JSAC: A JAX/Flax Implementation of Vision-Based Soft Actor-Critic with Enhancements

Fahim Shahriar

April 26, 2025

1 Overview

JSAC is a Soft Actor-Critic (SAC) (Haarnoja et al., 2018) based Reinforcement Learning (RL) system designed for high-performance and stable learning from images. JSAC is well-suited for real-world, vision-based robotic tasks and learning from scratch. JSAC includes three features essential for real-world learning: (1) improved learning performance through the integration of state-of-the-art RL approaches, (2) accelerated learning via a JAX/Flax-based implementation, and (3) asynchronous neural network updates and replay buffer sampling for fast, smooth environment interactions. JSAC utilizes DrQ style image augmentation (Kostrikov et al., 2021), ensembles of Q-functions (Chen et al., 2021), and weight clipping (Elsayed et al., 2024) for robust learning performance from raw pixels. The JAX/Flax-based implementation of neural networks improves the action sample time and the update time by a factor of 3 to 4 compared to a PyTorch-based implementation. Parallelization of network updates and replay buffer sampling enables efficient usage of computing resources.

JSAC supports three types of environments: (1) proprioception-only, (2) image-only, and (3) proprioception-image-based environments. In a real-world setup, proprioception data correspond to numerical values such as joint positions or velocities of robotic arms, or wheel velocity and battery status of wheeled robots. Images are typically captured using an onboard camera and often stacked over multiple time-steps. In simulated environments, proprioception refers to numerical components of the state, and images are rendered views generated by the simulator.

2 Neural Network Architecture

JSAC's neural networks consist of four components: (1) an encoder network to process images, (2) an actor network, (3) a critic network, and (4) a temperature parameter. The networks are described in detail below.

2.1 Encoder Network

The encoder network converts an image of the shape $H \times W \times C$ into a latent vector $z \in \mathbb{R}^{d_z}$, where H, W, C denote the height, width, and stacked channels of the image, respectively. The encoder network consists of four CNN layers. Each CNN layer, except the last layer, is followed by a normalization layer (LayerNorm, if the hyperparameter *layer_norm* is set to True) and a LeakyReLU layer. The output of the last CNN layer is flattened to form the output vector z.

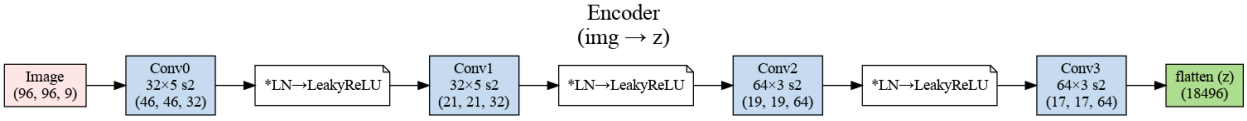


Figure 1: The Encoder Network structure for an input image of shape $96 \times 96 \times 9$ (three RGB images of the shape 96×96 stacked on the channel axis). LN denotes LayerNorm, and * denotes that LayerNorm is activated when the hyperparameter *layer_norm* is set to True. Each blue rectangle (convolutional layer) shows three lines: the layer name (top), the number of output channels with square kernel and stride sizes (middle), and the output-tensor shape (bottom).

2.2 Actor Network

The actor network maps (z, p) to a Tanh-squashed Gaussian policy π, where z is the output of the Encoder and p is the proprioception vector. The latent vector z passes through a Dense layer of size 64 (defined by the *latent_dim* hyperparameter), followed by LayerNorm and a tanh activation, and is concatenated with the proprioception vector p. The concatenated vector is then passed to a two-layer multilayer perceptron (MLP) with 1024 hidden units per layer, and each hidden layer is followed by LayerNorm (if the hyperparameter *layer_norm* is set to True) and ReLU. Two parallel linear heads output the mean μ and the log standard deviation $\log \sigma$. The pair $(\mu, \exp \log \sigma)$ parameterizes a Gaussian distribution, and the action (a) and its log-probability $(\log a)$ are obtained by sampling this Gaussian and applying a tanh transformation.

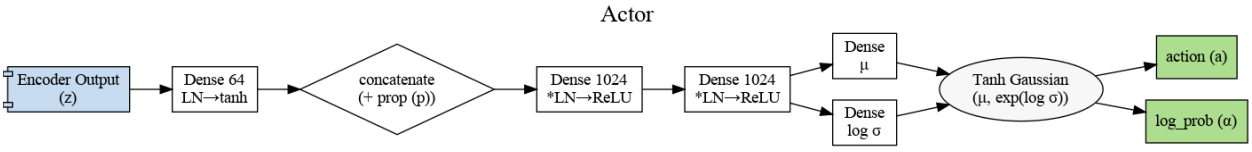


Figure 2: The Actor Network.

2.3 Critic Network

The critic network takes the encoder output z and the action a and returns an ensemble of Q_N Q-values, where $Q_N = 5$. As in the actor network, z is fed to a dense layer of size 64,

LayerNorm (if the hyperparameter *layer_norm* is set to True), and a tanh activation. The resulting vector is then concatenated to the action vector a. Each Q-network applies the same two-layer MLP used in the actor (1024 units per layer with LayerNorm and ReLU), followed by a final dense layer of size 1.

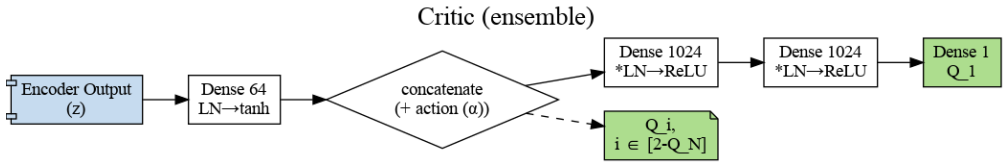


Figure 3: The Critic Network.

2.4 Temperature Parameter

JSAC automatically tunes the scalar temperature parameter α, which balances the reward maximization and entropy maximization objectives in SAC. This is implemented by maintaining a single learnable logarithmic parameter, $\log \alpha$.

2.5 Network Initialization

All dense and convolutional kernels use orthogonal initialization (Hu et al., 2020) with scale $\sqrt{2}$ (Eq. 1).

$$W_{\text{init}} \sim \text{Orthogonal}\big(\text{scale} = \sqrt{2}\big) \tag{1}$$

3 Learning Updates

Critic Update

For each transition $(s_t, a_t, r_t, s_{t+1}, m_t)$ sampled from the replay buffer, the one-step bootstrap target, y_t, is calculated as shown in Eq. 2.

$$y_t \;=\; r_t \;+\; \gamma\, m_t \left[\min_{i=1,\dots,Q_N} Q_{\phi_i'}(s_{t+1}, a_{t+1}) \;-\; \alpha\, \log \pi_\theta(a_{t+1} \,|\, s_{t+1})\right] \tag{2}$$

$$m_t = \begin{cases} 0, & \text{if t is a terminal state,} \\ 1, & \text{otherwise.} \end{cases}$$

The critic parameters Q_{ϕ_i} are updated by minimizing the mean-squared Bellman error:

$$\mathcal{L}_{\text{critic}}(\phi) \;=\; \frac{1}{Q_N} \sum_{i=1}^{Q_N} \Big[\, Q_{\phi_i}(s_t, a_t) \;-\; y_t \Big]^2$$

Actor Update

The actor parameters θ are updated by minimizing the expected soft-value objective:

$$\mathcal{L}_{\text{actor}}(\theta) \;=\; \Big[\, \alpha \, \log \pi_\theta(a_t \mid s_t) \;-\; \min_{i=1,\ldots,Q_N} Q_{\phi_i}(s_t, a_t) \Big] \tag{3}$$

Temperature Update

The entropy coefficient α is adjusted by minimizing the loss

$$\mathcal{L}_\alpha(\alpha) = \Big[-\alpha \big(\log \pi_\theta(a_t \mid s_t) + \bar{\mathcal{H}} \big) \Big]$$

where $\bar{\mathcal{H}}$ is the target entropy.

Stabilizing Neural Network Updates

JSAC implements multiple approaches to stabilize neural network updates for the actor and critic networks. JSAC clips the gradients to a maximum global norm of 1.0 (Pascanu et al., 2013) to address the issue of exploding gradients. If the hyperparameter *apply_weight_clip* is set to True, after each parameter update, JSAC clips the network weights to be within $[-\kappa s_l,\, \kappa s_l]$ to prevent parameter divergence (Elsayed et al., 2024), following Eq. 4.

$$W_{\text{clipped}} = \text{clip}\big(W_{\text{new}},\, min = -\kappa s_l,\, max = \kappa s_l\big) \tag{4}$$

Where, $\kappa = 2.0$, and $s_l = \sqrt{2}$. Finally, LayerNorms are used before the activation layers if the hyperparameter *layer_norm* is set to True, to help prevent network collapse.

For the Mujoco and DMC experiments, ***apply_weight_clip*** **and** ***layer_norm*** **are both set to False**, as the experiments are stable without these features. However, for more complex simulated environments and real-world tasks, it is recommended to use these features.

DrQ-Style Image Augmentation

JSAC includes DrQ-style image augmentation (Kostrikov et al., 2021) to improve performance when learning directly from pixels. This augmentation is applied to the image components of the sampled states s_t and the next states s_{t+1} before they are processed by the networks. The augmentation implements a random shift by first padding each image with 4 pixels using edge replication, and subsequently extracting a random crop of the original size. This process uses independent random offsets for each image within the batch.

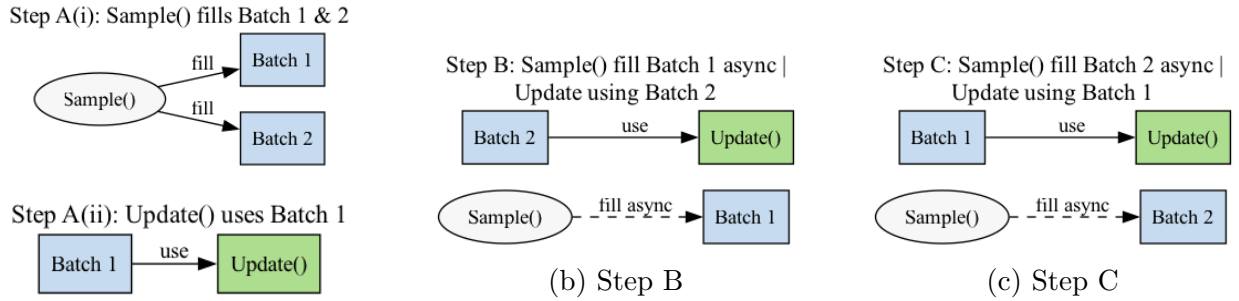
4 Asynchronous Computation

JSAC's computation can be grouped into three main components: (1) environment interactions, (2) learning updates, and (3) replay buffer sampling, and JSAC supports the parallel execution of all three components. In the environment interaction process, the agent receives the current state from the environment and uses the actor network to produce actions, which

are then applied to the environment. Performing learning updates and replay buffer sampling in parallel processes allows the agent to have shorter action cycle times, making the agent more responsive. This also improves resource utilization, allowing faster convergence (Yuan et al., 2022).
JSAC follows a similar inter-process data transfer approach described in Yuan et al. (2022, Section V. The Learning Architecture). In step t, the environment interaction process produces transitions $(s_t, a_t, r_t, s_{t+1}, m_t)$, which are then passed to the replay buffer using a multi-processing queue. The replay buffer samples a batch of data and passes it to the update process using shared memory buffers. Finally, the trained actor parameters, θ, are sent to the environment interaction process from the update process every K step, $(K = 8)$.

Asynchronous Replay Buffer with Shared Memory

Sharing a large volume of data, such as batch data for learning updates, over a multiprocessing queue can add significant computational overhead. JSAC utilizes shared memory based buffers and a round-robin batch data sampling strategy to virtually remove replay buffer sampling times for image-based tasks. The batch data sampling strategy is explained in Figure 4. The replay buffer contains two shared memory based batch data containers: Batch 1 and Batch 2. In the first sampling step (Step A), the sampling function fills both Batch 1 and Batch 2, and Batch 1 is sent to the learning update process. In the next step (Step B), Batch 2 is sent to the learning update process, and the sampling function fills up Batch 1 asynchronously. In the third step (Step C), Batch 1 is sent again to the learning update process, and the sampling function fills up Batch 2 asynchronously. This process is repeated for all remaining steps.



(a) Step A (b) Step B (c) Step C

Figure 4: Replay buffer operations.

5 Hyperparameters

JSAC hyperparameters used in different tasks are described in detail here:
https://github.com/fahimfss/JSAC/blob/main/hyperparameters.md

References

Chen, Xinyue et al. (2021). *Randomized Ensembled Double Q-Learning: Learning Fast Without a Model*. arXiv: 2101.05982 [cs.LG]. URL: https://arxiv.org/abs/2101.05982.

Elsayed, Mohamed et al. (2024). *Weight Clipping for Deep Continual and Reinforcement Learning*. arXiv: 2407.01704 [cs.LG]. URL: https://arxiv.org/abs/2407.01704.

Haarnoja, Tuomas et al. (2018). *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. arXiv: 1801.01290 [cs.LG]. URL: https://arxiv.org/abs/1801.01290.

Hu, Wei, Lechao Xiao, and Jeffrey Pennington (2020). *Provable Benefit of Orthogonal Initialization in Optimizing Deep Linear Networks*. arXiv: 2001.05992 [cs.LG]. URL: https://arxiv.org/abs/2001.05992.

Kostrikov, Ilya, Denis Yarats, and Rob Fergus (2021). *Image Augmentation Is All You Need: Regularizing Deep Reinforcement Learning from Pixels*. arXiv: 2004.13649 [cs.LG]. URL: https://arxiv.org/abs/2004.13649.

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio (2013). *On the difficulty of training Recurrent Neural Networks*. arXiv: 1211.5063 [cs.LG]. URL: https://arxiv.org/abs/1211.5063.

Yuan, Yufeng and A. Rupam Mahmood (2022). *Asynchronous Reinforcement Learning for Real-Time Control of Physical Robots*. arXiv: 2203.12759 [cs.RO]. URL: https://arxiv.org/abs/2203.12759.