AIM - Task 1 - 2 Goal Reaching Acrobot

Tariq Anwaar Jaheer Hussain

May 15, 2025

Theory

The code can be found at the following GitHub link:
https://github.com/jhtariq/AIM$_task$

In this project, a codebase was given that had the implementation code of DDPG for a 2D goal-reaching acrobot. The high-level goal of this task was to improvize the sandbox in as many ways, making sure the code is of production level standards. One of the major contributions as part of this task is the implementation of SLAC for the problem proposed. SLAC is an algorthimic technique that uses a VAE to train the actor and critic networks in a RL environment.

Problem Statement

The problem statement is a 2D acrobot with 2 links. There is a target (box) that is constantly moving in every iteration. The goal is to come up with a policy that will allow the link to exactly catch the target in as many iterations as possible.

Motivation for Improvization
The initial codebase had a vanilla DDPG implementation. An analysis was done to understand the strengths and shortcomings of the DDPG algorithm, keeping in mind real-world scenarios.

The implementation was pretty robust and computationally economical. However, to get the 2D state of the target (box), the method had used forward-kinematics. In a real-world scenario, especially where this 2 link acrobot could represent a heavily simplified excavator, even though there will motor encoders at every joint to give out the angles, it would be subject to high noise. Hence, forward-kinematics to calculate the state is not a viable and accurate solution. Hence, images become a viable source of information that could be potentially used to get the state of the target. However, images are of high dimensionality and hence it would be a computationally expensive process to use images to develop a RL policy.

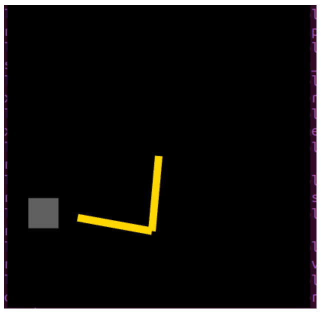
Stochastic Latent Actor Critic for RL policy
SLAC is an innovative algorithm that is capable of using images to train RL policies at a nominally acceptable computational cost. It ”compresses” images by using a VAE, converting an image to a latent representation. This VAE is not the conventional one but rather a stochastic latent model (explained in further sections). The next page goes through implementation of SLAC for the given problem.

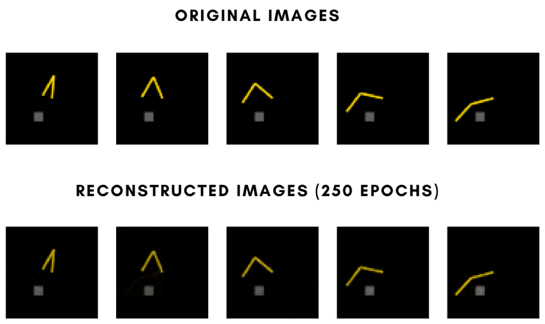
Architecture Details

This page goes through the implementation of SLAC for the given problem. The problem can be briefly split into latent space conversion, critic training and actor training. Figure 1i shows a screenshot of the 2D environment. Figure 1ii shows a premature screenshot of the output of the decoder. More mature screenshots are uploaded in the github link submitted. All parameters pertaining to the properties of the links can be found in the file called env.py

Latent Space Conversion - VAE Explained

The integration of a VAE pipeline to train a RL policy is probably the most innovative concept of SLAC as the dimensionality decreases extensively due to the conversion to latent space. Conventional VAEs usually have an encoder and a decoder, where the encoder (posterior) converts the image to the latent space and the decoder tries to convert the latent space back to the original image. However, this directly cannot be used to train a RL policy. Afterall, we are trying to estimate the state of the target. Hence in this latent dynamics model, we have another term called prior. The prior takes in the latent space output as well as the action and then tries to estimate the latent space representation. This VAE is governed by 2 loss terms. The KL divergence loss measures the difference between the posterior and the prior while the reconstruction loss measures how close the decoder can reconstruct from the given latent space. For implementation, the model details can be found in the file, `model_distribution_with_network.py`


(i) Environment



(ii) Latent Representation

Figure 1: Environment and VAE Setup

Actor and Critic Networks

The critic and actor are the fundamental components of any RL algorithm. The critic learns to predict the expected return from a latent-state-action pair. It comes up with the best estimates of the Q value function and the TD error is computed. The reward here combines several terms, namely the next immediate rewards as well as a entropy term that encourages exploration over exploitation.

The actor on the other hand learns a stochastic policy and is based on the distribution over actions conditioned on history. The loss term includes the entropy term, score of the action from the Q value function and the confidence of the actor in sampling an action.

Code Implementation Details

The code was modified such that it remains modular to a good extent. Several unit tests were performed to evaluate the sanity of individual subsystems like the encoder, decoder, prior and the latent model. Dummy values were used to check the size of the desired outputs of these subsystems. All of these tests can be found under the /tests folder.

Wrapper file for environment

Next, the file env.py represents the environment which was given. There wasn't much changes done to it. As explained above a function was scripted to take screenshots of the environment, which are then passed to the VAE. To maintain the overall modularity, a wrapper function was made. This wrapper function was also necessary as one cannot use the pre-developed OpenGYM or DMcontrol wrappers available in the SLAC repo. The wrapper file is named as `arm_env_wrapper.py`.

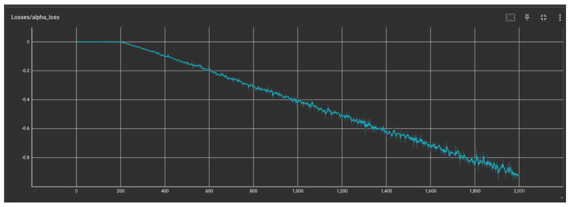
Two important functions defined here are the step and the reset functions. These functions are triggered at the end of every episode and bring about the next observation.

VAE, Critic and Actor

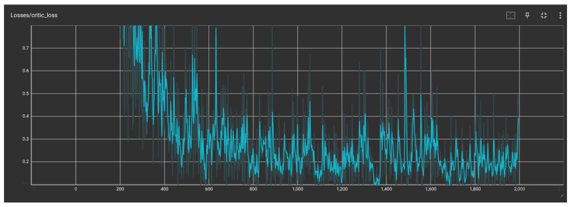
The SLAC paper had implemented the VAE, Critic and Actor architectures. The code was used for repeatability and convenience. The VAE file is `arm_env_wrapper.py`. The critic file is `critic_network.py`. The actor file is `actor_distribution_network.py`.

Training and Evaluation

The SLAC github repository had also provided an exhaustive training and evaluation file called `train_eval.py`. A custom training and evaluation file was created, named `run_slac.py`, customly importing the ARM environment wrapper and other necessary files. Note that this is the primary file and is used to run the entire architecture. A set of training parameters were given in the SLAC repo. Due to time and computational constraints, I was not able to run a full-fledged run and had used down-scaled numbers. I have presented the loss curves based on the preliminary results. Figures 2i and 2ii represent the actor and critic loss respectively. The results shown are run for around 2000 iterations, which is significantly lesser that the reccomended range of 10000000 iterations.



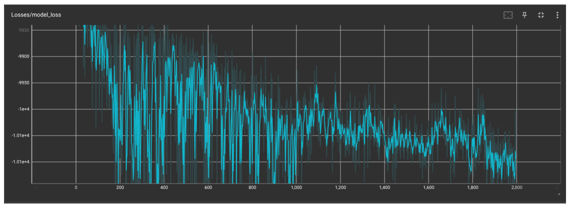
(i) Actor Loss



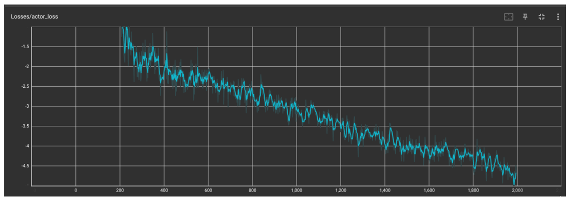
(ii) Critic Loss

Figure 2: Environment and VAE Setup

Figure 3 represents the model loss, which refers to the VAE model loss. It is to be noted here that in the SLAC algorithm the VAE, Actor and Critic get trained in parallel. Furthermore, we can optimize the number of training iterations in one master iteration for each of these. There is also an option to use a pre-trained VAE model. Since I did not use a pre-trained model, the losses start at 10000 and slowly falls as it trains. The alpha loss refers to the loss function plot of one of the learning parameters, alpha.



(i) Model Loss



(ii) Alpha Loss

Figure 3: Environment and VAE Setup

Scopes of Improvement

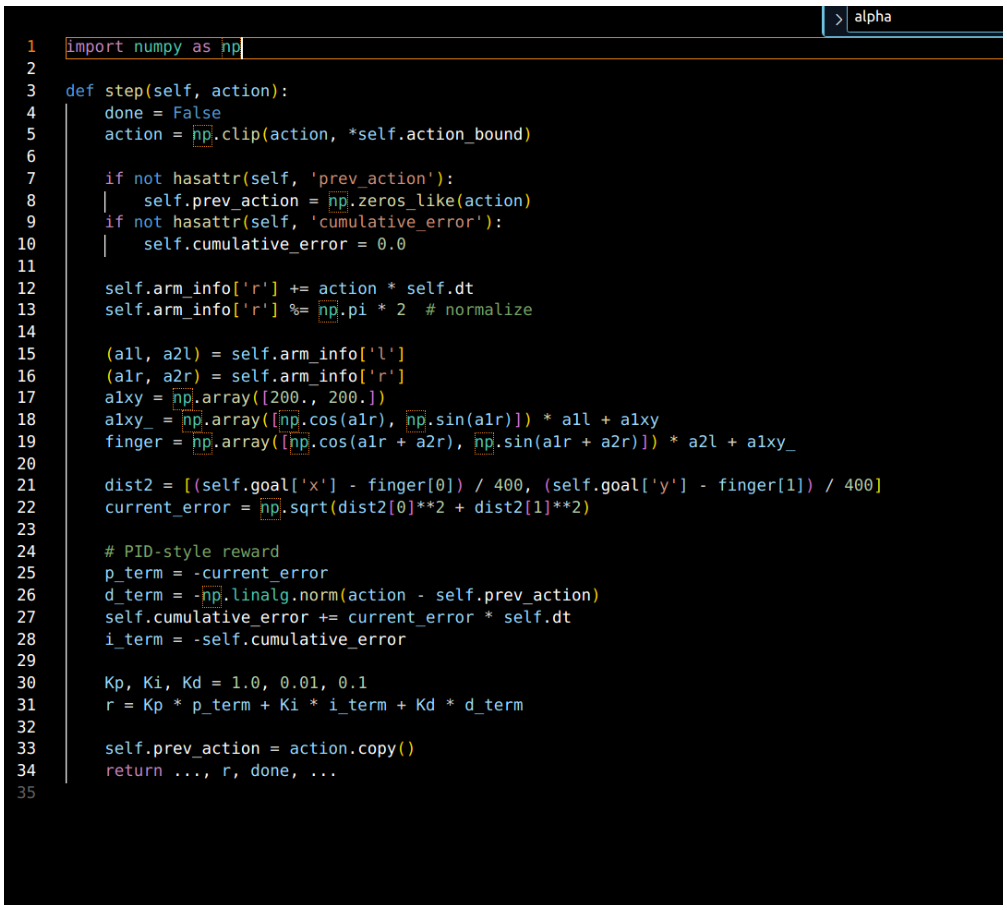
Due to some time constraints with my graduation, I was not able to actively explore other areas of this sandbox which can potentially improve the task accuracy. Here are some starter thoughts and individual code snippets that try to bring an essence of the proposed plans.

Hyperparameter Tuning

Due to time and computational constraints, I could not tune the hyperparameters.

Reward function tuning

The current reward function uses the euclidean distance between the finger and the target. From experience, it has been found that sometimes changing the reward term to exponential(distance) could help improve the training performance. Furthermore, as suggested, a PID tuning that penalizes jerky motions and long trajectory motions could help in regularizing the actions. The proposed action regularizer is as shown in Figure 4.



```python
import numpy as np

def step(self, action):
    done = False
    action = np.clip(action, *self.action_bound)

    if not hasattr(self, 'prev_action'):
        self.prev_action = np.zeros_like(action)
    if not hasattr(self, 'cumulative_error'):
        self.cumulative_error = 0.0

    self.arm_info['r'] += action * self.dt
    self.arm_info['r'] %= np.pi * 2  # normalize

    (a1l, a2l) = self.arm_info['l']
    (a1r, a2r) = self.arm_info['r']
    a1xy = np.array([200., 200.])
    a1xy_ = np.array([np.cos(a1r), np.sin(a1r)]) * a1l + a1xy
    finger = np.array([np.cos(a1r + a2r), np.sin(a1r + a2r)]) * a2l + a1xy_

    dist2 = [(self.goal['x'] - finger[0]) / 400, (self.goal['y'] - finger[1]) / 400]
    current_error = np.sqrt(dist2[0]**2 + dist2[1]**2)

    # PID-style reward
    p_term = -current_error
    d_term = -np.linalg.norm(action - self.prev_action)
    self.cumulative_error += current_error * self.dt
    i_term = -self.cumulative_error

    Kp, Ki, Kd = 1.0, 0.01, 0.1
    r = Kp * p_term + Ki * i_term + Kd * d_term

    self.prev_action = action.copy()
    return ..., r, done, ...
```

Figure 4: Action Regularizer

Testing and Production Strategy

To ensure that the SLAC implementation is robust and production-ready, the following strategies can be considered:

- **Real-world Testing:** Current system uses visual debugging and observation checks. To generalize better, the pipeline can include noise injection, randomized goal motion and even motion blur.
- **Testing Process:**
  - *Unit Tests* for encoder, decoder, prior, and critic to validate shapes and loss trends. (Implemented)
  - *Integration Tests* to ensure the environment, replay buffer, and VAE interact properly.(Implemented)
  - *Regression Tests* using fixed seeds to detect performance drops after new changes.(Not Implemented)
- **Release Strategy:** A modular training file (`run_slac.py`) allows easy parameter edits and launching via:

  ```
  python run_slac.py
  ```

  For finer-grained control, one can adjust settings in `train_eval.py`. Git versioning with branches (e.g., `main`, `dev`) helps manage team development.
- **Streamlined Pipeline:** Testing automation via GitHub Actions or shell scripts, coupled with unit tests, ensures safe updates. Logs and TensorBoard summaries help trace training health.
- **Sim and Hardware Testing:** While current tests are in simulation, hardware-in-the-loop (HIL) integration is feasible via serial or ROS communication. Latent policy outputs can be tested on real-world robot arms using the same wrapper structure.