# Tanmay Joshi-SAIDL Report RL Bonus

Tanmay Joshi

March 2025

## 1 Using Supervised Learning Technique for learning State Action Implementation-

***Problem Statement***: Use any self-supervised learning technique (BYOL, SALE, etc.) to learn state-action representations and integrate it with your previous implementation.
***Sub-topics***: (a) Pick a suitable approach to learn the representations (concatenating them, using shared parameters, state-dependent action representation, etc.) and justify your choice. (b) Demonstrate the effectiveness of your learned representations in relevant ways.
***Paper Used***-https://arxiv.org/pdf/2306.02451
***Background***- The paper ***"For SALE: State-Action Representation Learning for Deep Reinforcement Learning "*** proposes a ***state-action learned embeddings (SALE)***, a method that learns embeddings jointly over both state and action by modeling the dynamics of the environment in latent space . The paper also observes the surprising effect of extrapolation error when significantly expanding the action-dependent input and introduce a simple clipping technique to mitigate it. The paper also explores ***the usage of checkpoints in RL***. Similar to representation learning, early stopping and checkpoints are standard techniques used to enhance the performance of deep learning models. A similar effect can be achieved in RL by fixing each policy for multiple training episodes, and then at test time, using the highest-performing policy observed during training. Similar to TD3 the ***paper proposes a new algorithm TD7*** useful in both offline and online RL settings.
***State Action Representation Learning-*** SALE is a representation learning technique that learns embeddings to enhance reinforcement learning performance. It employs two encoders, $f$ and $g$, to encode the state $s$ and state-action pair $(s, a)$ as:

$$z_s := f(s), \quad z_{sa} := g(z_s, a). \tag{1}$$

These embeddings are used to augment the input to the value function $Q$ and policy $\pi$ networks:

$$Q(s, a) \to Q(z_{sa}, z_s, s, a), \quad \pi(s) \to \pi(z_s, s). \tag{2}$$

The encoders are trained with a mean squared error (MSE) loss to predict the next state embedding:

$$\mathcal{L}(f, g) := \left\| g(f(s), a) - |f(s')|_\times \right\|^2 = \left\| z_{sa} - |z_{s'}|_\times \right\|^2, \tag{3}$$

where $|\cdot|_\times$ denotes the stop-gradient operation. The training of $f$ and $g$ is decoupled from the value function and policy to maintain stability.

**Normalization.** To prevent instability due to scale collapse or growth in the embedding space, SALE uses a custom normalization layer called *AvgL1Norm*, defined as:

$$AvgL1Norm(x) := \frac{x}{\frac{1}{N}\sum_{i=1}^{N} |x_i|}, \tag{4}$$

where $x_i$ is the $i$-th component of the vector $x$. This normalization preserves the relative scale without needing running statistics (unlike BatchNorm), and is applied to $z_s$ and the linear-transformed inputs to $Q$ and $\pi$, but not to $z_{sa}$, which is matched to a normalized target.

**Network Inputs.** The complete inputs to the value and policy networks are:

$$Q(z_{sa}, z_s, AvgL1Norm(Linear(s, a))), \quad \pi(z_s, AvgL1Norm(Linear(s))). \tag{5}$$

The linear layers are trained end-to-end and considered part of the value and policy networks.

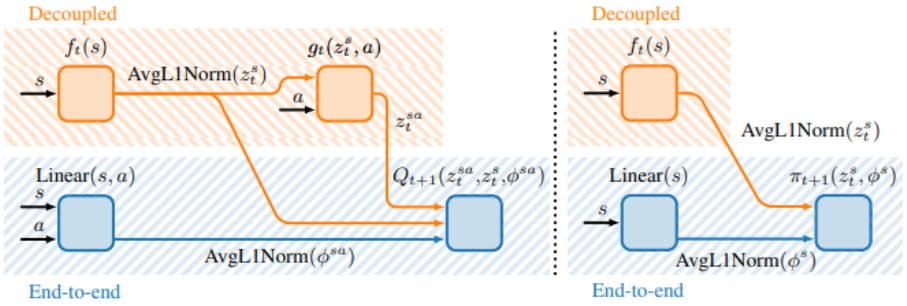
**Fixed Embeddings and Target Networks.** To avoid inconsistencies across updates, SALE freezes embeddings during each policy/value update. At iteration $t+1$, the networks use embeddings from the previous encoders $(f_t, g_t)$:

$Q_{t+1}(z_{sa}^{(t)}, z_s^{(t)}, s, a) \approx r + \gamma Q_t(z_{sa}^{\prime(t-1)}, z_s^{\prime(t-1)}, s', a'), \quad where a' \sim \pi_t(z_s^{\prime(t-1)}, s')$,
$\pi_{t+1}(z_s^{(t)}, s) \approx \arg\max_\pi Q_{t+1}(z_{sa}^{(t)}, z_s^{(t)}, s, a), \quad where a \sim \pi(z_s^{(t)}, s)$.

The current value function $Q_{t+1}$ is trained against the previous $Q_t$ (target network), while $f_{t+1}, g_{t+1}$ are updated with the loss in Equation (3), using $z_s^{\prime(t+1)}$ as the target embedding (without a target network). Every $n$ steps, all target networks are synchronized:

$$Q_t \leftarrow Q_{t+1}, \quad \pi_t \leftarrow \pi_{t+1}, \quad (f_{t-1}, g_{t-1}) \leftarrow (f_t, g_t), \quad (f_t, g_t) \leftarrow (f_{t+1}, g_{t+1}). \tag{6}$$

**Extrapolation Error-** Extrapolation error refers to the tendency of deep value functions to assign unrealistic values to rarely seen state-action pairs. This issue is particularly problematic in offline RL due to the lack of feedback. However, we observe a similar phenomenon in online RL, especially when increasing the dimensionality of the state-action input to the value function.




(a) Value function $Q$ with SALE (b) Policy $\pi$ with SALE

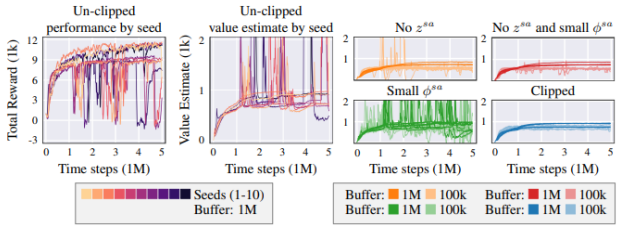
As shown in Figure below (Ant environment), increasing the size of the state-action embedding $z_{sa}$ or the output of the linear layer $\phi_{sa} := Linear(s, a)$ can cause instability in value estimates. Performance may exhibit large dips corresponding to sudden spikes in value predictions. Ablations reveal that these effects stem primarily from $z_{sa}$ and $\phi_{sa}$, while the state embedding $z_s$ remains unaffected.

**Mitigation via Clipping** - In online RL, such overestimations are eventually corrected through interaction with the environment. Thus, stabilizing value estimates until correction suffices. SALE addresses this by clipping the target value during training, using the observed value range from dataset $\mathcal{D}$, estimated via mini-batches. The updated value function target becomes:

$$Q_{t+1}(s, a) \approx r + \gamma \cdot clip\left(Q_t(s', a'), \min_{(s,a)\in\mathcal{D}} Q_t(s, a), \max_{(s,a)\in\mathcal{D}} Q_t(s, a)\right), \quad (7)$$

where $a' \sim \pi_t(z'_s, s')$. This value clipping helps stabilize learning without altering the input dimension.

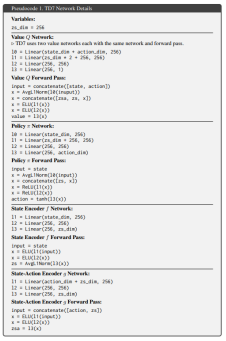
Figure for above explanation-




**Stabilizing RL with Decoupled Representation Learning** : A checkpoint is a snapshot of the parameters of a model, captured at a specific time during

training. In supervised learning, checkpoints are often used to recall a previous set of high-performing parameters based on validation error, and maintain a consistent performance across evaluations .
In RL, using the checkpoint of a policy that obtained a high reward during training, instead of the current policy, could improve the stability of the performance at test time.
For off-policy deep RL algorithms, the standard training paradigm is to train after each time step (typically at a one-to-one ratio: one gradient step for one data point). However, this means that the policy changes throughout each episode, making it hard to evaluate the performance. Similar to many on-policy algorithms , we propose to keep the policy fixed for several assessment episodes, then batch the training that would have occurred.
• Standard off-policy RL: Collect a data point → train once.
• Proposed: Collect N data points over several assessment episodes → train N times To ensure sample-efficient training while discouraging unstable policies, the **minimum** performance across a few evaluation episodes is used instead of the mean. This penalizes poor episodes more heavily and allows early resumption of training if a policy underperforms, saving resources. Although using many assessment episodes (20+) doesn't harm final policy quality, it slows early learning due to reduced policy updates and data diversity. Hence, the number of assessment episodes is kept low during early training and gradually increased later.

***TD7 algorithm:***


Pseudocode 1: TD7 Network Details

**Variables:**

zs_dim = 256

**Value Q Network:**

// TD7 uses two value networks each with the same network and forward pass.

l0 = Linear(state_dim + action_dim, 256)
l1 = Linear(zs_dim + 2 * 256, 256)
l2 = Linear(256, 256)
l3 = Linear(256, 1)

**Value Q Forward Pass:**

input = concatenate([state, action])
a = Avg(!Norm(l0(input)))
a = concatenate([zsa, za, a])
a = ELU(l1(a))
a = ELU(l2(a))
value = l3(a)

**Policy π Network:**

l0 = Linear(state_dim, 256)
l1 = Linear(zs_dim + 256, 256)
l2 = Linear(256, 256)
l3 = Linear(256, action_dim)

**Policy π Forward Pass:**

input = state
a = Avg(!Norm(l0(input)))
a = concatenate([zs, a])
a = Relu(l1(a))
a = Relu(l2(a))
action = tanh(l3(a))

**State Encoder f Network:**

l1 = Linear(state_dim, 256)
l2 = Linear(256, 256)
l3 = Linear(256, zs_dim)

**State Encoder f Forward Pass:**

input = state
a = ELU(l1(input))
a = ELU(l2(a))
zs = Avg(!Norm(l3(a)))

**State Action Encoder g Network:**

l1 = Linear(action_dim + zs_dim, 256)
l2 = Linear(256, 256)
l3 = Linear(256, zs_dim)

**State Action Encoder g Forward Pass:**

input = concatenate([action, zs])
a = ELU(l1(input))
a = ELU(l2(a))
zsa = l3(a)

**My implementation of TD7 and SALE: State Encoder:** A neural network that maps an input state (a vector from the environment) to a latent representation, $z_s$.
**Layers:** Two hidden layers with ReLU activations, followed by an output layer that produces the latent vector $z_s$.

**State Action Encoder:** This network takes the concatenation of a state's latent representation ($z_s$) and an action, and maps it to a state-action latent embedding, $z_{sa}$.
**Layers:** Similar structure as the State Encoder.

**Value Network:** A network that takes the state latent vector $z_s$ as input and outputs a scalar value (akin to a Q-value or state-value).
**Layers:** Two hidden layers with ReLU activations, followed by an output linear layer that produces the value.

**Policy Network:** The actor network that maps the original state to an action.
**Activation:** The final output uses a `tanh` activation scaled by the maximum action value to ensure the action is within the valid range.

**Prioritized Replay Buffer**
**Purpose:** Stores transitions (state, next_state, action, reward, done) along with a priority for sampling.
**Sampling:** When sampling a batch, transitions are selected based on their priorities, and importance-sampling weights are computed to correct for bias introduced by prioritized sampling.
**Priority Update:** After training on a batch, the TD errors (the absolute differences between predictions and targets) are used to update the priorities for each sample in the buffer.

**TD7 Agent**
The `TD7` class ties together all the networks and training logic.
**Components:** It consists of the Encoders (both the state encoder and state-action encoder, along with their target copies for stability), the Policy (the actor network and its corresponding target network), and the Value Networks (three separate value networks, and their target networks, employed to reduce overestimation bias by taking the minimum value estimate).
**Optimizers:** Separate Adam optimizers are set up for each network component.
Action Selection: Given a state, the policy network outputs an action, and external noise can be added for exploration.
**Training Routine (train method):** For each training iteration: (1) **Sampling:** A batch of transitions is sampled from the replay buffer. (2) **Encoding:** Current states are passed through the state encoder to obtain latent vectors, $z_s$, while next states are encoded using the target state encoder. (3) **Target Value Calculation:** The target networks for the value networks produce value estimates for the next states; the minimum of these estimates is taken, and an n-step discounted target is computed as

$$target\_value = reward + \big(done\_mask \times \gamma^{n_{step}} \times min\_target\_value\big).$$

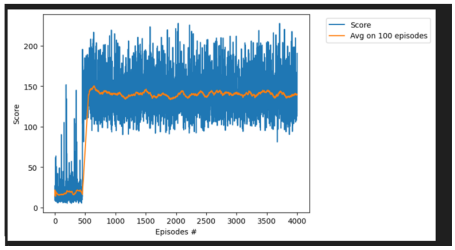
(4) **Value Network Update:** Each value network is updated by minimizing the mean squared error between its prediction (using $z_s$) and the computed target value. (5) **Replay Buffer Priorities Update:** TD errors (the absolute differences between predicted and target values) are computed and used to update the sampling priorities. (6) **Policy Update (Delayed):** Every other training iteration, the policy network is updated to maximize the value predicted by the first value network (i.e., gradient ascent on the Q-value), and Polyak averaging is used to softly update all target networks.

**Training and Evaluation Functions**

**twind7train:** This function runs the training loop over multiple episodes. In the Episode Loop, the environment is reset at the start of each episode; at each timestep within an episode, actions are chosen (randomly at first, then via the policy with added noise) and transitions are stored in the replay buffer; after each episode, if enough samples exist, training is performed, statistics are printed, and the agent is periodically saved.

**play:** After training, this function is used to evaluate the learned policy over a few episodes.

***Results and Evaluation:*** From this plot, we can see that the raw episode




returns (blue line) start out relatively low, then quickly jump to a higher level around the 200–300 episode mark. After that point, the returns fluctuate but remain roughly in the 100–200 range. Meanwhile, the 100-episode moving average (orange line) shows a smoother trend that settles somewhere in the 120–150 range for most of the training run.

Overall, this indicates that the agent does learn a policy that significantly outperforms its initial performance—evidenced by the sharp jump early on—and then plateaus at a stable (but noisy) level. The fluctuation in the raw scores (blue line) is common in reinforcement learning, where stochasticity in both the policy and the environment can cause variability in individual episode returns.

Key takeaways:

1. **Rapid Early Improvement**: The agent transitions from near-random performance to much higher returns within the first few hundred episodes.

2. **Plateau / Convergence**: After this jump, the performance stabilizes (albeit with noise) around a certain range, indicating the agent has converged to a reasonably good, though not necessarily optimal, policy.

3. **Noise vs. Average**: The moving average (orange) is a better indicator of true performance than the per-episode score, illustrating that despite noisy returns, the policy remains fairly consistent on average.

We can see that the for TD3 average rewards were around 40 and the convergence occured after 2000 episodes. Whereas in TD7 the average rewards are near to 140 and the covergence occurs around 600-700 episodes.

TD7 builds upon the foundational TD3 algorithm by introducing several key

enhancements aimed at improving performance stability and learning efficiency. One major improvement is the use of state-action learned embeddings, which encode both state and action information into a shared latent space. This expressive representation allows the algorithm to capture nuanced relationships between states and actions, leading to better credit assignment and more accurate Q-value estimates. Additionally, this approach contributes to smoother learning updates by enabling the network to disentangle important features before value prediction. TD7 also incorporates a robust checkpointing mechanism with a minimum-performance criterion, where periodic checkpoints and fallback policies help prevent performance regressions and promote stability. This ensures that the agent maintains a baseline of effective behavior while minimizing wasted training episodes. A critical distinction from TD3 lies in TD7's use of multiple value networks—three instead of two—taking the minimum output to further mitigate overestimation bias. Moreover, TD7 leverages n-step returns (e.g., 3-step or 7-step) to propagate rewards over multiple steps, accelerating learning and improving value propagation. These innovations result in more consistent and reliable learning curves, as evidenced by lower variance in performance and convergence to higher, more stable average scores. The synergy of prioritized replay, latent embeddings, multi-step returns, and checkpoint-based recovery allows TD7 to avoid large performance swings and catastrophic forgetting. In summary, TD7 extends TD3 with richer representations, enhanced stability mechanisms, and improved learning dynamics, yielding a more robust and effective reinforcement learning agent.