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
SaO2	40-100	%	2708-6
SpO2	30-100	%	59408-5
PaO2	20-600	mmHg	19255-9
PaCO2	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

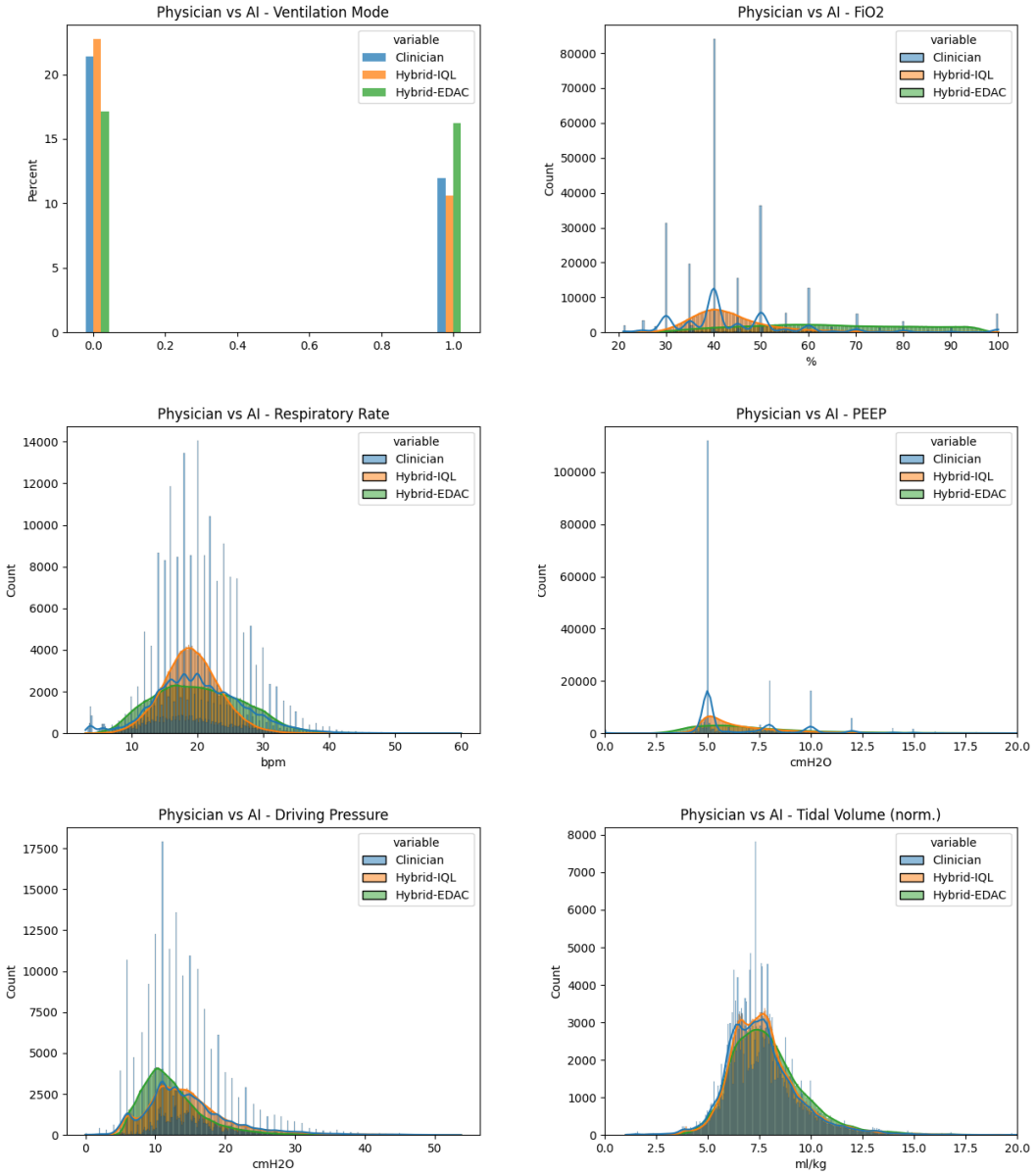

Figure 1: Action distribution of normalized tidal volume for trained and dataset policies

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 repo [1] was used as base. We only show the core updated parts of the code.

F.1 HybridIQL

IQL Policy Forward

```
LOG_STD_MIN = -20.0
LOG_STD_MAX = 2.0

    def forward(self, obs: torch.Tensor):
        base_out = self.policy_base(obs)

        # Get continuous action distribution
        mean = self.cont_mean_out(base_out)
        std = torch.exp(self.log_std.clamp(LOG_STD_MIN, LOG_STD_MAX))
        continuous_action_dist = Normal(mean, std)

        # Get discrete actions distribution
        logits = self.discrete_out(base_out)

        discrete_action_dist = OneHotCategorical(logits=logits)
        return continuous_action_dist, discrete_action_dist
```

IQL Policy Update

```
EXP_ADV_MAX = 100.0

        def _update_policy(
            self,
            adv: torch.Tensor,
            observations: torch.Tensor,
            discrete_actions: torch.Tensor,
            continuous_actions: torch.Tensor,
            log_dict: Dict,
    ):
        exp_adv = torch.exp(self.beta * adv.detach()).clamp(max=EXP_ADV_MAX)
        cont_dist, disc_dist = self.actor(observations)

        log_prob_cont = cont_dist.log_prob(continuous_actions)
        log_prob_disc = disc_dist.log_prob(discrete_actions)
        log_prob_disc = log_prob_disc.unsqueeze(-1) if log_prob_disc.dim() == 1 else
        ↪  log_prob_disc

        log_prob = torch.cat([log_prob_cont, log_prob_disc], dim=-1)
        bc_losses = -(log_prob.sum(-1, keepdim=False))

        policy_loss = torch.mean(exp_adv * bc_losses)
        log_dict["actor_loss"] = policy_loss.item()
        self.actor_optimizer.zero_grad()
        policy_loss.backward()
        self.actor_optimizer.step()
        self.actor_lr_schedule.step()
```

[1] https://github.com/tinkoff-ai/CORL

F.2 HybridEDAC

We modifed the SOFT Actor Critic using the implementation from repo[2]. Except the critic for the EDAC accepts both discrete and continuous actions on the input side.

EDAC Policy Forward

```python
    def forward(
            self,
            state: torch.Tensor,
            deterministic: bool = False,
    ):
        hidden = self.trunk(state)
        mu, log_sigma = self.mu(hidden), self.log_sigma(hidden)
        # Clip log_sigma as between -3 and 1
        log_sigma = torch.clip(log_sigma, self.log_std_min, self.log_std_max)
        cont_policy_dist = Normal(mu, torch.exp(log_sigma))

        # Compute logits for the discrete branch
        logits = self.discrete_out(hidden)
        # Also compute the softmax probabilities (used for log-probs and alpha updates)
        prob_disc = softmax(logits, dim=-1)
        log_prob_disc = torch.log(prob_disc + 1e-8)


        if deterministic:
            cont_action = cont_policy_dist.mean
            disc_action = OneHotCategorical(logits=logits).mode
        else:
            cont_action = cont_policy_dist.rsample()
            disc_action = OneHotCategorical(logits=logits).sample()

        # Apply tanh squashing to continuous actions
        tanh_action = torch.tanh(cont_action)

        # Compute log probability for continuous actions with tanh correction.
        all_log_prob_c = cont_policy_dist.log_prob(cont_action)
        all_log_prob_c -= torch.log(1.0 - tanh_action.pow(2) + 1e-6)
        log_prob_cont = all_log_prob_c.sum(dim=-1, keepdim=True)


        actions = TensorDict(
            {
                'discrete_actions': disc_action,
                'continuous_actions': tanh_action
            },
            batch_size=state.shape[0]
        )

        return actions, log_prob_cont, log_prob_disc, prob_disc
```

Alpha Loss Function

```python
    def _alpha_loss(self, state: torch.Tensor):
        with torch.no_grad():
            _, action_log_prob_cont, action_log_prob_disc, prob_disc = self.actor(state)

        loss_cont = (-self.log_alpha_cont * prob_disc * (
                prob_disc * action_log_prob_cont + self.target_entropy_cont)).sum(-1).mean()
        loss_disc = (-self.log_alpha_disc * prob_disc * (action_log_prob_disc +
        ↪  self.target_entropy_disc)).sum(
            -1).mean()
        return loss_cont, loss_disc
```

[2] https://github.com/nisheeth-golakiya/hybrid-sac

Actor Loss Function

```
    def _actor_loss(self, state: torch.Tensor):
        action, action_log_prob_cont, action_log_prob_disc, prob_d = self.actor(state)
        cont_action = action['continuous_actions']
        disc_action = action['discrete_actions']
        q_value_dist = self.critic(state, cont_action, disc_action)
        q_value_min = q_value_dist.min(0).values.unsqueeze(-1)

        # Note: here the loss is weighted by the discrete probability.
        loss_disc = (prob_d * (self.alpha_disc * action_log_prob_disc -
        ↪  q_value_min)).sum(-1).mean()
        loss_cont = (prob_d * (self.alpha_cont * prob_d * action_log_prob_cont -
        ↪  q_value_min)).sum(-1).mean()
        loss = loss_cont + loss_disc
        return loss
```

Critic loss

```
    def _critic_loss(
            self,
            state: torch.Tensor,
            discrete_action: torch.Tensor,
            cont_action: torch.Tensor,
            reward: torch.Tensor,
            next_state: torch.Tensor,
            done: torch.Tensor,
    ) -> torch.Tensor:
        with torch.no_grad():
            next_action, next_action_log_prob_cont, next_action_log_prob_disc,
            ↪  next_action_prob_disc = self.actor(
                next_state)
            next_action_cont = next_action['continuous_actions']
            next_action_disc = next_action['discrete_actions']
            q_next = self.target_critic(next_state, next_action_cont,
            ↪  next_action_disc).min(0).values.unsqueeze(-1)
            q_next = next_action_prob_disc * (
                    q_next - self.alpha_cont * next_action_prob_disc *
                    ↪  next_action_log_prob_cont - self.alpha_disc *
                    ↪  next_action_log_prob_disc)

            v_next = q_next.sum(-1).unsqueeze(-1)
            q_target = reward + self.gamma * (1 - done) * v_next

        q_values = self.critic(state, cont_action, discrete_action)

        critic_loss = ((q_values - q_target.view(1, -1)) ** 2).mean(dim=1).sum(dim=0)

        diversity_loss = self._critic_diversity_loss(
            state=state,
            discrete_action=discrete_action,
            cont_action=cont_action
        )

        loss = critic_loss + self.eta * diversity_loss
        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.

Hot Encoded Unique Actions

```
    def get_index_to_hot_factored_action(dataset_actions, list_of_actions,
    ↪  bins_per_action_dimension):
        dataset_actions = tensor(dataset_actions)
        dataset_actions = torch.unique(dataset_actions, dim=0)
```

```
        encoded_action_dimensions = []

        for i, action in enumerate(list_of_actions):
            encoded_action_dimensions.append(
                one_hot(dataset_actions[:, i].squeeze(),
                ↪  num_classes=bins_per_action_dimension[i]).reshape(-1,
                ↪  bins_per_action_dimension[i])
            )
    return torch.cat(encoded_action_dimensions, dim=1).to(dtype=float32)
```

Critic loss

```
    def get_specific_action_q_value(q_value, actions):
        values = q_value * actions
        value = values.sum(dim=1).unsqueeze(-1)
        return value


    def get_q_values(q_factored, index_to_hot_factored_action):
        return q_factored @ index_to_hot_factored_action.T


    def learn(self, buffer: ReplayBuffer, **kwargs):
        self.critic_1.train()
        self.critic_2.train()
        batch = buffer.sample(batch_size=self.batch_size)
        states = batch.observations
        actions = batch.actions
        next_states = batch.next_observations
        rewards = batch.rewards
        dones = batch.terminals

        with torch.no_grad():
            target_q_1_factored = self.critic_1(next_states)
            target_q_2_factored = self.critic_2(next_states)
            target_q_1 = get_q_values(target_q_1_factored,
            ↪  self.index_to_hot_encoded_factorized_action)
            target_q_2 = get_q_values(target_q_2_factored,
            ↪  self.index_to_hot_encoded_factorized_action)

            target_q_values = torch.max(
                torch.min(
                    target_q_1,
                    target_q_2
                ),
                dim=1
            ).values

            target_q_values = target_q_values.unsqueeze(1)
            td_target = rewards + (self.gamma * target_q_values * (1 - dones))

        q1_values_factored = self.critic_1(states)
        q1_values = get_q_values(q1_values_factored,
        ↪  self.index_to_hot_encoded_factorized_action)
        q1_predicted = get_specific_action_q_value(q1_values_factored, actions)

        q2_values_factored = self.critic_2(states)
        q2_values = get_q_values(q2_values_factored,
        ↪  self.index_to_hot_encoded_factorized_action)
        q2_predicted = get_specific_action_q_value(q2_values_factored, actions)

        conservative_loss_1 = (torch.logsumexp(q1_values, dim=1, keepdim=True) -
        ↪  q1_predicted).mean()
        conservative_loss_2 = (torch.logsumexp(q2_values, dim=1, keepdim=True) -
        ↪  q2_predicted).mean()
```

```
        conservative_loss_1 = self.cql_alpha * conservative_loss_1
        conservative_loss_2 = self.cql_alpha * conservative_loss_2

        critic_1_loss = F.mse_loss(q1_predicted, td_target)
        critic_2_loss = F.mse_loss(q2_predicted, td_target)

        cql_loss = conservative_loss_1 + conservative_loss_2 + 0.5 * (critic_1_loss +
        ↪  critic_2_loss)

        self.optimizer.zero_grad()
        cql_loss.backward()
        clip_grad_norm_(self.critic_1.parameters(), self.clip_grad_max_norm)
        clip_grad_norm_(self.critic_2.parameters(), self.clip_grad_max_norm)
        self.optimizer.step()

        self.target_update()

```

F.4 Autoencoder

Autoencoder Forward

```
    def forward(self, state, cont_action, disc_action):
        state_action = torch.cat((state, cont_action, disc_action), dim=-1)
        ae_trunk_out = self.ae_trunk(state_action)
        state_mean = self.state_mean(ae_trunk_out)
        state_log_var = self.state_log_var(ae_trunk_out)
        clamped_state_log_var = torch.clip(state_log_var, self.log_var_min, self.log_var_max)
        state_dist = torch.distributions.Normal(state_mean, torch.exp(clamped_state_log_var)
        ↪  + 1e-6)
        cont_action_mean = self.cont_action_mean(ae_trunk_out)
        cont_action_log_var = self.cont_action_log_var(ae_trunk_out)
        clamped_cont_action_log_var = torch.clip(cont_action_log_var, self.log_var_min,
        ↪  self.log_var_max)
        cont_action_dist = torch.distributions.Normal(cont_action_mean,
        ↪  torch.exp(clamped_cont_action_log_var) + 1e-6)

        disc_action_dist =
        ↪  torch.distributions.OneHotCategorical(logits=self.disc_action_out(ae_trunk_out))

        return state_dist, cont_action_dist, disc_action_dist
```

Autoencoder Training Step

```
    def learn(self, buffer: ReplayBuffer, **kwargs) -> Dict[str, float]:
        self.model.train()
        batch_size = self.batch_size
        batch = buffer.sample(batch_size)
        log_dict = {}

        state = batch.observations
        action = batch.actions
        continuous_action = action['continuous_actions']
        discrete_action = action['discrete_actions']

        state_dist, cont_action_dist, disc_action_dist = self.model(
            state=state,
            cont_action=continuous_action,
            disc_action=discrete_action
        )

        state_loss = -state_dist.log_prob(state).mean()
        cont_action_loss = -cont_action_dist.log_prob(continuous_action).mean()
        disc_action_loss = -disc_action_dist.log_prob(discrete_action).mean()

        loss = state_loss + cont_action_loss + disc_action_loss
```

```
23
24
25          self.optimizer.zero_grad()
26          loss.backward()
27          self.optimizer.step()
28
```