Report on Reinforcement Learning, Sim-to-Real techniques and Forward-Forward

Alessandro Arneodo
s331361
Politecnico di Torino
s331361@studenti.polito.it

Daniele Ercole
s328755
Politecnico di Torino
s328755@studenti.polito.it

Davide Fassio
s323584
Politecnico di Torino
s323584@studenti.polito.it

Abstract

This report investigates Reinforcement Learning for robotic systems, focusing on the sim-to-real transfer problem in the OpenAI Gym Hopper environment. Control policies for a simulated robot are trained using some basic and advanced algorithms such as REINFORCE, Actor-Critic and Soft Actor-Critic (SAC). Uniform Domain Randomization (UDR) is then employed to enhance policy robustness to changes and uncertainties in the environment. In the project extension it is demonstrated the feasibility of Forward-Forward in the Reinforcement Learning framework, testing it in the OpenAI Gym Cart Pole environment. Code available at: https://github.com/DSLproject/MLDL_2024_RL

1. Introduction

Reinforcement learning (RL) has emerged as a powerful paradigm in artificial intelligence, allowing agents to learn optimal behaviors through interactions with their environment. Using reward signals to drive learning, RL has achieved significant milestones in areas ranging from game playing to robotic control. Training the agent in a simulated environment offers several advantages like cost, speed and safety. However, one of the persistent challenges in deploying RL in real-world scenarios is the sim-to-real gap, which refers to the discrepancies between simulated environments and the complexities of the real world. Addressing this gap is crucial for ensuring that policies learned in simulation can be effectively transferred and applied robustly in practical applications.

The sim-to-real gap arises due to differences in dynamics, sensor noise, and unmodeled environmental factors that are often not fully captured in simulations. This challenge requires innovative approaches to improve the generalization and adaptability of RL agents. Techniques such as domain randomization, curriculum learning, physics-based simulations and real world fine tuning are actively explored to bridge this gap.

In parallel, recent advancements in the field of deep learning have introduced novel algorithms that could potentially enhance RL frameworks. Forward-Forward, proposed by Hinton in 2022 [1], addresses some typical issues of backpropagation such as efficiency on analog devices, the frequent need to interrupt inference for training, and the biological implausibility of replicating it.

This report details the progress and findings of our project, which involves the implementation and evaluation of various algorithms and techniques within the RL framework. Additionally, it includes initial research on Forward-Forward, a novel approach distinct from the traditional backpropagation.

2. Related work

Reinforcement Learning (RL) is a branch of machine learning where an agent learns to make decisions by interacting with an environment to maximize cumulative rewards. Key components of RL include the agent, environment, states, actions, and rewards. Traditional RL algorithms primarily utilize value-based methods, which estimate action values to guide decision-making.

Our project explores various algorithms employing different architectures. Initially, we used the REINFORCE algorithm, introduced by Ronald J. Williams in 1992 [2], as a foundational method in policy gradient techniques. REINFORCE optimizes the agent's policy directly by computing gradients of expected rewards, advancing RL by providing a framework for optimizing policies in complex, uncertain environments.

Subsequently, we employed Actor-Critic methods to combine the strengths of value-based and policy-based approaches [3]. Here, the actor updates the policy directly, while the critic estimates the value function to provide feedback, reducing policy gradient variance and enhancing stability and convergence.

Building on these methods, we utilized SAC, a state-of-the-art algorithm introduced by Haarnoja et al. [4]. in 2018. SAC incorporates entropy regularization to promote exploration and enhance stability in off-policy learning.

To make SAC perform better and bridge the sim-to-real gap, we used Uniform Domain Randomization [5], which makes RL agents more robust by randomly changing environment settings during training.

In our research we employed the OpenAI Gym toolkit [6]. Specifically, we used the Hopper environment from the MuJoCo suite to simulate a robot with 3 degrees of freedom. For preliminary investigations, we also utilized the simpler Cart Pole environment.

In the last part of this work, we tested the Forward-Forward algorithm on the CartPole environment to evaluate its effectiveness. This paradigm, proposed in Hinton et al. [1], suggests a novel learning approach that substitutes backpropagation with two forward passes using positive and negative data.

Our implementation of Forward-Forward drew inspiration from Nebuly's work [7], which provides an implementation on CIFAR-10, guiding our approach in this study.

3. Methodology

This work focuses on the exploration of RL algorithms in the context of sim-to-real transfer for robotic systems and the following are the key phases:

- Basic RL algorithm: implement REINFORCE and Actor-Critic algorithms to establish initial benchmarks for training control policies in the Gym Hopper environment.
- Advanced RL algorithm: integrate advanced RL algorithms like SAC to establish performance baselines and compare effectiveness across simulated environments.
- Uniform Domain Randomization: implement UDR techniques to systematically vary dynamics parameters during training, enhancing policy robustness against environment variations.

As an extension of the project, we established the mathematical foundations and subsequently implemented and demonstrated the effectiveness of the Forward-Forward algorithm on the Cart Pole environment.

3.1. OpenAI Gym Hopper environment

The Hopper is an OpenAI Gym environment based on the MuJoCo physics engine (Multi-Joint dynamics with Contact). The hopper is a two-dimensional one-legged figure composed of four body parts: torso, thigh, leg and foot. The goal is to make hops that move in the forward (right) direction while maintaining its balance and upright posture. The observation space is a 11D vector representing the height, joint angles, linear velocity and angular joint velocities. The action space is a 3D vector representing the torques applied to each joint. The agent obtain rewards for going forward and for each timestep it keeps the balance.

3.2. REINFORCE

The first basic RL training algorithm implemented is the REINFORCE algorithm, a policy gradient method employed to train an agent on the Custom Hopper environment. This method is particularly suited for continuous action spaces and involves learning a stochastic policy that directly maps states to a probability distribution over actions.

3.2.1 Network architecture

The policy network, parameterized by θ, is modeled as a neural network that takes the state s_t as input and outputs a probability distribution over actions a_t. The network architecture includes an input layer (that corresponds to the state space dimensions of the environment), 2 hidden layers (two fully connected layers with 64 neurons each, activated by the tanh function) and an output layer (that corresponds to the action space dimensions). Additionally, we use a learned parameter to model the standard deviation of the action distribution, which is activated by the softplus function to ensure positivity. The output thus represents a Gaussian distribution from which actions can be sampled.

3.2.2 Weights update

To train the policy network, we first compute the discounted sum of future rewards, known as the return G_t, from each time step t: $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ where γ is the discount factor.

After that, we can proceed with the policy gradient update. The objective is to maximize the expected return by adjusting the policy parameters θ. The gradient of this objective is estimated using the following equation:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot G_t \right] \tag{1}$$

and the policy loss is computed as

$$L(\theta) = -\log \pi_\theta(a_t|s_t) \cdot (G_t - b_t) \tag{2}$$

Here, b_t is a baseline subtracted from the return to reduce variance without introducing bias. In our implementation, we used a constant baseline: first it was set to 0 and then was set to 20. The optimizer used for updating the policy network parameters is Adam, which is well-suited for dealing with noisy gradients and adaptive learning rates.

3.2.3 Training process

The REINFORCE algorithm was trained without performing a hyperparameter grid search and the following fixed values are used: learning rate = 1e-3 and γ (discount factor) = 0.99. The training process was carried out in the source environment for 100K episodes, both with and without baseline, totaling 5 different runs for each configuration.

3.3. Actor-Critic

The second basic RL algorithm implemented is the Actor-Critic (AC) algorithm, a well-suited technique for continuous action spaces that combines elements of both policy-based (Actor) and value-based (Critic) methods to improve training efficiency and stability.

3.3.1 Network architecture

The Actor-Critic algorithm is based on two different architectures: Actor and Critic network. The actor component is responsible for learning a policy that directly maps environment states to actions, and it is the same used by the REINFORCE algorithm. The Critic component, on the other hand, estimates the value function $V(s)$, providing feedback to the Actor about the quality of its chosen actions. For our implementation, the Critic network consists of a fully connected neural network architecture similar to the Actor, with the exception of outputting a single scalar value representing the expected return (value) of being in a particular state.

3.3.2 Weights update

After each episode, the collected experiences are used to compute the advantage estimation for policy updates and the bootstrapped discounted returns for value updates. The Actor's loss is computed using the policy gradient approach, optimizing for higher expected rewards by maximizing the advantage of chosen actions. Simultaneously, the Critic's loss is calculated to minimize the discrepancy between the estimated and actual returns, ensuring accurate value predictions.

Advantage estimation To update the Actor network, we first compute the advantage function $A(s, a)$. The advantage function measures how much better or worse an action is compared to the expected value of the state. It is calculated as:

$$A(s, a) = Q(s, a) - V(s), \tag{3}$$

where $Q(s, a)$ is the action-value function, and $V(s)$ is the state value function estimated by the Critic.

Policy gradient update The Actor's loss is computed using the policy gradient approach, which aims to maximize the expected cumulative reward. The policy gradient theorem provides the foundation for this update:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\pi_\theta, a\sim\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a|s) A(s, a)\right], \tag{4}$$

where θ are the parameters of the Actor network, and $\pi_\theta(a|s)$ is the policy. The Actor's loss function can then be defined as:

$$L_{\text{Actor}} = -\mathbb{E}\left[\log \pi_\theta(a|s) A(s, a)\right]. \tag{5}$$

By minimizing this loss, we encourage the policy to choose actions that have higher advantages.

Value function update The Critic network aims to minimize the discrepancy between the estimated state values $V(s)$ and the discounted returns. The target return R_t is calculated as:

$$R_t = r_t + \gamma V(s_{t+1}), \tag{6}$$

where r_t is the reward received at time step t, and γ is the discount factor. For the Critic's we used the Smooth L1 loss to improve robustness to outliers with respect to MSE loss:

$$\text{SmoothL1Loss}(x, y) = \begin{cases} 0.5(x - y)^2 & \text{for } |x - y| < 1, \\ |x - y| - 0.5 & \text{otherwise.} \end{cases} \tag{7}$$

Gradient descent After computing the loss functions for both Actor and Critic, we perform gradient descent updates using the Adam optimizer:

$$\theta \leftarrow \theta - \alpha \nabla_\theta L_{\text{Actor}}, \tag{8}$$

$$\phi \leftarrow \phi - \beta \nabla_\phi L_{\text{Critic}}, \tag{9}$$

where θ represents the parameters of the Actor network, ϕ represents the parameters of the Critic network, and α and β are the learning rates for the Actor and Critic networks, respectively. These learning rates can be different, allowing for tailored optimization of each network.

3.3.3 Training process

As we did for the REINFORCE algorithm, AC was trained without performing a hyperparameter grid search and the following fixed values are used: learning rate = 1e-3 and γ (discount factor) = 0.99. The training process was carried out in the source environment for 100K episodes for 5 different runs.

3.4. Soft Actor-Critic

After implementing two basic RL algorithms, we moved on to a more advanced one: Soft Actor-Critic. SAC is an off-policy method designed for continuous action spaces, offering better stability and efficiency than traditional algorithms due to also its use of entropy to enhance learning.

We used the implementation from Stable-Baselines 3 [8], a state-of-the-art RL library known for its efficient and stable algorithms.

3.4.1 Entropy in Soft Actor-Critic

Entropy plays a crucial role in SAC by quantifying the uncertainty or randomness in the policy's actions. Maximizing the entropy alongside expected cumulative reward encourages the policy to explore a wide range of actions, balancing exploration and exploitation.

In SAC, the policy $\pi(a|s)$ outputs a probability distribution over actions a given a state s, so we can define the entropy $H(\pi)$ as:

$$H(\pi) = -\mathbb{E}_{a\sim\pi}[\log \pi(a|s)] \tag{10}$$

By maximizing $H(\pi)$, SAC prioritizes policies that are more uncertain or diverse in actions, promoting more robust policies that are less likely to get stuck in suboptimal solutions.

3.4.2 Source and target environments

Due to practical constraints, no experiments were conducted on real robots, so we cannot claim an actual sim-to-real transfer scenario. To address the absence of a real setup, we created two custom domains: a source domain for algorithm training, simulating the virtual environment, and a target domain representing the real world. To simulate the reality gap, the torso mass of the hopper in the source domain was decreased by 1 kg.

3.4.3 Hyperparameter tuning

Since the SAC algorithm was already implemented, we focused on finding the best hyperparameters for training. We conducted a small grid search over the learning rate, buffer size, and batch size. The specific values tested are shown in Table 1.

Hyperparameter	Values
Learning Rate	[1e-4, 3e-4]
Buffer Size	[1e6, 5e6]
Batch Size	[256, 512]

Table 1. Hyperparameter Grid for SAC

For the first step, training and evaluation were conducted in the *source* environment. For each combination of hyperparameters, a new SAC model was trained and tested in the *CustomHopper-source-v0* environment. Specifically, each model was trained for 50K timesteps in the training environment and then evaluated over 50 episodes in the test environment to determine their performances. The combination yielding the highest mean reward was selected as the optimal set of hyperparameters.

3.4.4 Training and testing process

Using the hyperparameters identified in the previous step, we proceeded with the training and testing phases also in the *target* environment and for a larger number of timesteps. The SAC agent was trained both in the *CustomHopper-source-v0* and *CustomHopper-target-v0* environment for a total of 400K timesteps.

In the evaluation phase, the performance of the trained SAC models were assessed in both the *CustomHopper-source-v0* and *CustomHopper-target-v0* environment according to the following (training $\rightarrow$ test) configurations:

- Source $\rightarrow$ Source
- Source $\rightarrow$ Target
- Target $\rightarrow$ Target

The mean reward over 50 evaluation episodes was recorded to verify the agent's ability to adapt to changes in the environmental dynamics.

3.5. Uniform Domain Randomization with SAC

Starting from the previous results, we employed SAC to train agents in the source environment using Uniform Domain Randomization (UDR) to enhance the robustness of the learned policy. A custom Hopper environment was utilized, focusing on randomizing the body mass parameters to achieve this robustness.

3.5.1 Randomization process

The randomization process involved sampling new mass values from a uniform distribution within a defined range around the original values. This approach introduced controlled randomness into the training environment, compelling the agent to perform well across a spectrum of dynamics rather than overfitting to a single, static set of parameters.

To achieve this, we implemented a custom callback derived from `BaseCallback` in `stable_baselines3`. This callback modified the body mass parameters, except the torso, at the end of each episode, ensuring that the policy learned to adapt to the varying dynamics.

The mass values were sampled uniformly within a specified range around their nominal values, set to $[3.93, 2.71, 5.09]$. During training, a parameter α defined the range of uniform distribution, sampling body mass from $[(1-\alpha) \times \text{mass}, (1+\alpha) \times \text{mass}]$. We varied α to analyze its impact on the policy's robustness, using values of 0.1, 0.4, 0.7, and 1.

3.5.2 Training and testing process

The SAC agent was trained in the source environment for 400K timesteps, during which the body mass parameters were randomized using the UDR strategy. After training, the model was tested in the target environment over 50 episodes. The average returns were compared to evaluate how well UDR improved the policy's ability to generalize.

3.6. OpenAI Gym Cart Pole environment

The Cart Pole is part of the OpenAI Gym Classic Control environments, a set of basic environments to test and benchmark policies. The goal is to balance a pole on top of a moving cart for as long as possible by applying appropriate forces. The observation space is a 4D vector representing: cart position, cart velocity, pole angle and pole angular velocity. The action space is binary: push left or push right. The agent obtains a reward of +1 for each timestep the pole remains balanced.

3.7. Forward-Forward

The Forward-Forward algorithm is a greedy multi-layer learning procedure.

The goal of the network is to produce a high goodness for positive data and a low goodness for negative data. This is mathematically presented as a logistic regression:

$$P(positive) = \sigma(G - \theta) \tag{11}$$

Theta is an arbitrarily set threshold.

3.7.1 Why research Forward-Forward

FF is a novel approach that remains largely unexplored, presenting significant opportunities for innovative research.

FF can provide several advantages over backpropagation:

- Backpropagation is difficult to implement on analog hardware, while FF is more efficient on very low-power analog devices
- Backpropagation is biologically implausible, while FF can be used a model of learning in cortex [9]
- Backpropagation doesn't work on streams of sensory input without taking frequent time-outs for training, while with FF the negative pass can be delayed in time (mimicking sleep)

3.7.2 Goodness

The goodness is a function that takes in input the activations of the hidden layers and gives as output a real number.

Hinton explored two different measures of goodness for a layer: the sum of the squared neural activities or the negative sum of the squared activities. Going forward, we will consider only the positive goodness:

$$G_i = \sum_j y_j^2 \tag{12}$$

For the multi-layer case Hinton suggest to sum all the goodnesses without considering the first hidden layer:

$$G = \sum_{i=2}^{n} G_i \tag{13}$$

3.7.3 Positive and negative data

Understanding positive and negative data is crucial for the whole algorithm.

The forward passes on positive and negative data operate in exactly the same way as each other, but on different data and with opposite objectives. The positive pass operates on "real data" and adjusts the weights to increase the goodness, whereas the negative pass operates on "negative data" and adjusts the weights to decrease the goodness.

Choosing what are positive and negative data is still a human choice that depends on the task at hand.

3.7.4 Architecture

A feed-forward FF network can be structured like a Multi-Layer Perceptron but with the output/action concatenated with the input/state as input layer and no output layer.

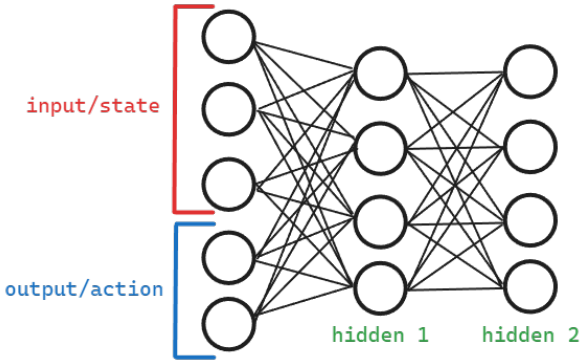

Figure 1. Forward-Forward multi-layer model

The hidden layers are activated with ReLU.

To prevent the information about goodness from leaking to subsequent layers (which would make further learning unnecessary), the activation of the hidden layer is normalized to have unit norm.

3.7.5 Training

The model's layers are trained one at a time (greedily).

During the training of each layer, the weights are adjusted like a single-layer regression model.

3.7.6 Inference

To make inference on the network we must find the output/action that maximizes the goodness.

In the discrete case (classification) we iterate over all the possibilities and chose the one with the greatest goodness.

In the continuous case (regression) we perform gradient ascent to maximize the goodness with respect to the output/action.

3.7.7 Adaptation for Reinforcement Learning

To integrate the algorithm into the RL framework, we made several key changes, while trying to preserve the original ideas.

The goodness is extended to include also the first layer, but we inserted a penalty coefficient:

$$G = \gamma * G_1 + \sum_{i=2}^{n} G_i \qquad \gamma \in [0, 1] \tag{14}$$

To distinguish between positive and negative data, we employed the Actor-Critic paradigm, utilizing a Critic network trained via backpropagation. The Actor network (FF) architecture is: $(state\_space + action\_space) \times 64 \times 64$. The Critic network architecture is: $(state\_space) \times 64 \times 64 \times 1$.

We used the value function to evaluate the quality of the actions taken. If the value function increases in the next timestep, the action is considered good, and we can classify the (state, action) pair as positive data, otherwise it is considered negative data. The final action is always regarded as bad because it leads to the end of the episode.

To implement progressive training without a fixed dataset or a predetermined number of epochs, we decided to provide the agent with the maximum number of episodes it will be trained on. Using this information, the FF network can train each layer for $\frac{\#episodes}{\#layers}$.

To determine the next action, we only consider the layers that are either fully trained or currently in training.

4. Results

4.1. REINFORCE vs Actor-Critic

The comparison of the performance of REINFORCE (both with and without a baseline) and Actor-Critic is visualized in Figure 2. The line plot with confidence interval shading represents the average rewards of 5 different runs obtained over the training episodes for each algorithm.

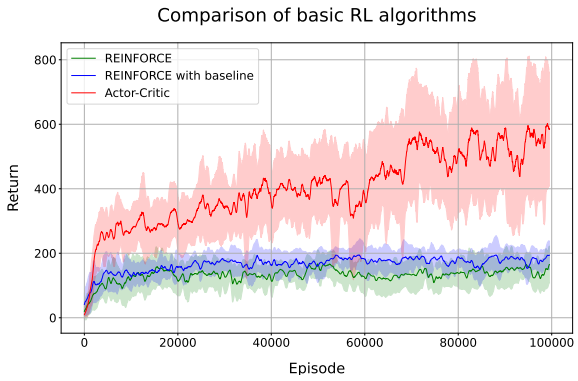

Figure 2. Comparison of the basic RL algorithms

The plot reveals that AC consistently outperforms REINFORCE. REINFORCE with a baseline shows better performance compared to its no-baseline counterpart, indicating the effectiveness of variance reduction. However, AC still achieves higher rewards throughout the training process.

4.2. SAC vs basic RL algorithms

The comparison plot between SAC and the basic RL algorithms is shown in Figure 3.

We must specify that the implementation of SAC in the Stable-Baselines 3 library allows us to set only the number of timesteps, not the number of episodes, for the training phase. Since the number of timesteps per episode varies during training, the results do not all have the same number of episodes.

For this reason, the visualization represents the first 1182 episodes, which is the smallest number of episodes obtained with 400K timesteps.

Although the comparison may seem unequal, the rapid convergence of SAC compared to the other algorithms is immediately evident. Despite the comparison being made only over 1182 episodes, SAC achieves a significantly higher average reward than AC after 100K episodes, demonstrating its superior learning ability even over a limited number of episodes. It must be noted that SAC training is much more time-consuming than the other algorithms, therefore, even though it is trained for only 400K timesteps instead of 100K episodes, it still takes longer.

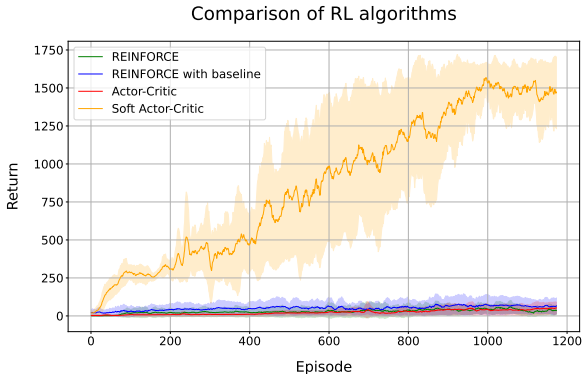

Figure 3. Comparison between SAC and the basic RL algorithms

4.3. SAC transfer learning

The results of the hyperparameter search performed in the source environment are presented in Table 2. From these results, we can conclude that the best parameters are a learning rate of 0.0003, a buffer size of 5M, and a batch size of 256.

Learning rate	Buffer size	Batch size	Mean reward
0.0001	1 M	256	340
0.0001	1 M	512	443
0.0001	5 M	256	285
0.0001	5 M	512	371
0.0003	1 M	256	322
0.0003	1 M	512	700
0.0003	5 M	256	1285
0.0003	5 M	512	693

Table 2. Hyperparameter search results for SAC (50K timesteps) in the Source → Source configuration

With the optimal hyperparameters identified, we trained and tested two agents in both the source and target environments. The following results were obtained over 50 test episodes:

- Source → Source: 1682
- Source → Target: 982
- Target → Target: 1289

These results indicate that we expect lower performance in the *source* → *target* configuration compared to the *target* → *target* configuration due to discrepancies between the source and target domains.

This significant drop in performance highlights the challenges associated with transferring policies trained in one domain to a different, unobserved domain, emphasizing the importance of addressing the reality gap in sim-to-real scenarios.

The *source* → *target* and *target* → *target* values will serve as the lower and upper bounds, respectively, for the subsequent Domain Randomization phase.

4.4. SAC with UDR

By employing UDR, the trained agents demonstrated improved robustness when transferring from the source to the target environment, highlighting the effectiveness of domain randomization techniques in overcoming the sim-to-real gap. This method significantly mitigates the performance drop typically observed when applying policies trained in simulation to real-world scenarios.

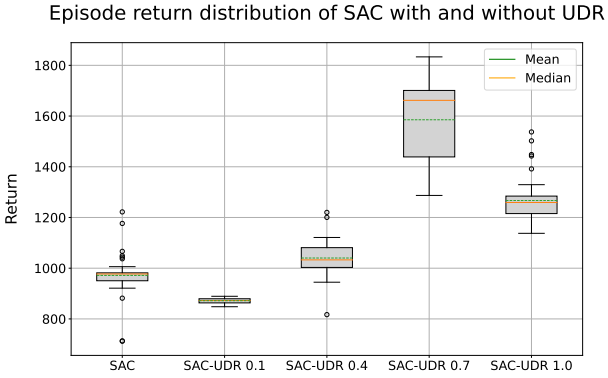

Figure 4. Boxplot of the rewards distribution of SAC without and with different levels of UDR in the source → target configuration. The number next to the box plot name on the horizontal axis is the value of α.

Figure 4 demonstrates the effectiveness of UDR and its limits. Insufficient randomization results in the policy failing to generalize, while excessive randomization prevents the convergence. Finding an optimal balance requires testing various levels of randomization. In our situation, it appears that randomizing with 70% is the preferred choice, despite the runs experiencing significant volatility.

4.5. Forward-Forward in Cart Pole

In Figure 5, we present a comparison between our novel approach, Forward-Forward, and the established benchmark, Actor-Critic. Each line represents the average of 5 runs, with the shaded area indicating the standard deviation.

Over 500 training episodes, it is evident that FF has successfully learned to balance the pole, consistently achieving a return of approximately 75. This result clearly demonstrates the algorithm's capability to adapt in a reinforcement learning context and effectively learn a efficient policy.

5. Conclusions

This report provides a comprehensive analysis of Reinforcement Learning applied to robotic systems, with a focus

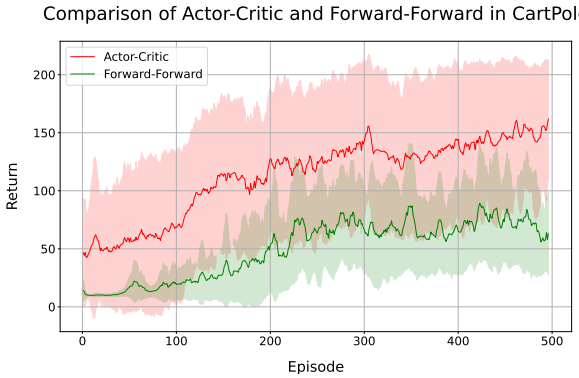

Figure 5. Comparison between Forward-Forward and Actor-Critic

on the sim-to-real transfer problem within the OpenAI Gym Hopper environment. Our investigation included the implementation and evaluation of both basic and advanced RL algorithms, such as REINFORCE, Actor-Critic, and Soft Actor-Critic. The results demonstrated the effectiveness of these algorithms in training control policies for simulated robots.

To address the sim-to-real gap, we employed Uniform Domain Randomization, which significantly improved the robustness of the policies to environmental changes and uncertainties. Our findings suggest that UDR is a viable technique for enhancing the generalization of RL agents.

Moreover, the project also explored the feasibility of integrating Forward-Forward into the RL framework. We tested this approach in the OpenAI Gym Cart Pole environment, demonstrating its potential as a promising alternative to traditional backpropagation.

In conclusion, our work contributes to the ongoing efforts to bridge the sim-to-real gap in RL applications for robotic control and opens new avenues for future research on Forward-Forward.

6. Further research

6.1. Real world

A limitation of this study is the absence of real-world testing. Such tests are essential for definitively assessing the performance of the developed policies and randomization techniques. Ultimately, these tests are crucial for the productive deployment of robots, ensuring they can perform useful work reliably and effectively in real-world scenarios.

6.2. Forward-Forward

Given that FF is a new algorithm, there is a significant amount of research yet to be conducted. Some potential areas for further investigation include:

- Training the algorithm in environments requiring continuous actions
- Testing the scalability with larger and more complex architectures
- Exploring different goodness measures (Hinton proposal)
- Exploring different activation functions (Hinton proposal)

References

[1] Geoffrey Hinton. The forward-forward algorithm: Some preliminary investigations, 2022. 1, 2

[2] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992. 1

[3] M. L. Littman, R. S. Sutton, and S. P. Singh. Advantage updating applied to a differential game. In *Advances in Neural Information Processing Systems*, pages 349–356, 1995. 1

[4] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1861–1870. PMLR, 10–15 Jul 2018. 2

[5] J. Tobin, T. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30, 2017. 2

[6] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016. 2

[7] Nebuly. Optimization: Forward-forward. https://github.com/nebuly-ai/nebuly/tree/main/optimization/forward_forward, 2024. Accessed: 2024-07-15. 2

[8] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. 4

[9] T. P. Lillicrap, A. Santoro, L. Marris, C. J. Akerman, and G. Hinton. Backpropagation and the brain. *Nature Reviews Neuroscience*, 21(6):335–346, 2020. 5