SAIDL Report-2

Tanmay Joshi

March 2025

1 Reinforcement Learning - 2-a

Problem Statement:Implement Independent Gaussian Noise variation from Noisy Nets for Exploration Paper from your actor - network from TD3
a)Evaluate performance gained and experiment with different sample distributions.Report your findings and inferences.
b)How would you test the adversarial robustness of the system?
Paper:https://openreview.net/pdf?id=rywHCPkAW

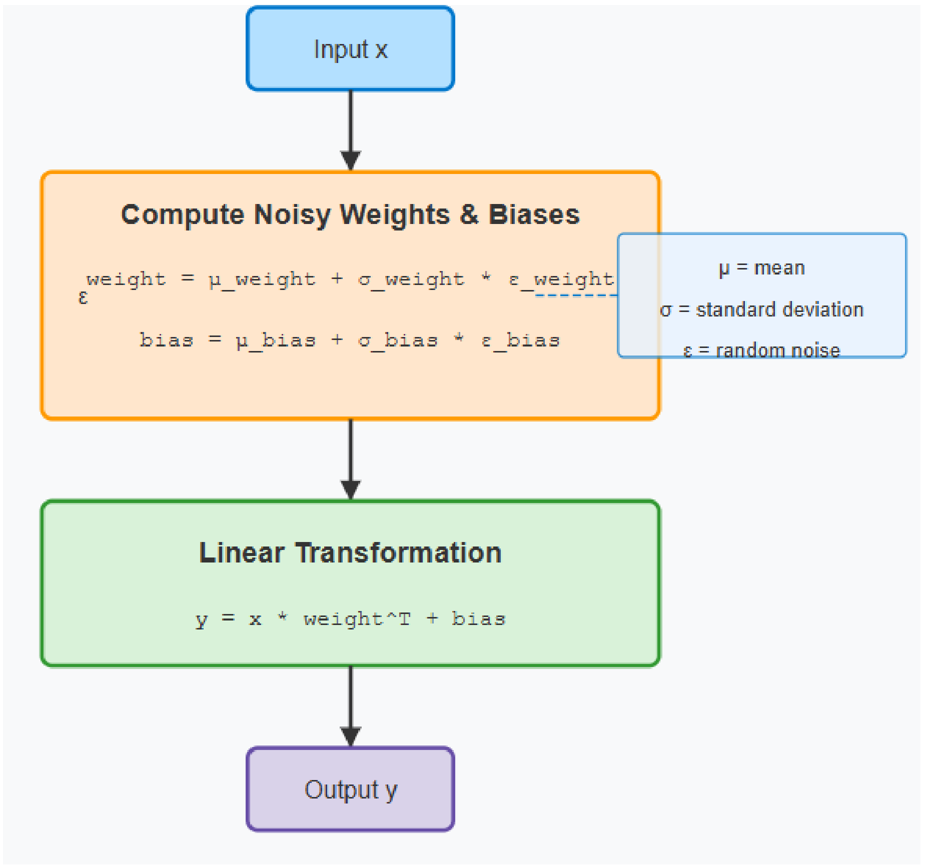
1)Independent Gaussian Noise:
Background: In the formulation of **Independent Gaussian Noise** for Noisy Networks, each weight and bias in the neural network is perturbed independently using Gaussian noise. Specifically, the noisy parameters are defined as

$$\theta = \mu + \sigma \circ \varepsilon,$$

where μ and σ are learnable parameters, ε is a noise vector sampled from a standard normal distribution, and $\circ$ denotes element-wise multiplication. For a linear layer with input dimension p and output dimension q, the weight matrix $w \in R^{q\times p}$ and bias vector $b \in R^q$ are replaced with noisy versions:

$$w = \mu_w + \sigma_w \circ \varepsilon_w, \quad b = \mu_b + \sigma_b \circ \varepsilon_b.$$

Here, each element $\varepsilon^w_{i,j}$ and ε^b_j is sampled independently from a standard normal distribution $\mathcal{N}(0, 1)$. Consequently, each noisy linear layer requires $pq + q$ independent noise variables. This approach ensures maximum flexibility and randomness, allowing precise perturbation at the level of individual parameters. However, it incurs higher computational cost due to the need to generate and apply a large number of independent noise values, which can be inefficient for resource-constrained environments or single-threaded agents.
My implementation of Independent Gaussian Noise-
1)Class Independent Gaussian Noise- The layer receives an input tensor x representing a batch of data, which is then processed by a custom linear transformation. During training, the layer introduces stochasticity by incorporating noise into its parameters. Specifically, noise values ε are generated from



a specified distribution (Gaussian, Beta, or Gamma) using the `reset_noise` method. These noise values are then combined with the learnable parameters. The noisy weight is computed as

$$weight = \mu_{weight} + \sigma_{weight} \circ \varepsilon_{weight},$$

where μ_{weight} and σ_{weight} are the base and scaling parameters for the weights, and $\circ$ denotes element-wise multiplication. Similarly, the noisy bias is computed as

$$bias = \mu_{bias} + \sigma_{bias} \circ \varepsilon_{bias}.$$

After these computations, the layer applies a standard linear transformation using the function $F.linear(x, weight, bias)$ to yield the final output y. This output y is then passed on to subsequent layers in the network, allowing the model to benefit from both deterministic and stochastic learning components.
TD3 implementation for independent gaussian noise: (The dual actor critic network remains the same) The **TwinDelayedAgent** class is an implementation of a variant of the TD3 algorithm that leverages noisy networks for

improved exploration. In the constructor, two copies of the `PolicyNetwork` are created: one represents the current policy (actor) and the other is the target policy, with the latter initialized by copying the weights from the former. Similarly, two `ValueNetwork` instances (serving as critics) are instantiated—one for the online estimation and another as the target network. Separate Adam optimizers are used for updating the policy and value networks.
During training, a batch of experience is sampled from the replay buffer and converted to tensors. The target Q-value is computed by feeding the next state into the target policy to obtain the next action, and then evaluating this action with the target value network, taking the minimum of the two Q-value estimates. This target Q-value is combined with the immediate reward, using a discount factor γ, to form the regression target. The critic loss is the sum of mean squared errors between the current Q-values (obtained from the online value network) and this target.
For the actor update, the policy network is optimized by maximizing the Q-value estimated by the first head of the value network, which is equivalent to minimizing the negative Q-value. Following these updates, a soft update with a parameter τ is applied to slowly track the changes in the online networks with the target networks. Finally, the agent calls `reset_noise()` on the policy network to resample the noise injected into its parameters, thereby ensuring fresh exploratory behavior. The `select_action` method converts a given state into a tensor, processes it through the policy network, and returns the action as a NumPy array.
Training Function- The function `train_td3` (which trains the TwinDelayedAgent) begins by creating a Hopper-v4 environment using Gym. It extracts the dimensions of the state and action spaces and determines the maximum action value. An agent is then instantiated as a `TwinDelayedAgent`, with its policy (actor) using a specified noise distribution (passed as `noise_dist`) for its noisy linear layers. A replay memory, here renamed as `PrioritizedMemory`, is also initialized to store experience tuples.

For each episode (up to `num_episodes`), the environment is reset and the cumulative reward is initialized. The inner loop runs for a maximum of `max_steps` per episode. In each step:

- The agent selects an action based on the current state using its policy network.
- This action is executed in the environment (via `env.step(action)`), returning the next state, reward, and termination signals.
- The experience tuple `(state, action, reward, next_state, done)` is added to the replay memory.
- The state is updated to the next state and the episode reward is accumulated.
- If the episode is done (either terminated or truncated), the inner loop exits.

- Additionally, if the replay memory contains more than 1000 samples, the agent trains on a batch of 64 experiences using its `train` method.

Every 50 episodes, the function prints out the current noise distribution, episode number, and episode reward for monitoring. After all episodes have been run, the environment is closed, and the list of episode rewards is returned.
Finally, the code trains separate agents using three different noise distributions: `'gaussian'`, `'beta'`, and `'gamma'`. It then plots the episode rewards over time to compare the performance of the agent under these different noise settings.

What are 'beta' and 'gamma' distributions? The Beta distribution is defined on the interval $[0, 1]$ and is controlled by two shape parameters, typically denoted as α and β. Its shape can be symmetric or skewed depending on the values of these parameters. In the implementation, after sampling from the Beta distribution, the values are rescaled (e.g., mapping the range $[0, 1]$ to $[-1, 1]$) to serve as noise. This allows for a bounded, flexible form of randomness that can be tuned via α and β. The Gamma distribution is defined for positive values and is determined by a shape parameter and a scale parameter. It typically has a mean of $shape \times scale$ and a variance of $shape \times scale^2$. In the noisy network, samples from the Gamma distribution are normalized (by subtracting the mean and dividing by the standard deviation) to produce noise that can be applied similarly to Gaussian noise. This distribution can provide a different kind of variability compared to the Beta or Gaussian distributions.

Noisy Parameters - An improvement to the code In this implementation, the parameter `noise_params` is an optional dictionary that allows you to customize the parameters of the noise distribution used for perturbing the weights and biases.

For the Beta Distribution (`noise_dist == 'beta'`):
The `noise_params` dictionary can include keys "`alpha`" and "`beta`", which control the shape of the Beta distribution. By default, if these keys are not provided, the values

$$\alpha = 2.0 \quad and \quad \beta = 2.0$$

are used. After sampling from the Beta distribution (which normally yields values in the interval $[0, 1]$), the code rescales these samples to roughly the range $[-1, 1]$ by subtracting 0.5 and multiplying by 2.

For the Gamma Distribution (`noise_dist == 'gamma'`):
In this case, the `noise_params` dictionary can include the keys "`shape`" and "`scale`", which determine the Gamma distribution's characteristics. The default values are

$$shape = 2.0 \quad and \quad scale = 1.0.$$

The Gamma distribution produces positive values with a mean of

$$mean = shape \times scale$$

and a standard deviation of

$$std = \sqrt{shape} \times scale.$$

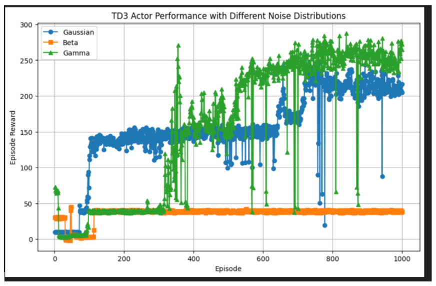
After sampling, these values are normalized by subtracting the mean and dividing by the standard deviation to obtain noise with zero mean.

For the default Gaussian noise, no additional parameters are needed since the noise is sampled from the standard normal distribution $\mathcal{N}(0, 1)$.

Thus, the `noise_params` dictionary provides flexibility to tune the noise characteristics when using Beta or Gamma distributions in your noisy network layers.

Experiments and Results:

With default parameters: The figure shows the performance of three TD3



agents over 1,000 episodes with default noise parameters. Each curve corresponds to a different noise distribution used for perturbing the network parameters in the NoisyNet-based actor.

Gaussian Noise (Blue):
The agent using Gaussian noise (i.e., samples from $\mathcal{N}(0, 1)$) exhibits the highest and most stable rewards. The unbounded nature of Gaussian noise allows for more flexible exploration of the parameter space, enabling the agent to discover and sustain high-return policies.

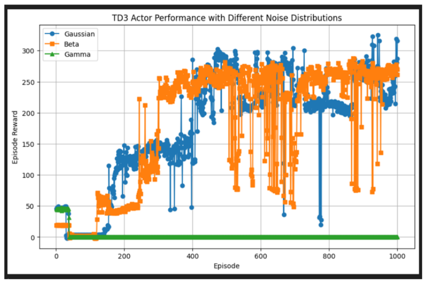
Beta Noise (Orange):
The agent with Beta noise maintains relatively low rewards throughout training. Although the Beta distribution is rescaled to $[-1, 1]$ with default parameters ($\alpha = 2.0$, $\beta = 2.0$), it may result in more constrained exploration. This could lead to the agent getting stuck in suboptimal regions of the policy space.

Gamma Noise (Green):
The Gamma noise-based agent achieves moderate rewards, performing better than Beta but generally not reaching the level of Gaussian noise. The Gamma distribution, even after normalization, provides a different pattern of perturbation that may not align as well with the exploration requirements of the Hopper-v4 environment.

Conclusions:

- With default parameters, **Gaussian noise** yields the best overall performance, suggesting that its flexibility in exploration is well-suited for this environment.
- **Beta** and **Gamma** noise, while providing alternative perturbation schemes, do not perform as well under default settings. Their exploration capabilities might be enhanced by tuning their respective parameters (α, β for Beta; shape and scale for Gamma).
- In practice, when considering Beta or Gamma noise for exploration, careful parameter tuning is recommended to potentially match or exceed the performance achieved with Gaussian noise.

  For different parameters-



In this experiment, we evaluate the performance of a TD3-based agent using different noise distributions for exploration. Specifically, we compare:

- **Beta noise** ($\alpha = 3.0,\ \beta = 4.0$)
- **Gaussian noise** ($\mathcal{N}(0, 1)$)
- **Gamma noise** ($shape = 3.0,\ scale = 0.5$)

Beta Noise (Orange Curve)
With $\alpha = 3.0$ and $\beta = 4.0$, the Beta-distributed noise yields the highest rewards overall, indicating that these parameters encourage effective exploration for the given environment. The agent consistently converges to higher-return policies compared to the other distributions.

Gaussian Noise (Blue Curve)
Using standard Gaussian noise, the agent achieves moderate performance. While Gaussian noise often serves as a reliable baseline, it does not reach the same peak performance seen with the tuned Beta parameters in this environment.

Gamma Noise (Green Curve)
With $shape = 3.0$ and $scale = 0.5$, the Gamma-distributed noise produces more volatile returns, occasionally spiking to high rewards but generally

trailing the other distributions. Further tuning of the Gamma parameters might be necessary to improve its stability and overall performance.

Conclusion

- **Beta noise** with the specified parameters outperforms both Gaussian and Gamma noise, highlighting the importance of tuning the distribution.
- **Gaussian noise** remains a dependable default but shows lower returns compared to the tailored Beta distribution in this setup.
- **Gamma noise** can achieve moderate results but appears more sensitive to parameter choices, suggesting that careful experimentation with shape and scale is needed for optimal exploration.

Why does Gaussian Noise show low rewards in this setup? Gaussian noise, drawn from the standard normal distribution $\mathcal{N}(0, 1)$, is widely used for parameter perturbation in noisy networks due to its inherent symmetry and predictable statistical properties. Its symmetric nature, with zero mean and unbounded support, ensures that weight and bias updates can be both positive and negative, providing balanced exploration during training. Although the raw noise has a variance of 1, the effect on the network is controlled by learnable scaling parameters, σ, which determine the magnitude of the perturbation. This consistent variance and bell-curve shape contribute to stable exploration behavior, making Gaussian noise a reliable baseline. In contrast to more skewed or bounded alternatives, Gaussian noise typically avoids extreme outliers while still allowing occasional larger deviations, enabling the agent to effectively explore the parameter space without overshooting the optimal solution. As a result, Gaussian noise offers a flexible yet straightforward approach to injecting randomness into the network, facilitating efficient exploration in reinforcement learning applications.

2 Reinforcement Learning - 2-b

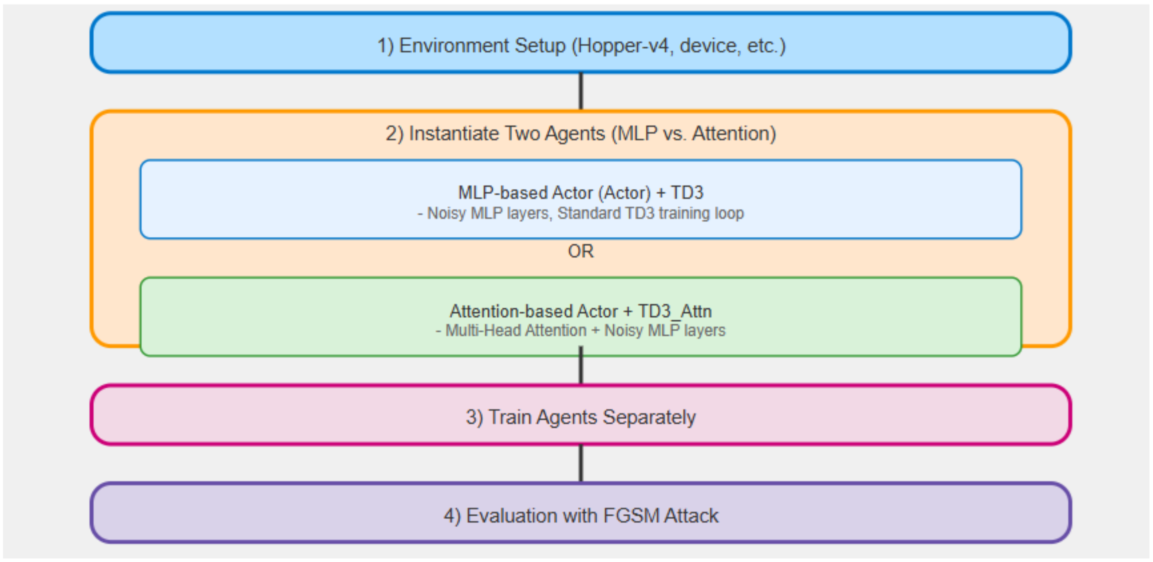
Testing Adversarial Robustness of the System using FGSM based Attack- In our experimental setup, adversarial robustness is tested using an ***FGSM-based attack*** on the input state. The function `fgsm_attack` computes the gradient of the critic's Q-value with respect to the state, and perturbs the state in the direction that would most degrade the performance of the agent. By varying the adversarial perturbation strength (denoted by ϵ), we evaluate how the performance (average episode reward) degrades under increasingly adversarial conditions.

Two variants of the TD3 agent are considered: a baseline agent with an MLP-based actor that uses noisy linear layers, and an alternative agent where the actor incorporates an attention-based

encoder block. The attention mechanism works by embedding individual state features as tokens and **applying multi-head self-attention,** which allows the network to capture global relationships among state features. **This may provide enhanced robustness against adversarial perturbations compared to a standard MLP encoder,** which lacks this global context.

Moreover, different encoder blocks such as Convolutional Neural Networks (CNNs) or attention heads inherently process data differently. **CNNs, with their localized receptive fields, might be more sensitive to small, localized perturbations, whereas attention mechanisms can re-weight global information and potentially mitigate the impact of adversarial noise.** The evaluation thus provides insight into how architectural choices affect the resilience of the agent in adversarial scenarios.

Code Workflow to check robustness-

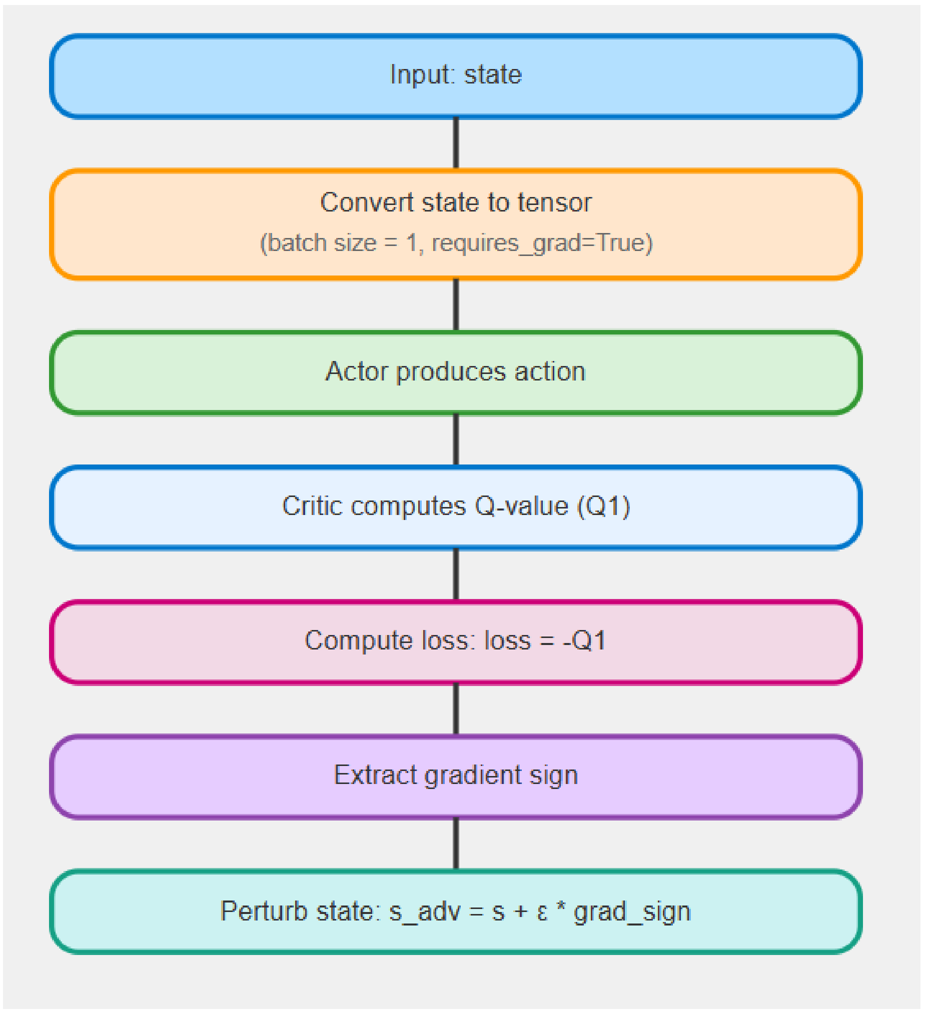


1) Environment Setup (Hopper-v4, device, etc.)
2) Instantiate Two Agents (MLP vs. Attention)
MLP-based Actor (Actor) + TD3
- Noisy MLP layers, Standard TD3 training loop
OR
Attention-based Actor + TD3_Attn
- Multi-Head Attention + Noisy MLP layers
3) Train Agents Separately
4) Evaluation with FGSM Attack

Implementation of fgsm:

The FGSM (Fast Gradient Sign Method) attack in this code is designed to create a small adversarial perturbation to the input state that degrades the agent's performance. Here's a step-by-step explanation:

1. **Input Conversion and Gradient Setup:** The function first converts the input state into a PyTorch tensor and reshapes it into a batch with one sample. By setting `s.requires_grad = True`, it marks the state tensor for gradient computation, which is necessary


Input: state
Convert state to tensor
(batch size = 1, requires_grad=True)
Actor produces action
Critic computes Q-value (Q1)
Compute loss: loss = -Q1
Extract gradient sign
Perturb state: s_adv = s + ε * grad_sign

to determine how changes in the state affect the critic's output.

2. **Forward Pass through the Networks:** The agent's actor network is used to produce an action from the current state. Then, this state and action are passed into the critic's Q1 function to compute the Q-value. This value represents the critic's estimation of the quality of that state-action pair.

3. **Defining the Loss:** The loss is defined as the negative of the Q-value (`loss = -q_val`). The idea is that by maximizing the loss (or equivalently, minimizing the Q-value), the attack degrades the agent's performance. Essentially, it seeks to find a small perturba-

tion in the state that would lower the Q-value.

4. **Backward Pass to Compute Gradients:** The gradients of the loss with respect to the state tensor are computed via `loss.backward()`. These gradients indicate the direction in which the state should be altered to increase the loss.

5. **Creating the Adversarial Perturbation:** The sign of the gradient is then taken (using `s.grad.data.sign()`), which gives the direction of steepest ascent for the loss. Multiplying this sign with a small factor `epsilon` (which controls the perturbation magnitude) and adding it to the original state produces the adversarial state.

6. **Returning the Perturbed State:** Finally, the adversarial state tensor is detached from the computation graph, converted back to a NumPy array, and flattened to be used in further evaluation.

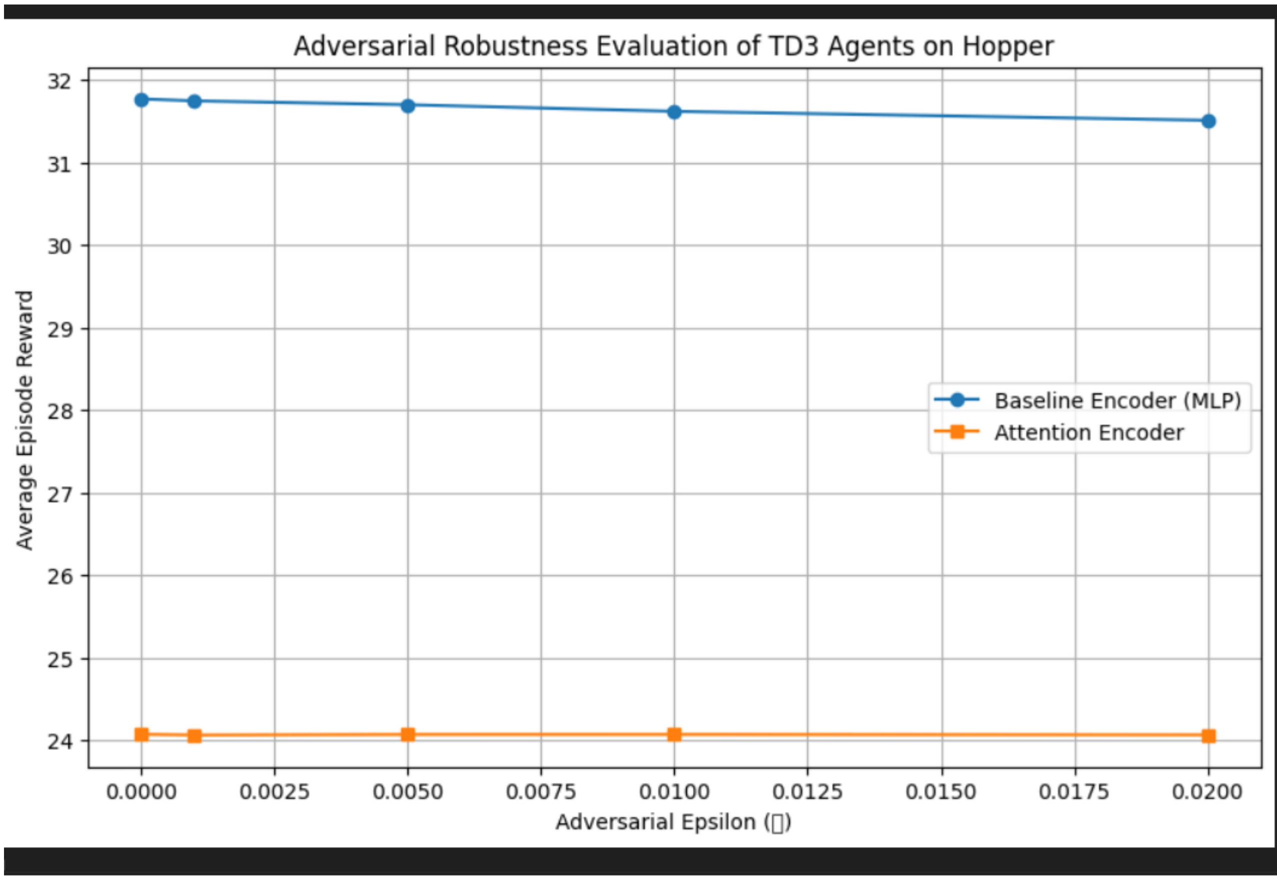
Results- From the plot, we see two lines representing average episode rewards under varying adversarial perturbation strengths (\epsilon) in the Hopper environment:

- **Blue line (Baseline Encoder, MLP):** Stays around 31 reward
- **Orange line (Attention Encoder):** Stays around 24 reward

Despite increasing the adversarial strength from 0.0 to 0.02, both lines remain nearly flat, indicating that **neither agent experiences a drastic drop in performance** due to adversarial perturbations. However, the absolute reward values differ significantly:

1. **Baseline MLP Encoder (Blue)**
   - Maintains a higher reward (around 31) across all tested \epsilon values.
   - This suggests the MLP-based actor already has a good policy for Hopper, and the FGSM attack in this range does not meaningfully degrade its performance.
2. **Attention Encoder (Orange)**
   - Remains around 24 reward, consistently lower than the baseline.
   - While it does not decline much with \epsilon, it starts from a lower performance level, indicating that in this particular setup (architecture, hyper parameters, training conditions), the attention-based actor did not learn a policy as strong as the baseline MLP.

Key Observations



- **Robustness to Adversarial Noise:** Both agents appear robust in the sense that their performance does not sharply decline with increasing \epsilon. The nearly flat lines imply that small adversarial perturbations do not dramatically affect either actor's decisions in Hopper.
- **Absolute Performance Gap:** The baseline MLP agent retains a consistently higher reward than the attention-based agent, suggesting that in this specific environment and training configuration, the simpler MLP architecture happens to learn a better overall policy.

Therefore, while **both agents demonstrate similar resilience** (little performance drop) under adversarial noise in the tested \epsilon range, the **baseline MLP agent** ends up with **higher absolute rewards**, and the **attention-based agent** settles at a lower reward plateau.

Why did I select FGSM to test the robustness of these settings?

There are two main reasons for these choices:

- **Simplicity and Speed:** FGSM is one of the simplest gradient-based adversarial attack methods. It requires only a single gradient computation (as opposed to multi-step methods like Projected Gradient Descent), making it relatively fast and straightforward to implement.
- **Baseline for Adversarial Attacks:** Because of its simplicity, FGSM is often used as a first check on how susceptible a model is to adversarial perturbations. It provides an easily interpretable baseline: if a model is vulnerable to this single-step attack, it may also be vulnerable to more sophisticated methods.

Sources- 1) ***"Explaining and Harnessing Adversarial Examples,"*** Goodfellow and colleagues introduced FGSM as a fast and effective method for generating adversarial examples. Although originally presented in the context of supervised learning, FGSM has since been adapted and applied to reinforcement learning, where it helps test the robustness of Q-learning agents by perturbing input states.