Robust Policy Learning for Transfer Learning in Robots*

Rishhanth Maanav V Siddharth Nayak

1 Introduction

Reinforcement Learning agents have been used for a wide range of tasks, such as games and robotic control. The agent tries to learn policies and value functions through trial and error by interacting with the environment until it converges to an optimal policy. Robustness and stability are quite critical in Reinforcement Learning; however, neural networks are vulnerable to noise from unexpected sources and are not likely to withstand disturbances. Also, whenever robots are trained with Reinforcement Learning agents, they are generally either initialized with hand-made policies or policies from simulated agents. The simulated agents, however may not capture all of the noise in a real-world robot. Thus, the transfer learning may not be efficient enough for faster convergence on robots. In this paper we evaluate two robust reinforcement learning algorithms and compare them along with the vanilla algorithm and a naive algorithm described further in the paper. The main idea in this paper is that we can use adversarial examples to choose perturbations during the training of the agent. In this paper we try to implement the robust reinforcement learning algorithms simulated in OpenAI Gym and MujoCo Environments with perturbations in the observations. We use the Reacher-v2 environment of OpenAI Gym with MujoCo backend for our evaluation.

2 Markov Decision Process Formulation for Reacher

Reacher is an environment in OpenAI Gym with MujoCo backend. The reacher is basically a two linked robotic arm where each of the link in the arm has a motor to control it's rotation. The main goal of the agent is to make the robotic arm reach a ball which is present in the environment. The state vector for Reacher is 11-dimensional. The state vector for the agent is as follows: $\vec{s} \in \chi \subseteq \mathbb{R}^1 1$

- cosine of angle of link-1 $cos\theta_1 \in [-1, 1]$
- cosine of angle of link-2 $cos\theta_2 \in [-1, 1]$
- sine of angle of link-1 $sin\theta_1 \in [-1, 1]$
- sine of angle of link-2 $sin\theta_2 \in [-1, 1]$
- angular velocity of link-1 $\dot{\theta_1} \in \mathbb{R}$
- angular velocity of link-2 $\dot{\theta_2} \in \mathbb{R}$
- position of the ball (target) $\vec{x}_{ball} \in \mathbb{R}^2$
- vector joining the ball and the tip of the arm $\vec{x}_{ball} - \vec{x}_{fingertip} \in \mathbb{R}^3$ the third dimension is redundant here because it is always zero.

The actions space is 2-dimensional. The two actions to be taken at each time step are: $\vec{a} \in A \subseteq \mathbb{R}^2$

- Applying torque-1 on motor-1 $\rightarrow a_1 \in \mathbb{R}$
- Applying torque-2 on motor-2 $\rightarrow a_2 \in \mathbb{R}$

Once an agent takes an action, the state transitioned to is a deterministic function ($f : \chi \mathrm{x} A \rightarrow \chi$) of the present state and the action. This transfer function is captured by the predefined dynamics of the system in

*Equal Contribution by both Authors

MujoCo physics simulator. Note that the torque applied can be continuous in values and thus the agent has to apply values of torque which are close to the most optimal value.
The reward (R) for the agent at each time-step is:

$$R_{dist} = -\|\vec{x}_{fingertip} - \vec{x}_{ball}\|_2$$

where $\vec{x}_{fingertip}$ and $\vec{x}_{ball}$ are the co-ordinates of the fingertip and the ball respectively.

$$R_{action} = -\left(a_1^2 + a_2^2\right)$$

Here R_{action} penalizes the agent if it applies very large torques to the motors. This also signifies the amount of effort the agent is putting in to control the robot.

$$R = R_{dist} + R_{action}$$

The episode is terminated after 50 time-steps which is a default setting given by OpenAI Gym / MujoCo. The best possible average reward for an agent, for 50 time-steps per episode is -3.75.
A policy determined by the agent is a probability distribution over the continuous action space.

3 Algorithm

We have implemented:

- Naive Robust Reinforcement Learning (NRL)
- Adversarially Robust Reinforcement Learning (ARPL)[1]
- Robust Adversarial Reinforcement Learning (RARL)[2]

We used Trust Region Policy Optimisation (TRPO)[3] combined with the above three methods, for training the agent.

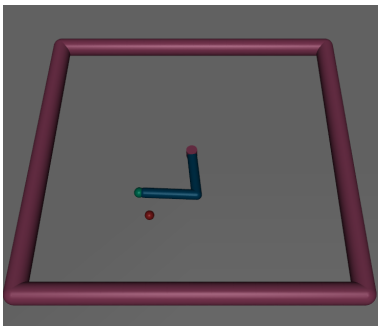

Figure 1: The Reacher Environment

3.1 Trust Region Policy Optimisation

TRPO is a policy gradient method which improves the stability and convergence of other general policy gradient and actor critic algorithms. TRPO modifies the unconstrained optimisation problem in policy gradient methods to a constrained optimisation problem as described below,

$$\underset{\theta}{maximize}\ \widehat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\widehat{A}_t\right]$$

$$\text{subject to } \widehat{\mathbb{E}}_t\Big[KL\big[\pi_{\theta_{old}}(.|s_t), \pi_\theta(.|s_t)\big]\Big] \leq \delta$$

Note that the expectation is taken over the trajectory sampled by the policy π_{old}

$\pi_\theta(a_t|s_t)$ policy parametrised by theta (network) ; a_t is the action taken at time 't' ; s_t is the state at time 't';
$\widehat{A}_t$is the estimated "generalized" advantage at time 't' where $\widehat{A}_t = Q(s_t, a_t) - V_\theta(s_t) + \tau\gamma\widehat{A}_{t+1}$;
with γ as the discount factor and τ as the generalization factor
Q is the return from a state obtained when a trajectory is sampled by $\pi; Q(s, a) = r + \gamma V_\theta(\overline{s}')$

In Actor-Critic models, the value function formulation of Q is used to generate an approximate return which is the sum of the current reward and the discounted expectation of the parametrised value of the next state. If generalisation is not used, the advantage is the same as the TD error.
V_θ is the value function of a state which is parametrised by the critic network ;
δ is a hyperparameter ; KL stands for the Kullback-Leibler divergence

The maximization objective function has the ratio $\frac{\pi_\theta}{\pi_{\theta_{old}}}$ as the importance weight (from importance sampling). This is included because the maximization has to be done on sample trajectories generated by the policy parametrised by the new parameters θ whereas we sample only using the older policy $\pi_{\theta_{old}}$. Optimising this objective without any constraint is prone to numerical instability. Thus, TRPO makes sure that the policy is not moving too far from the current parameter θ_{old} by constraining the KL divergence to be bounded by δ as denoted above. For notational convenience,

$$L_{\pi_{\theta_{old}}}(\pi_\theta) = \widehat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \widehat{A}_t \right]$$

The constraint optimization problem is now the equivalent of solving,
$\underset{\theta}{maximize}\, L_{\pi_{\theta_{old}}}(\pi_\theta) - \beta.\overline{KL}_{\pi_{\theta_{old}}}(\pi_\theta)$;where β is a Lagrange Multiplier.

Algorithm 1 TRPO

described here is one step of the TRPO update rule :

$\underset{\theta}{maximize}\, L_{\pi_{\theta_{old}}}(\pi_\theta) - \beta.\overline{KL}_{\pi_{\theta_{old}}}(\pi_\theta)$

done by approximating L linearly near θ_{old} and KL quadratically near θ_{old}

$\underset{\theta}{maximize}\, g.(\theta - \theta_{old}) - \frac{\beta}{2}(\theta - \theta_{old})^T.F.(\theta - \theta_{old})$

where $g = \frac{\partial}{\partial\theta} L_{\pi_{\theta_{old}}}(\pi_\theta)\Big|_{\theta=\theta_{old}}$ and $F = \frac{\partial^2}{\partial^2\theta} \overline{KL}_{\pi_{\theta_{old}}}(\pi_\theta)\Big|_{\theta=\theta_{old}}$

Quadratic part of L is negligible compared to KL term.

F is positive semidefinite

$\theta \leftarrow \theta_{old} + \frac{1}{\beta} F^{-1} g$

The matrix F which was described above is the Fisher Information matrix and g is the policy gradient. The update step $(\theta - \theta_{old})$ is the natural policy gradient.

The computation of the Fisher Information matrix (F) is expensive. It requires a second-order oracle to compute the hessian of the KL divergence term. Also, the parameter space is very high dimensional. Thus, optimizing the policy using the above described method is computationally expensive. So, we use conjugate gradient method to approximately obtain the update step without explicitly forming the Fisher Matrix F.

TRPO uses Generalized Advantage Estimation (GAE) for estimating advantages. These estimates require the future returns for computing the advantages. Thus, TRPO updates are done after the episodes are sampled completely in a fashion similar to Monte-Carlo methods.

3.2 Adversarial Robust Policy Learning (ARPL)

ARPL, Mandlekar et al., is an algorithm to generate additive adversarial perturbations to states while training policy gradient algorithms on simulators. These perturbations generated by ARPL aim to capture the

noise in the dynamics noise along with the process noise of the system. Dynamics noise is the error in measurement of physical quantities like mass, friction, etc. The dynamics noise influence the state, but only through a function which captures the dynamics of the system. Process noise is the noise which adds to the state directly.

ARPL uses a gradient based perturbation technique for adding adversarial perturbations to the states. There have been works in gradient based perturbations in the past. One of the most important contributions to adversarial perturbations is that of Fast Gradient Sign Method (FGSM) from Goodfellow et al. FGSM focuses on adversarial perturbations of an image input to a deep learning model where the change in each pixel is determined by the sign of the gradient over a defined loss function η and limited by a factor ε. The perturbations generated by FGSM could be extended to policy gradient algorithms where it is formulated as,

$$\delta = \varepsilon sign(\nabla_s \eta(\pi_\theta)) \qquad \text{(FGSM)}$$

where η is a loss function defined over the policy π parametrized by the policy network parameters θ and ε is a hyperparameter denoting the strength of the perturbations

This idea of FGSM is more suited for images while training deep learning models. ARPL extends this idea of gradient based perturbations of states to policy gradient algorithms.

In ARPL the perturbations of the states are computed as,

$$\delta = \varepsilon \nabla_s \eta(\pi_\theta) \qquad \text{(ARPL)}$$

where ε is a hyperparameter indicating the strength of the perturbation,η is a loss function over the policy π for perturbations. Note that this δ is not the same as that in TRPO. The policy π is parametrised by the policy network parameters θ

This loss function defined over the policy must be such that the added gradient perturbations to the states generally encourage the agent to take some corrective measures in its actions considering the noise present in the system. Note, that this loss function is different from the policy gradient loss and the value network loss. We'll describe the specific loss function which we use for our setting in the Experiments and Hyperparameters section.At each timestep in an episode, we perturb the states using ARPL with a probability ϕ which is a hyperparameter.

Algorithm 2 Adversarially Robust Policy Learning (ARPL)

Data: hyperparameters:ϕ, ϵ, k
Initialize θ_0
for $i = 0, 1, 2, ..., max\_iter$ **do**
    // Perform k-Batch Rollout(s) with Adv. Perturb.
    for each $t = 1, ..., Tk, \forall k$ **do**
        Sample trajectory with policy $\pi(\theta_i)$: $\tau_{ik} = \{s_t, a_t\}_{t=0}^{T_{k-1}}$
        Compute adv. perturb. $\delta = \epsilon\big(\nabla_{s_t} \eta(\pi_{\theta_i})\big)$
        add perturbation(δ) with Bernoulli probability (ϕ)
    // Batch Update to Policy Network
    $\theta_{i+1} \leftarrow policy\_grad(\theta_i, \{\tau_{i1}, ...\tau_{ik}\})$ here the update step follows from TRPO
    update params of value network using TD error
Result: Robust Policy π

3.3 Robust Adversarial Reinforcement Learning (RARL)

In RARL we have two agents: Protagonist: the one who tries to improve the performance and Adversary: the one who tries to degrade the performance. Both these agents play a zero-sum game. This adversary training helps the protagonist agent to explore more severe situations and thereby making it more adaptable to noisy environments.Both the protagonist and the adversary receive feedback from the environment and both give an action based on the observation. The difference is that the adversary takes an action that tries to fail the protagonist in it's task and make it get a lower reward. This is quite similar to the method used by Goodfellow et al[6] in Generative Adversarial Network(GAN) where two networks:Generator network and Discriminator Network compete with each other in a zero-sum game.

In RARL, there are 2 policy networks, one for the protagonist and the other for the adversary generating the policies π_p and π_a respectively. Similarly, there are two values networks for predicting the value function for the protagonist and the adversary V_p and V_a respectively. The action taken to sample the trajectory now is influenced by both the protagonist and the adversary. The exact mathematical formulation is described in the upcoming section.

3.3.1 The update rule

The Protagonist Policy π_p and the Adversary Policy π_a both participate in a zero-sum game. The actions of the protagonist and the adversary both affect the next state. The action taken by the algorithm to sample a trajectory is given as,

$$a = a_{protagonist} + D * a_{adversary}$$

where D is a scalar hyperparameter called the difficulty level. Here this parameter 'D' is set to confine the magnitude of the action from the adversarial agent. This is to ensure that the learning does not become unstable.

The expected reward $Q(s, a)$ is the cumulative discounted reward.

$$Q(s_k, a_k) = \mathbb{E}\left(\sum_{t=k}^{T} \gamma^{t-k} * \big(r(s_t, a_t)\big)\right) = \mathbb{E}\left(r + \gamma * V(\overline{s})\right)$$

The advantage function $A(s, a)$ is defined as the difference between the expected and the estimated reward: $A(s, a) = Q(s, a) - V(s)$. In the zero-sum game we have $r_{protagonist} = -r_{adversary} = r$. The update of the parameters are as follows:

$$\theta_p \leftarrow \theta_p + \alpha * A_{protagonist} * \nabla_{\theta_p} log(\pi_p)$$

$$\theta_a \leftarrow \theta_a + \alpha * A_{adversary} * \nabla_{\theta_a} log(\pi_a)$$

Here $A_{protagonist}$ is the difference between the discounted return ($Q(s, a)$) and the value network prediction ($V_{protagonist}(s)$). The discounted return is approximated by the sum of the current reward from the trajectory and the discounted expectation of the value of the next state predicted by the value network. The same applies to the adversary also but here the rewards are negated and hence the discounted return is also negated. For TRPO the advantages are generalized estimates and not the vanilla advantage where $A_t^{gen} = A_t + \gamma * \tau * A_{t+1}^{gen}$ with γ as the discount factor, τ as the generalization factor (GAE factor) and A_t is the vanilla Advantage at time t.

While testing, only the protagonist policy is used to sample actions.

Algorithm 3 Robust Adversarial Reinforcement Learning (RARL)

Data: hyperparameters:D
Initialize θ_p and θ_a for the policy networks
Initialize w_p and w_a for the value networks
for $i = 0, 1, 2, ..., max\_iter$ **do**
    perform k-batch rollouts
    for $t = 1, 2, 3...max\_timesteps$
        take action a = $a_{protagonist} + D * a_{adversary}$
    Update params θ_p and θ_a using TRPO update rule with rewards as R and -R respectively.
    Update params w_p and w_a with the Temporal Difference error at every step

3.4 Naive Algorithm and the Vanilla model

In the Vanilla model we do not perturb the states while training. In essence the agent does not know about the anything about the perturbations. In the Naive Algorithm, while training we perturb the states of the agent randomly as:

$$s_t \leftarrow s_t + \alpha * x$$

Here: α is a hyperparameter which we set to 0.1 while training and $x \sim \mathcal{N}(0, 1)$ is a random number sampled from a Gaussian distribution with zero mean and standard deviation equal to one.
Note that the algorithms ARPL, RARL, and Naive algorithms serve as additives to the vanilla TRPO Algorithm. Thus, they can be extended to any other Policy Gradient algorithm.

4 Experiments and Hyperparameters

For all our experiments we used TRPO as the basic algorithm on top of which we built various robust algorithms (ARPL, RARL, and Naive). TRPO is an Actor Critic algorithm consisting of 2 networks, one parametrising the policy and the other parametrising the value. The action space is continuous. We assume that the distribution of actions in the policy is Gaussian. The policy network in our case predicts the mean and the standard deviation of the Gaussian. While sampling actions from a policy these Gaussian distributions with mean and standard deviation predicted by the network.The policy and value network remain the same across all experiments with configurations as described below.

In the states of Reacher, instead of just the angles, the cosine and sine of the angles are specified. This is to avoid the discontinuity which would have otherwise occured in trying to represent the angles in their principle domain $[0, 2\pi]$, ie. $\theta(mod 2\pi)$.

4.1 Policy Network (Actor)

The input to the policy network is the state vector 11-dimensional. The action space is 2-dimensional, so the policy network must predict 2 means and standard deviations. The policy network has the configuration:

- Input Layer Dimension: 11
- Hidden Layer 1 Dimension: 64
- Hidden Layer 2 Dimension: 64
- Output Layer Dimension: 2

The standard deviations are independent of the state (act similar to bias terms) and are learnable parameters. We used $tanh$ activation at all hidden layers.

4.2 Value Network (Critic)

The input to the value network is the state vector 11-dimensional. The value network has the configuration:

- Input Layer Dimension: 11
- Hidden Layer 1 Dimension: 64
- Hidden Layer 2 Dimension: 64
- Output Layer Dimension: 1

We used $tanh$ activation at all hidden layers of the value network.

4.3 Common Hyperparameters

We used the following hyperparameters for training all our agents,

- Discount Factor γ = 0.995
- Generalisation Factor for GAE in TRPO τ = 0.97
- L2-Regularisation rate λ = 0.001
- KL divergence constraint bound for TRPO δ = 0.01
- Damping coefficient for Conjugate Gradient method = 0.1

At every training iteration we rollout 300 sample trajectories using the policy at that iteration in order to have 300*50=15000 samples of (state,action,reward tuples). We train each agent for 1000 iterations.

4.4 ARPL

In the ARPL algorithm, we train the agent while perturbing the states. The perturbations are gradient-based as seen before.

$$\delta = \varepsilon \nabla_s \eta(\pi_\theta)$$

For training our ARPL agent, we use $\eta(\pi_{theta})$, as the squared L2-norm of the mean vector predicted by the policy network, ie. if $\mu = [\mu_1, \mu_2]$ is the mean vector predicted by the policy network, we use $\|\mu\|_2^2$ as the ARPL loss function. The intuition behind this loss is that, the state perturbed by the positive gradient of this loss will result in a higher value of the mean predicted by the policy network. Hence, by perturbing the state by this method, we trick the agent to believe that the perturbed state gave the same reward as the previous unperturbed state with the same action. Thus, the agent is encouraged to apply higher valued actions.

We use $\varepsilon = 0.1$ as the perturbation strength. Also, we used a curriculum learning approach to increase ϕ, the probability of adding perturbations. We varied ϕ from 0.0 to 0.1 in 1000 iterations, increasing it every 100 iterations. This is done to maintain stability while training.

For training the naive algorithm we used a constant probability $\phi = 1.0$. Here the strength of perturbation was $\varepsilon = 0.1$.

4.5 RARL

In RARL, both the protagonist and the adversary have the same policy and value network configuration (the same discussed above). The difficulty level 'D' was set at 0.1 for stability while training. To improve stability, we train only the protagonist for the first 50 iterations. The protagonist and the adversary are then trained together at every iteration starting from the 50th iteration.

5 Results

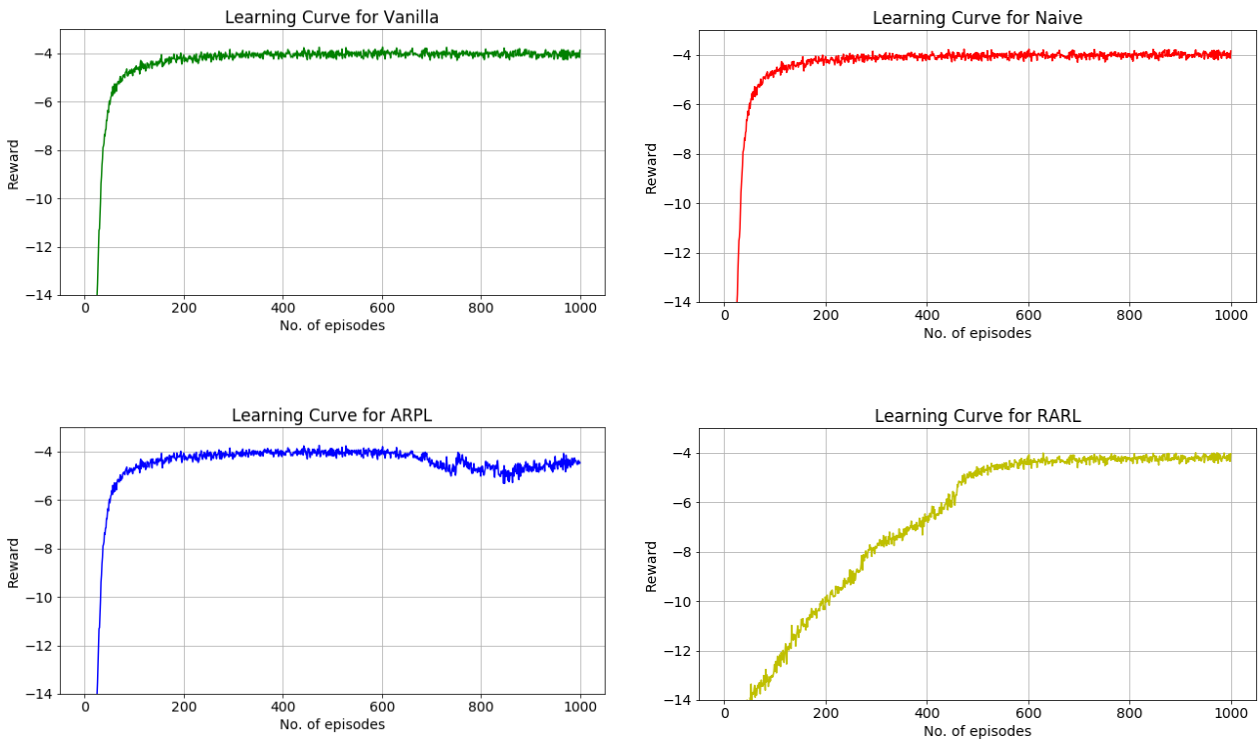

Figure 2: The Learning curves for the four different methods. For ARPL, the rewards saturate after 200 episodes and then after 600 episodes it decreases slightly. This is because of the curriculum learning method used while training. i.e. the frequency of perturbations has increased. But the reward does increase again after 900 episodes. In RARL the saturation of the reward is slower than all other three algorithms because of the presence of the adversary agent even in the initial stages. Also the Naive and the Vanilla network almost have the same learning curve.

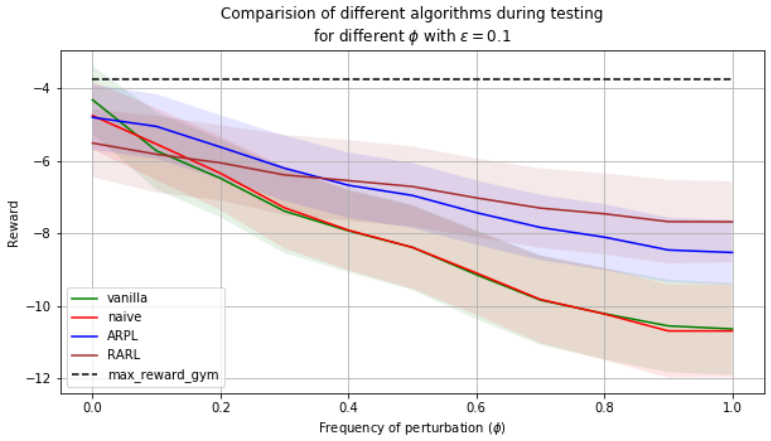

Figure 3: Performance of agents with $\varepsilon = 0.1$ (test time)

Average rewards for 1000 episodes for $\varepsilon = 0.1$ while testing				
ϕ	Vanilla	Naive	ARPL	RARL
0	-4.31 ± 1.85	-4.75 ± 1.87	-4.79 ± 1.79	-5.50 ± 1.87
0.1	-5.72 ± 2.13	-5.53 ± 1.97	-5.04 ± 1.77	-5.81 ± 2.10
0.2	-6.47 ± 2.13	-6.34 ± 2.01	-5.61 ± 1.74	-6.05 ± 2.06
0.3	-7.38 ± 2.31	-7.29 ± 2.24	-6.19 ± 1.79	-6.38 ± 2.20
0.4	-7.92 ± 2.27	-7.90 ± 2.19	-6.67 ± 1.80	-6.54 ± 2.22
0.5	-8.38 ± 2.34	-8.38 ± 2.28	-6.95 ± 1.78	-6.70 ± 2.20
0.6	-9.13 ± 2.44	-9.08 ± 2.30	-7.42 ± 1.76	-7.01 ± 2.16
0.7	-9.83 ± 2.46	-9.81 ± 2.40	-7.83 ± 1.79	-7.29 ± 2.18
0.8	-10.21 ± 2.52	-10.21 ± 2.52	-8.10 ± 1.81	-7.45 ± 2.22
0.9	-10.54 ± 2.54	-10.68 ± 2.57	-8.45 ± 1.78	-7.67 ± 2.29
1.0	-10.63 ± 2.53	-10.68 ± 2.55	-8.52 ± 1.78	-7.67 ± 2.22

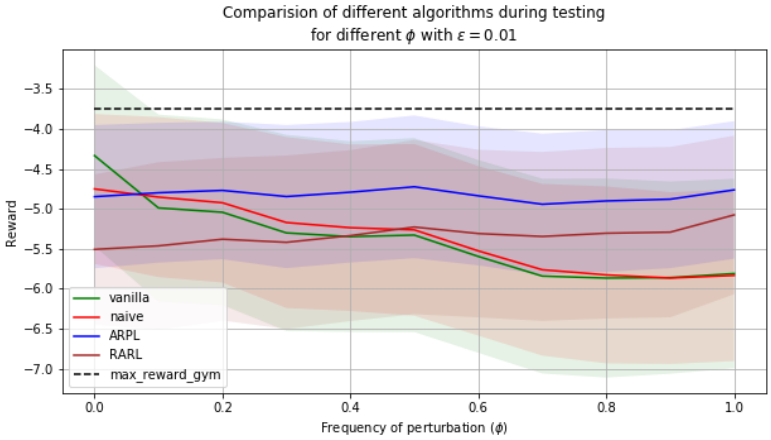

Figure 4: Performance of agents with $\varepsilon = 0.01$ (test time)

Average rewards for 1000 episodes for $\varepsilon = 0.01$ while testing				
ϕ	Vanilla	Naive	ARPL	RARL
0	-4.33 ± 2.25	-4.75 ± 1.87	-4.84 ± 1.79	-5.50 ± 1.87
0.1	-4.98 ± 2.33	-4.85 ± 1.99	-4.79 ± 1.74	-5.46 ± 2.08
0.2	-5.04 ± 2.31	-4.92 ± 1.99	-4.76 ± 1.71	-5.37 ± 2.03
0.3	-5.30 ± 2.45	-5.17 ± 2.13	-4.84 ± 1.78	-5.41 ± 2.16
0.4	-5.34 ± 2.38	-5.23 ± 2.08	-4.79 ± 1.75	-5.33 ± 2.13
0.5	-5.32 ± 2.42	-5.26 ± 2.14	-4.72 ± 1.78	-5.22 ± 2.17
0.6	-5.59 ± 2.40	-5.52 ± 2.15	-4.83 ± 1.73	-5.30 ± 2.09
0.7	-5.84 ± 2.43	-5.76 ± 2.14	-4.94 ± 1.76	-5.34 ± 2.10
0.8	-5.86 ± 2.48	-5.82 ± 2.20	-4.90 ± 1.77	-5.30 ± 2.12
0.9	-5.85 ± 2.40	-5.86 ± 2.14	-4.87 ± 1.72	-5.29 ± 2.12
1.0	-5.80 ± 2.36	-5.83 ± 2.13	-4.76 ± 1.71	-5.07 ± 1.97

We evaluated the performances of Vanilla, Naive, ARPL and RARL with different frequencies of perturbations (or probabilities of perturbing at each time step) and two different strengths of perturbations (0.1 0.01). While evaluation, we perturb the states in the MujoCo Environment as follows:

$$s_t = s_t + \varepsilon * x$$

with Bernoulli Probability ϕ, where $x \sim \mathcal{N}(0, 1)$, at each time step. Here ε represents the strengths of perturbations in the state and ϕ represents the probability with which the perturbations are added. These experiments were run on 1000 episodes whose trajectories were sampled using the policy (from the policy network) obtained after training each of the 4 agents. The mean and standard deviation of the total reward obtained were computed for these 1000 episodes. These have been presented in the tabular form previously along with the corresponding plots.

6 Conclusions

Based on the empirical results obtained from the evaluation of the different robust algorithms on the Reacher environment, we can draw some conclusions on their performance.

6.1 Perturbations with $\varepsilon = 0.1$

As seen in Fig 3, at lower frequencies of perturbations all four algorithms perform comparably well. As the frequency of perturbation is increased, the Vanilla and the Naive algorithm degrade in their performances. Whereas, ARPL and RARL degrade much lesser compared to Naive and Vanilla. This shows that the ARPL and RARL are quite robust to noise when compared to Vanilla and Naive. Another peculiar thing about the graph is that ARPL performs slightly better than RARL at smaller frequencies, but RARL is better at higher frequencies of perturbation. This is because while training RARL always has perturbations (adversary agent) while training whereas in ARPL we have curriculum learning, where the frequency of perturbations keeps on increasing with number of episodes while training. When there is no perturbation, RARL performs worse than others. This is because, RARL gets higher penalty (higher negative reward) for taking unnecessary actions to counter the non-existent perturbations. Also, ARPL does better than RARL in this case because the ARPL agent has perturbations included in the state while the RARL agent has the adversary perturbing its action while training. Thus, the RARL agent assumes the presence of the adversarial agent at all time instances and overcompensates for its effect even when it does not exist.

6.2 Perturbations with $\varepsilon = 0.01$

As seen in Fig 4, at lower frequencies we have almost comparable performances in all four algorithms. As the frequency is increased the Vanilla and the Naive algorithm degrade in their performance. In ARPL the performance is almost constant with increasing frequency in perturbation. RARL increases slightly in performance as the frequency of perturbation is increased. This serves as an evidence for the robustness of RARL and ARPL. However, RARL performs worse than the other 3 algorithms when there is no perturbation due to reasons seen in the previous subsection. In this case the strength of perturbation ε is only one-tenth of the previous case. Hence, the performance of all 4 algorithms is generally better in this case than the previous case.

As a final conclusion, both ARPL and RARL are robust to noise when evaluated with perturbations/noise in the states. RARL works well (compared to ARPL) when we know that frequency of noise occurence and the strength of noise in the system is high.But ARPL works better (when compared to RARL) when we know that the frequency of noise occurrence and its strength in the system is low.

7 Future Work

Future work includes working on a model where we combine both the ARPL and RARL. In this model we perturb the state, as done in ARPL and perturb the action with an adversary agent, as done in RARL. This kind of dual perturbation may include the benefits of both ARPL and RARL.

8 References

[1] A. Mandlekar, Y. Zhu, A. Garg, Li Fei, S Savarese. *Adversarially Robust Policy Learning: Active Construction of Physically-Plausible Perturbations*. 2017

[2] L. Pinto, J. Davidson, R. Sukthankar, A. Gupta. *Robust Adversarial Reinforcement Learning*. 2017

[3] J. Schulman, S. Levine, P Moritz, M. I. Jordan, P Abbeel. *Trust Region Policy Optimization*. 2014

[4]V. R. Konda and J. N. Tsitsiklis. *Actor-Critic Algorithms*. 2000

[5] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. *Intriguing properties of neural networks*, Dec. 21, 2013. arXiv: 1312.6199v4 [cs.CV].

[6] I. J. Goodfellow, J. P. Abadie, M. Mirza, B. Xu, D. W. Farley, S. Ozair, A. Courville, Y. Bengio. *Generative Adversarial Networks*. 2014

[8] D. Bertsekas, J. N. Tsitsiklis. *Neuro Dynamic Programming, 1st Edition*. 1996

[9] R. Sutton, A. Barto. *Introduction to Reinforcement Learning, 2nd Edition*. 2018

9 Acknowledgements

We implemented different algorithms on top of TRPO, the vanilla code for which was released by Ilya Kostrikov through his GitHub handle. The link for the same is https://github.com/ikostrikov/pytorch-trpo. This implementation has been done using PyTorch.