↪ next_action_prob_disc = self.actor(
15             next_state)
16         next_action_cont = next_action['continuous_actions']
17         next_action_disc = next_action['discrete_actions']
18         q_next = self.target_critic(next_state, next_action_cont,
19         ↪ next_action_disc).min(0).values.unsqueeze(-1)
20         q_next = next_action_prob_disc * (
21             q_next - self.alpha_cont * next_action_prob_disc *
22             ↪ next_action_log_prob_cont - self.alpha_disc *
23             ↪ next_action_log_prob_disc)
24
25         v_next = q_next.sum(-1).unsqueeze(-1)
26         q_target = reward + self.gamma * (1 - done) * v_next
27
28         q_values = self.critic(state, cont_action, discrete_action)
29
30         critic_loss = ((q_values - q_target.view(1, -1)) ** 2).mean(dim=1).sum(dim=0)
31
32         diversity_loss = self._critic_diversity_loss(
33             state=state,
34             discrete_action=discrete_action,
35             cont_action=cont_action
36         )
37
38         loss = critic_loss + self.eta * diversity_loss
39         return loss

```

F.3 FactoredCQL

`index_to_hot_factored_action` is unique one hot encoded set of possible actions in factored form. Which is obtained by applying unique to discretized dataset actions.

```

1                                     Hot Encoded Unique Actions
2
3  def get_index_to_hot_factored_action(dataset_actions, list_of_actions,
4      ↪ bins_per_action_dimension):
5      dataset_actions = tensor(dataset_actions)
6      dataset_actions = torch.unique(dataset_actions, dim=0)

```

```

4     encoded_action_dimensions = []
5
6     for i, action in enumerate(list_of_actions):
7         encoded_action_dimensions.append(
8             one_hot(dataset_actions[:, i].squeeze(),
9                     ↪ num_classes=bins_per_action_dimension[i]).reshape(-1,
10                     ↪ bins_per_action_dimension[i])
11         )
12     return torch.cat(encoded_action_dimensions, dim=1).to(dtype=float32)

```

Critic loss

```

1  def get_specific_action_q_value(q_value, actions):
2      values = q_value * actions
3      value = values.sum(dim=1).unsqueeze(-1)
4      return value
5
6
7  def get_q_values(q_factored, index_to_hot_factored_action):
8      return q_factored @ index_to_hot_factored_action.T
9
10
11 def learn(self, buffer: ReplayBuffer, **kwargs):
12     self.critic_1.train()
13     self.critic_2.train()
14     batch = buffer.sample(batch_size=self.batch_size)
15     states = batch.observations
16     actions = batch.actions
17     next_states = batch.next_observations
18     rewards = batch.rewards
19     dones = batch.terminals
20
21     with torch.no_grad():
22         target_q_1_factored = self.critic_1(next_states)
23         target_q_2_factored = self.critic_2(next_states)
24         target_q_1 = get_q_values(target_q_1_factored,
25             ↪ self.index_to_hot_encoded_factorized_action)
26         target_q_2 = get_q_values(target_q_2_factored,
27             ↪ self.index_to_hot_encoded_factorized_action)
28
29         target_q_values = torch.max(
30             torch.min(
31                 target_q_1,
32                 target_q_2
33             ),
34             dim=1
35         ).values
36
37         target_q_values = target_q_values.unsqueeze(1)
38         td_target = rewards + (self.gamma * target_q_values * (1 - dones))
39
40     q1_values_factored = self.critic_1(states)
41     q1_values = get_q_values(q1_values_factored,
42         ↪ self.index_to_hot_encoded_factorized_action)
43     q1_predicted = get_specific_action_q_value(q1_values_factored, actions)
44
45     q2_values_factored = self.critic_2(states)
46     q2_values = get_q_values(q2_values_factored,
47         ↪ self.index_to_hot_encoded_factorized_action)
48     q2_predicted = get_specific_action_q_value(q2_values_factored, actions)
49
50     conservative_loss_1 = (torch.logsumexp(q1_values, dim=1, keepdim=True) -
51         ↪ q1_predicted).mean()
52     conservative_loss_2 = (torch.logsumexp(q2_values, dim=1, keepdim=True) -
53         ↪ q2_predicted).mean()

```

```

49     conservative_loss_1 = self.cql_alpha * conservative_loss_1
50     conservative_loss_2 = self.cql_alpha * conservative_loss_2
51
52     critic_1_loss = F.mse_loss(q1_predicted, td_target)
53     critic_2_loss = F.mse_loss(q2_predicted, td_target)
54
55     cql_loss = conservative_loss_1 + conservative_loss_2 + 0.5 * (critic_1_loss +
56     ↪ critic_2_loss)
57
58     self.optimizer.zero_grad()
59     cql_loss.backward()
60     clip_grad_norm_(self.critic_1.parameters(), self.clip_grad_max_norm)
61     clip_grad_norm_(self.critic_2.parameters(), self.clip_grad_max_norm)
62     self.optimizer.step()
63
64     self.target_update()

```

F.4 Autoencoder

```

_____ Autoencoder Forward _____
1  def forward(self, state, cont_action, disc_action):
2      state_action = torch.cat((state, cont_action, disc_action), dim=-1)
3      ae_trunk_out = self.ae_trunk(state_action)
4      state_mean = self.state_mean(ae_trunk_out)
5      state_log_var = self.state_log_var(ae_trunk_out)
6      clamped_state_log_var = torch.clip(state_log_var, self.log_var_min, self.log_var_max)
7      state_dist = torch.distributions.Normal(state_mean, torch.exp(clamped_state_log_var)
8      ↪ + 1e-6)
9      cont_action_mean = self.cont_action_mean(ae_trunk_out)
10     cont_action_log_var = self.cont_action_log_var(ae_trunk_out)
11     clamped_cont_action_log_var = torch.clip(cont_action_log_var, self.log_var_min,
12     ↪ self.log_var_max)
13     cont_action_dist = torch.distributions.Normal(cont_action_mean,
14     ↪ torch.exp(clamped_cont_action_log_var) + 1e-6)
15
16     disc_action_dist =
17     ↪ torch.distributions.OneHotCategorical(logits=self.disc_action_out(ae_trunk_out))
18
19     return state_dist, cont_action_dist, disc_action_dist

```

```

_____ Autoencoder Training Step _____
1  def learn(self, buffer: ReplayBuffer, **kwargs) -> Dict[str, float]:
2      self.model.train()
3      batch_size = self.batch_size
4      batch = buffer.sample(batch_size)
5      log_dict = {}
6
7      state = batch.observations
8      action = batch.actions
9      continuous_action = action['continuous_actions']
10     discrete_action = action['discrete_actions']
11
12     state_dist, cont_action_dist, disc_action_dist = self.model(
13         state=state,
14         cont_action=continuous_action,
15         disc_action=discrete_action
16     )
17
18     state_loss = -state_dist.log_prob(state).mean()
19     cont_action_loss = -cont_action_dist.log_prob(continuous_action).mean()
20     disc_action_loss = -disc_action_dist.log_prob(discrete_action).mean()
21
22     loss = state_loss + cont_action_loss + disc_action_loss

```

23
24
25
26
27
28

```
self.optimizer.zero_grad()  
loss.backward()  
self.optimizer.step()
```
