TRANSFER LEARNING FOR CONTINUOUS CONTROL

by

Suzan Ece Ada

B.S., Industrial Engineering, Koç University, 2015

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Software Engineering

Boğaziçi University

2019

ACKNOWLEDGEMENTS

First and foremost I would like to express my profound gratitude to my supervisor Prof. H. Levent Akın. I'm thoroughly grateful for his guidance in my quest of researching the subject I truly am passionate about. His wit, wisdom, experience, and patience has directed my focus to the crucial points when I drifted away from the main subject. He has always encouraged me to discover novel applications in this field. I have always been inspired by his meritorious remarks and unparalleled teachings that broadened my vision during the course of my research.

I would like to thank Asst. Prof. Emre Uğur for his distinguished teaching and generosity for providing the necessary resources throughout my research. I have gained lots of insight and experience through the application of state-of-art algorithms made possible by his powerful Ubuntu server.

I am indebted to Prof. Fatih Alagöz for his support and inspirational teachings. I thank him for all the advice and constant positivity he has given me.

I am truly blessed for having a family who has always supported me by all odds throughout my academic journey. I would like to thank my mother Enise Ö. Ada, my father Prof. R. Mehmet Ada and my brother T. Can Ada for standing by my side in times of need.

I would also like to thank Can Mangır for his endless support and wise suggestions throughout my graduate school.

I am truly thankful for the friendship of Yiğit Yıldırım, Metehan Doyran, Rahmetullah Varol, Binnur Görer, Okan Aşık, İbrahim Özcan, Melisa Şener, Serkan Buğur, Mert İmre, Ahmet Tekden, Yunus Şeker and Ersin Başaran. Our fruitful discussions will be truly missed.

ABSTRACT

TRANSFER LEARNING FOR CONTINUOUS CONTROL

Agents trained with deep reinforcement learning algorithms are capable of performing highly complex tasks including locomotion in continuous environments. In order to attain a human-level performance, the next step of research should be to investigate the ability to transfer the learning acquired in one task to unknown tasks. Concerns on generalization and overfitting in deep reinforcement learning are not usually addressed in current transfer learning research. This issue results in simplistic benchmarks and inaccurate algorithm comparisons due to rudimentary assessments. In this thesis, we propose novel regularization techniques exclusive to policy gradient algorithms for continuous control through the application of sample elimination and early stopping. By discarding samples that lead to overfitting via strict clipping we will generate robust policies for a humanoid with high generalization capacity. We also suggest the inclusion of training iteration to the hyperparameters in deep transfer learning problems. We recommend resorting to earlier snapshots of parameters depending on the target task due to the occurrence of overfitting to the source task. We demonstrate that a humanoid is capable of performing forward locomotion in unseen environments with different gravities and tangential frictions using strict clipping and early stopping. Furthermore, we evaluate our propositions on a delivery task where a humanoid is required to carry a heavy box while walking and inter-robot transfer tasks where the humanoid transfers its learning to taller and shorter robots. Because source task performance is not indicative of the generalization capacity of the algorithm we propose three different transfer learning evaluation methods. We increase the generalization capacity of a state-of-art adversarial algorithm by introducing entropy bonus, proposing different critic architectures and using simpler adversaries. Finally, we evaluate the robustness of these adversarial algorithms on morphologically modified hopper environments and environments with unknown gravities according to the criteria we proposed.

ÖZET

SÜREKLİ KONTROL İÇİN ÖĞRENME AKTARIMI

Derin pekiştirmeli öğrenme algoritmaları ile eğitilen etmenler, sürekli ortamlarda hareket dahil olmak üzere oldukça karmaşık görevleri gerçekleştirme yeteneğine sahiptir. İnsan düzeyinde bir performans elde etmek için bir görevde edinilen öğrenmeyi bilinmeyen görevlere transfer etme yeteneğini geliştirmek bu alandaki araştırmalarının bir sonraki adımı olmalıdır. Derin pekiştirmeli öğrenmede genelleme, öğrenim aktarımı araştırmalarında yeterince ele alınmamaktadır ve hatalı değerlendirme kriterlerine yol açarak yanlış algoritma karşılaştırmalarına neden olmaktadır. Bu tezde, örnekleme seçilimi ve erken durdurma yoluyla sürekli kontrol için politika gradyan algoritmalarına özgü yeni düzenleme teknikleri önerdik. Kırpma parametresi ile örnekleme seçilimi önererek aşırı öğrenmeye engel olarak, yüksek genelleme kapasitesine sahip bir robot için dayanıklı politikalar elde ettik. Derin öğrenme aktarımı problemlerinde yaygın olarak kullanılan hiperparametrelere optimizasyon iterasyonunun da dahil edilmesini önerdik. Yöntemlerimizin geçerliliğini farklı yerçekimleri ve teğetsel sürtünme ortamlarına başarılı öğrenim aktarımı gerçekleştirerek kanıtladık. Ağır kutu taşıyan bir kurye robotu deneyi tasarladık ve metotlarımızın üstün performansını grafiklerle gösterdik. Standart insansı robottan daha uzun ve daha kısa insansı robotlara başarılı bir şekilde yürüme görevini aktardık. Kaynak görev performansı, algoritmanın genelleştirme kapasitesinin bir göstergesi olmadığı için üç farklı öğrenimi aktarımı değerlendirme yöntemi önerdik. Entropi bonusu, farklı eleştirmen mimarileri ve müfredat öğrenimi kullanarak dayanıklı çekişmeli pekiştirmeli öğrenme algoritmasının genelleştirme kapasitesini arttırdık. Çekişmeli ağlar için genelleştirilmiş avantaj hesaplayıcısı tasarladık ve geliştirdiğimiz bu yöntem ile zıplayıcı robotu ağırlaştırdığımız hedef ortamda daha iyi performans gösteren politikalar elde ettik. Çekişmeli algoritmaların dayanıklılığını morfolojik olarak değiştirilmiş zıplayıcı robotlarda ve bilinmeyen yerçekimli ortamlarda tasarladığımız kriterlere göre değerlendirdik.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

$\mathscr{A}$	Set of actions
$\hat{A}$	Decaying telescoping sum of advantage function estimators
a	Action
b	Batch size
d	Binary value for episode termination
$\mathbb{E}$	Expectation
$E_t(s)$	Eligibility in state s at time t
g_t	Gradient at time t
H	Horizon(Trajectory size)
i	process
L	Loss function at time t
$\mathscr{L}_{batch}$	Loss function of the batch of samples
$q_*(s,a)$	Value of q-function taking action a in state s following the optimal policy
$q_{\pi_\theta}(s,a)$	Value of q-function taking action a in state s following the policy π_θ
r	Reward
$\mathscr{S}$	Set of states
$S[\pi_\theta]$	Entropy bonus function
s	State
t	Time
v_t	Gradient step at time t
$v_*(s)$	Value of value function in state s following the optimal policy
$v_{\pi_\theta}(s)$	Value of value function in state s following the policy π_θ
$V_{\pi_\theta}(t)$	Estimate of value function at time t following policy π_θ
$Q_{\pi_\theta}(t)$	Estimate of q-function at time t following policy π_θ
α	Step size hyperparameter
β_1	Exponential decay rate for the estimate of first moment

β_2	Exponential decay rate for the estimate of second raw moment
γ	Discount factor
δ_t	Advantage function estimator
ε	Clipping parameter
ε_1	Epsilon
ζ	Clipping constant for observations
η	Learning rate
θ	Policy parameter
λ	Exponential weighting factor
μ	Mean
π	Policy
$\pi_\theta(a_t\|s_t)$	Probability of taking action a_t given state s_t under policy π parametrized by θ
$\rho(s_0)$	Initial state distribution
$\rho(s_{t+1}\|s_t, a_t)$	State transition distribution
σ	Standard deviation
τ	Trajectory
χ	Curriculum Parameter

LIST OF ACRONYMS/ABBREVIATIONS

ACC-RARL	Average Consecutive Critic Robust Adversarial Reinforcement Learning
Adam	Adaptive Moment Estimation
AMN	Actor Mimic Network
DNN	Deep Neural Network
DQN	Deep Q-Network
EPOpt	Ensemble Policy Optimization
Joint PPO	Joint Proximal Policy Optimization
LSTM	Long Short-Term Memory
MAML	Model Agnostic Meta Learning
MDP	Markov Decision Process
MLP	Multi Layer Perceptron
MLSH	Meta Learning for Shared Hierarchies
PPO	Proximal Policy Optimization
RARL	Robust Adversarial Reinforcement Learning
RARL PS-Curriculum	Robust Adversarial Reinforcement Learning with Policy Storage Curriculum
RL	Reinforcement Learning
RELU	Rectified Linear Unit
RMSProp	Root Mean Square Propagation
RNN	Recurrent Neural Network
SC-RARL	Shared Critic Robust Adversarial Reinforcement Learning
SGD	Stochastic Gradient Descent
SVM	Support Vector Machines
TD-learning	Temporal Difference learning
TRPO	Trust Region Policy Optimization
OptionGAN	Option Generative Adversarial Networks
OSC	Operational Space Control

1. INTRODUCTION

Inferring the general intuition of the learning process and harnessing this to learn an unseen task is necessary for an autonomous agent to operate in non-stationary real-life environments. Being able to adapt to the changes in an environment as quickly as possible by cross-task generalization is preeminent to attain artificial intelligence.

Humans learn new tasks by utilizing the learning process from similar previous tasks they have faced or observed in real life. Generalizing well among these closely related tasks often leads to higher performance. This learning process has two major components that inspire our methodology:

(i) developing a generalizable method from the process of learning a task,
(ii) being able to adapt to the changes in the environment via experiences gained during learning

In real life, a robot will encounter different environments when executing tasks in a non-stationary environment. Deep reinforcement learning methods require long training periods in the source domain to develop a strategy close to a human for the same source environment [11]. Likewise, imitation learning approaches generate a reward function by observing the execution of the task by an expert [12].

Manually engineering a reward function for every task or waiting for a robot to interact millions of times with the environment for the purpose of acquiring a new skill is time-consuming and impractical. A robot is expected to generalize to a similar task it hasn't encountered, adequately and quickly in order to coexist with humans in the real world. As an illustration, the robot trained solely for walking can't move forward when it is expected to carry a box or transfer its learning to a taller or a shorter robot. In order to obtain robust policies that can excel in the aforementioned tasks the robot should not only learn walking but to learn walking robustly in many different cases and the learning attained should be transferable to other types of robots. Hence, the tradeoff between the robustness of the

learning and the source task performance should be acknowledged when determining the most pertinent training model.

Analogous to the variety of ways humans carry out a simple task in the real world, robots can perform a task in continuous control environments distinctively even though the loss function is immutable. A human can perform locomotion in many different ways ranging from closer to ground running to a careful tiptoeing on a rope. Drawing inspiration from how humans are able to resort to past knowledge gained during learning to excel in a task we have focused on increasing scope of abilities gained during learning.

The objective of this thesis is to answer the questions below:

(i) Are current benchmarks for transfer reinforcement learning algorithms sufficient for algorithm comparison and what evaluation criteria would better suit transfer deep reinforcement learning algorithms?
(ii) How can we increase the generalization capacity of current state-of-art transfer learning methods?

We first provide an outline of our propositions to the initial question:

(i) Initially we prove that source task environment performance isn't indicative of generalization capacity and target task performance. In continous control environments, we will leverage the recent policy gradient methods to continuously acquire knowledge in the form of neural network parameters. In order to achieve an adequate performance in the target task environment, this statement will be the origin of our methodology.
(ii) Evaluation of generalization in deep reinforcement learning is still an open research area in transfer learning field [13]. In Chapter 5, we will show how failure of recognizing overfitting would lead to inaccurate algorithm comparisons. We suggest three transfer learning evaluation structures based on the policy buffer we will propose in Section 3.1.1 and employ one of them in our evaluations. Policy buffer consists of promising policy parameters saved during training in the source task environment and the design is inspired by the human memory.

(iii) In Chapter 3, we propose new environments inspired real life scenarios for benchmarking transfer learning. First, we increased the range of gravity and robot torso mass experiments used by Henderson et al. [14], Rajeswaran et al. [1] and Pinto et al. [15] respectively to demonstrate the limitations of the methods we propose. Then we introduced new morphology and tangential friction environments for humanoid. Robots should be able to transfer the learning they've attained to each other like humans. Subsequently, we designed two new target environments named tall humanoid and short humanoid, each having different loss function constraints and morphologies than the standard humanoid source environment. Because carrying a heavy object is among the expectations of a service robot, we designed a humanoid delivery environment.

The prevalent problem of overfitting is starting to be acknowledged in deep reinforcement learning as the expectation for the algorithm's robustness increases [16]. In order to answer the second question specified in our thesis objectives, we focused on various ways of increasing the algorithm's generalization capacity.

Our contributions in this work are directed to combat the overfitting problem in transfer learning and increase the robustness of the state-of-art algorithms are summarized follows:

(i) After proving that the source task performance is not indicative of the performance in the target environment we introduced a new hyperparameter namely the policy iteration for benchmarking transfer learning. We will show that earlier iterations of policies performed astoundingly well in target task environments due to the regularization effect of early stopping. As a result, recognition of the policy iteration as a hyperparameter will not only prevent inaccurate algorithm evaluations but also potentially increase the performance of recent algorithms.

(ii) We introduce the method of strict clipping to discard samples that cause overfitting. This regularization technique is developed for the policy gradient algorithm named Proximal Policy Optimization(PPO) but we will discuss its possible applications to other algorithms in future directions.

(iii) We propose a new advantage estimation technique for Robust Adversarial Reinforcement Learning (RARL) [15] in Section 3.1.3 named Average Consecutive Critic Robust Adversarial Reinforcement Learning (ACC-RARL) by involving both critics at each iteration. In morphological experiments, we further demonstrated that a hopper robot can hop with more than twice it's original torso mass using learning attained in the standard environment with this technique [15].

(iv) We compare the generalization capacity of different training methods we proposed in Section 3.1.3 on a hopper robot, namely advantage estimation techniques, entropies bonuses and different curriculums [17] after including the iteration of training as a hyperparameter. These findings will further asses the neccessity of the evaluation technique we propose and will be beneficial in constructing a convenient policy buffer.

This thesis is organized as follows. In Chapter 2 we will provide background information on deep reinforcement learning and bipedal locomotion and review recent state-of-art methods introduced for meta-learning and transfer learning. The algorithms and the evaluation structure that will be used for the experiments will be discussed in Chapter 3. The hardware and software specifications will be detailed in Chapter 4. In Chapter 5 the algorithms suggested in Chapter 3 will be compared using modified humanoid and hopper environments in MuJoCo simulator. [18] Finally, in Chapter 6, the conclusion section will include a summary our contributions, discussion of the implications of our results, fruitful future directions for research.

2. BACKGROUND

Recent research on transfer learning has been build on top of deep reinforcement learning algorithms that utilize deep neural network as universal function approximators. As a consequence, in this chapter primarily the fundamental reinforcement learning algorithms will be discussed and then a thorough introductory material on the contemporary deep neural network architectures and optimizers will be reviewed. Thenceforth, widely used policy gradient and the current state-of-art transfer learning algorithms will be surveyed.

2.1. Reinforcement Learning

Reinforcement learning (RL) is a set of methods where learning is attained by utilizing the data gathered from the environment through trial and error. The most basic reinforcement learning algorithms consist of an environment and an agent. The agent's sole mission until termination is to maximize the reward function in initially unknown environmental dynamics, exploiting only observations and reward signals. As a consequence, the design of an accurate reward function is the pivot point in achieving efficacious learning. Deep reinforcement learning is a popular field of research which incorporates deep neural networks into reinforcement learning algorithms.

2.1.1. Markov Decision Process

Reinforcement learning is inherently an iterative decision process of an agent formulated as a Markov Decision Process (MDP). We characterize the MDPs with the initial state distribution $\rho(s_0) : \mathscr{S} \to \mathbb{R}$, state transition distribution $\rho(s_{t+1}|s_t, a_t) : \mathscr{S} \times \mathscr{A} \times \mathscr{S} \to \mathbb{R}$, reward function $r_t : \mathscr{S} \times \mathscr{A} \times \mathscr{S} \to \mathbb{R}$ and discount factor $\gamma \in (0,1)$ by $(\mathscr{S}, \mathscr{A}, \rho(s_{t+1}|s_t, a_t)$, $r_t, \rho(s_0), \gamma)$. If the decision process is sampled using the policy $\pi_\theta(a_t|s_t) : \mathscr{S} \times \mathscr{A} \to [0,1]$ it becomes a Markov reward process because the agent is following a specific decision. MDPs must initially consort to the Markov property. Markov property enforces the independence of future states from the past states while being conditioned on the current state [6].

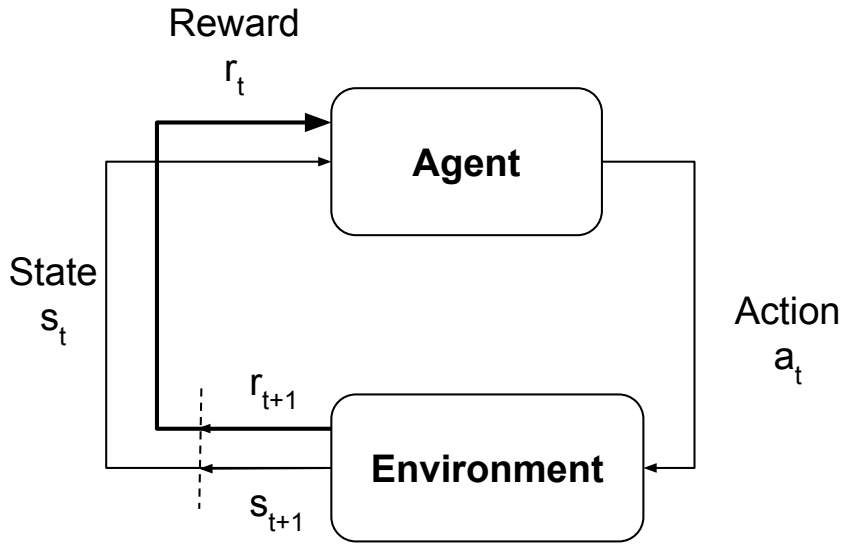

Figure 2.1. Reinforcement Learning from Environment Interaction [6]

In a reinforcement learning setting given in Figure 2.1, the framework by the aforementioned tuple with policy π translates to an agent taking actions in an environment based on the parametrized stochastic policy π_θ and receiving a corresponding reward while transitioning from one state to another. The transition loop is completed when agent obtains a reward and a state as a result of the action it chose and the environmental dynamics. Discount factor encourages the agent to give priority to rewards with higher proximity and induces the algorithm to converge in infinite horizon cases. In stochastic environments, the agent might not end up in the same state by taking the same action in the same state. Hence the reward r_t from that action depends on the complete transition information denoted by the tuple (s_t, a_t, s_{t+1}).

2.1.2. Value Function, Q-Function and Advantage Function

Value function $V_{\pi_\theta}(s)$ at state s_t is the expected value of rewards R_{t+k+1} discounted by parameter γ from state s_t onwards following policy π_θ. Action-value function also known as the *Q-function* $Q_{\pi_\theta}(s,a)$ is the expected value of discounted rewards G_t from state s_t onwards by taking action a_t in state s_t given by the policy π_θ. Both are used for the purpose

of evaluating current policies by calculating the necessary reward signals received from the interactions with the environment. *Value function* and *Q-function* equations in Sutton *et al.* [6] are given in Equation 2.1.

$$\begin{aligned} V_{\pi_\theta}(s) &\doteq E_{\pi_\theta}[G_t|S_t = s] = E_{\pi_\theta}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s] \\ Q_{\pi_\theta}(s,a) &\doteq E_{\pi_\theta}[G_t|S_t = s, A_t = a] = \mathbb{E}_{\pi_\theta}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \end{aligned} \tag{2.1}$$

As the name suggests the *advantage function* $A_{\pi_\theta}(s,a)$ provides the relative benefit of taking the action a_t compared taking to all the other possible actions at state s_t. Value function serves as a baseline and reduces variance. Current research methods increasingly use the advantage function and aim to estimate it for policy evaluation. As given in Equation 2.2, the advantage function employs both of the functions in Equation 2.1

$$A_{\pi_\theta}(s,a) \doteq Q_{\pi_\theta}(s,a) - V_{\pi_\theta}(s) \tag{2.2}$$

2.1.3. Bellman Optimality

Bellman equation for the value function in discrete action spaces is the sum of Q-values over the set of actions weighted by the probability of taking each action given state, $\pi_\theta(a|s)$ given in Equation 2.3. This equation provides a top-down model for the current and forthcoming states. In continuous cases, policy is a probability density function instead of a probability mass function so we integrate Q-values over the probability density function. Q-values are given by the sum of immediate reward r and the value-function of the next state $v_{\pi_\theta}(s')$ discounted by a factor γ weighted by the transition probabilities $p(s',r|s,a)$ that are dependent on the environment dynamics and agent's action.

$$v_{\pi_\theta}(s) = \sum_a \pi_\theta(a|s) \sum_{s',r} p\left(s', r|s, a\right) \left[r + \gamma v_{\pi_\theta}\left(s'\right)\right], \quad \text{for all } s \in \mathscr{S} \tag{2.3}$$

On-policy algorithms are methods where the same policy is evaluated and used for sampling for each episode. Conversely, *off-policy algorithms* are used in cases where the objective policy being evaluated is not the policy that the trajectories are being sampled. [6] Notice that the v_{π_θ} for the next state introduces an ambiguity. v_{π_θ} is learned through exhaustive iterations via *Dynamic Programming* [6] by the convergence of the decaying rewards. *Dynamic Programming* is an *on policy algorithm* that requires the environment dynamic to be known. Nevertheless, in cases where the agent is allowed to have limited interaction, environment dynamics are not known or the state and action spaces are large or continuous v_{π_θ} has to be approximated. For the case below, the agent has to perform a one-step-ahead search to determine the optimal v_{π_θ}. The optimal value of the value function is derived by taking the action that maximizes $\sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')$. Bellman optimality equation for $v_*(s)$ is given in Equation 2.4

$$\begin{aligned} v_*(s) &= \max_a E[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a] \\ v_*(s) &= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \end{aligned} \tag{2.4}$$

Unlike $v_*(s)$ the agent does not have to do a one-step-ahead search when finding $q_*(s,a)$ because the values of all actions are available at the cost of memory [6]. Bellman optimality equation for action-value function $q_*(s,a)$ is given in Equation 2.5:

$$\begin{aligned} q_*(s,a) &= E[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a')|S_t = s, A_t = a] \\ q_*(s,a) &= \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \end{aligned} \tag{2.5}$$

Exploration and exploitation dilemma arises from the tradeoff between the greedy and the random policies. Exploitation occurs when the agent acts according to what it has known so far and takes the maximum rewarding action at the cost of converging to a suboptimal policy. This eventually results in undiscovered states and unsampled actions during learning thus the agent should explore by taking either random actions or actions that are one standard deviation away from the mean in the continuous case. An exploration bonus is sometimes introduced into the reward function to promote state space discovery.

2.1.4. Model-Free Reinforcement Learning

In most of the real-world settings, the environment dynamics rely on complex interactions of many components. Model-free reinforcement learning algorithms are suitable for cases where the transition dynamics is not known. Transition dynamics are not used in policy gradient algorithms inspired by REINFORCE [19], instead they are estimated using samples. Likewise, Monte Carlo and Temporal Difference Learning [6] are also other widely used value estimation techniques under the topic of Model-free reinforcement learning.

2.1.5. Monte Carlo and Temporal Difference Learning

Monte-Carlo learning is a set of model-free algorithms used to approximate the value function at a particular state by sampling many trajectories. For instance, in first visit Monte Carlo, the counter that tracks the number of visits to a particular state is updated only the first time a state is visited. In the same manner, the cumulative rewards from time-step t onwards and other rewards from previous episodes for state s are added. Finally, the value function of the state s is averaged by the most recent sum of discounted rewards and the counter. Similar to that, in every visit Monte Carlo evaluation, the same process is repeated each time the state is visited in an episode. Counter incrementation and the value update schemes are alike in incremental Monte Carlo and every visit Monte Carlo learning. However, the update algorithm in incremental Monte Carlo learning includes a parameter α to discount

older value estimates as seen in Equation 2.6:

$$V(s_t) \leftarrow V(s_t) + \alpha \left(G_t^{(n)} - V(s_t) \right) \tag{2.6}$$

In *Temporal Difference learning (TD-learning)* the value of states are estimated by looking n-steps into the future and are computed by moving the current estimated value of that state closer to the estimated *Temporal Difference target (TD-target)*. [6]. TD(0) is shown in the Equation 2.7 where the value function is encouraged to move towards the biased TD-target computed as $r_{t+1} + \gamma V(s_{t+1})$ by one-step-lookahead.

$$V(s_t) \leftarrow V(s_t) + \alpha (r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \tag{2.7}$$

λ in forward-view *TD-learning* is the weight coefficient used in geometric average of n-step discounted rewards as seen in Equation 2.8. Each n-step look ahead is weighted geometrically to construct the TD-target value. The dilemma that occurs during the estimation process can be identified as a bias-variance tradeoff between the Monte Carlo learning that has higher variance and the TD-learning for one-step-lookahead which has lower variance but more bias. As might be expected incremental every-visit Monte Carlo learning is the same as TD(1).

$$G_t^{\lambda} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)} \tag{2.8}$$

In Monte Carlo learning, the trajectories should be rolled out until the end of the episode to compute the reward function whereas in *TD-learning* the value of states can be updated after at least 1 episode. In the case of offline updates, the updates are not made until

the episode termination whereas in online cases the updates can be made before the terminal state is reached. Monte Carlo updates have higher variance compared to TD-learning but they are unbiased because the rewards are summed until the end of the episode so the value function is close to the true value.

2.1.6. Eligibility Traces

Credit assignment problem arises from not knowing how much each action contributes to the result [20]. Backward $TD(\lambda)$ algorithm exploits the credit assignment problem in the value function estimation. Accrediting the states that are visited more frequently or the states towards the end more for the performance is used occasionally. Equation 2.9 shows the computation of the eligibility function of the state.

$$\begin{aligned} E_0(s) &= 0 \\ E_t(s) &= \gamma\lambda E_{t-1}(s) + 1(S_t = s) \end{aligned} \tag{2.9}$$

In backward *TD(λ)* algorithm defined in Equation 2.10, the value function is updated using the eligibility of the corresponding state $E_t(s)$, update parameter α and *TD-error* δ_t.

$$\begin{aligned} \delta_t &= R_{t+1} + \gamma V(s_{t+1}) - V(s_t) \\ V(s) &\leftarrow V(s) + \alpha\delta_t E_t(s) \end{aligned} \tag{2.10}$$

Accurate evaluation of the policy depends on the accuracy of the reward function.

2.2. Deep Neural Networks

The proliferation of hype revolving around deep neural networks is due to the proven successful real-world applications [11,21,22]. Complex tasks like image recognition, speech processing, recommender systems, action recognition, finance, and deep reinforcement learning have elaborate functional representations. Most of these implementations consist of numerous unknown parameters that cannot be hand-engineered and tuned specifically for each task. Deep neural networks are nonlinear function approximators that learn from the input data by optimizing the loss function via updating the predetermined set of parameters.

Deep reinforcement learning and supervised learning tasks are analogous in a sense that they both intend to translate the data to an output for a specific purpose based on a loss function. In deep reinforcement learning, the trajectories sampled from the environment are used as the input data and the output varies from Q-functions to the set of probabilities of actions depending on the method.

2.2.1. Artificial Neuron

The analogy between simplified biological neurons and artificial neurons are extensively mentioned in current research because the building block of deep learning has relevance to how a signal travels across the neurons [23]. Structure of an artificial neuron is inspired by the dendrites and axons of biological neurons. Signals traveling along the dendrites are analogous to data being fed to the neuron. Similar to the biological mechanism of impulse transmission to the axons, the connections emerging from the neuron representing each component of the input data have different strengths. The summation of the weighted input to the neuron becomes the input of a predetermined activation function. Bias value is added to the summation as given in Equation 2.11.

$$y = f\left(\sum_i w_i x_i + b\right) \tag{2.11}$$

Forward propagation is the process of computing the output values by sequentially using equations like Equation 2.11. The output value is used as an input to the loss function that will be used to update the function parameters by the most befitting optimization algorithm.

2.2.2. Activation Functions

In deep neural networks, artificial neurons essentially are all nonlinear functions which conform to the characteristic of receiving input and producing a related output. If nonlinear activations were not applied then the whole deep neural network function can be formulated as a single linear function which would make the hidden layers redundant.

2.2.2.1. Sigmoid. The *Sigmoid function* squashes the linear input value between 0 and 1. It basically decides whether the neuron should be activated or not by outputting values that are close to 1 or 0 most of the time. The Sigmoid function is given in Equation 2.12.

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.12}$$

However, the emergence of more sophisticated activation functions hindered the popularity of sigmoid function. For instance, *tanh* activation is the 0-centered alternative to the sigmoid function which allows the flow of both negative and positive gradients during backpropagation. Equivalently, the derivative of the sigmoid function is most of the time close to 0 which hinders gradient flow.

2.2.2.2. Hyperbolic Tangent. Similar to the sigmoid function *tanh function* squashes the input between -1 and 1. Presence of both negative and positive outputs increase expressivity of the network during backpropagation. Although hyperbolic tangent (*tanh*) is 0 centered compared to the sigmoid alternative, the gradient flow is still burdensome due to the multitude of tiny gradients close to 0. The *Tanh* function is given in Equation 2.13.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.13}$$

In Chapter 5 we will be using *tanh* as the activation function of the policy and value function approximator networks.

2.2.2.3. Rectified Linear Unit. The *Rectified Linear Unit*(RELU) is among the most popular choices because of its computational speed. The idea behind it is to exclude all negative input values and output their value as 0. The linearity of the operation is the rationale behind its computation speed. The *RELU* function is given in Equation 2.14.

$$RELU(x) = \max(0, x) \tag{2.14}$$

Learning rate selection is crucial, especially for this activation function because a neuron might die and start outputting only 0 values that doesn't effect the output.

2.2.2.4. Leaky RELU. *Leaky RELU* is an updated version of the RELU which prevents dying neurons by allowing slight negative values to pass over the neuron by multiplying the input with a small constant. Leaky RELU function is given in Equation 2.15 where β is a small hyperparameter between 0 and 1.

$$LeakyRELU(x) = \max(\beta x, x) \tag{2.15}$$

2.2.3. Neural Network Architectures

Deep neural networks (DNN) are interconnected artificial neurons in multiple layered structures. In Figure 2.2, a fully connected 3-layer network is shown with 2 hidden layers.

Primarily, the structure of the network is set out then the incoming data and the loss function update the parameters to approximate the parameters of the nonlinear function. Characteristics of the data and the task at hand determine the number of neurons at each layer, the number of layers and how the network will be connected.

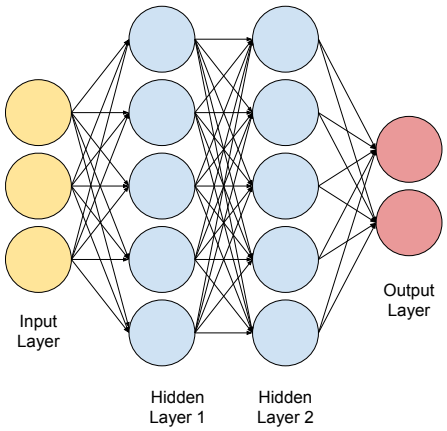

Figure 2.2. A fully connected 3-layer neural network

The data are transferred to the hidden layers through the input layers. Output layer represents how the deep neural network perceives the input data with the current parameters. The data and the loss function are used to determine the parameters of the unknown complex nonlinear function. Backward propagation is used to get DNN's perception of the data close to our expectations. Most of the time mini batches of data are fed to the model as input and a nonlinear function at each neuron transforms the data being fed to output which will be used as one of the inputs for another neuron's nonlinear function. The input layer is not counted as a layer in the $N-layer$ network naming convention, only the output and the hidden layers are taken into account. DNN model, where each neuron in a layer is connected to each layer in the upcoming layer are called *fully connected layers*.

The computation direction is always forward in a feed-forward artificial neural network whereas in recurrent neural networks a more complex architecture is seen where a neuron's output might be an input to a neuron in the previous layer. In Figure 2.3, a many-to-many

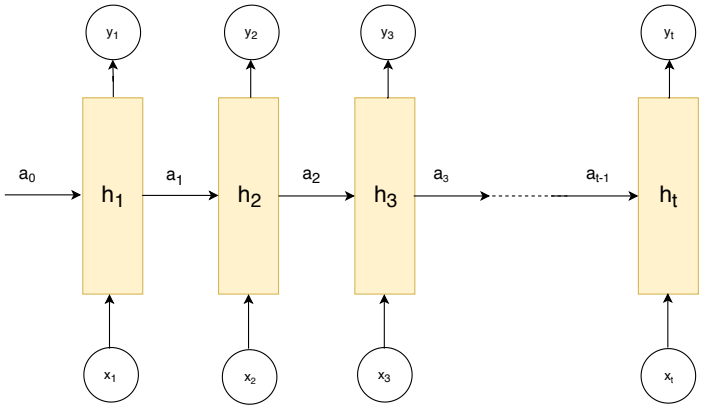

Figure 2.3. A many-to-many recurrent neural network [7]

recurrent neural network architecture is provided where the output of a layer is fed to the next layer along with the next input to exploit sequential structures in data [7].

2.2.4. Loss Functions

The loss function is the most significant component of a neural network that determines how the parameters will be updated to achieve optimal approximation. In neural network optimization, most of the time mini batch gradient descent is performed over a batch of input data. Data loss is the average of each input loss and computed as in Equation 2.16.

$$L = \frac{1}{N}\sum_{i} L_i \tag{2.16}$$

2.2.4.1. Suppor Vector Machine. *Support vectors* are N-1 dimensional hyperplanes used in classification and regression tasks to seperate data points in N-dimensional feature space. Multi-Class Suppor Vector Machine (SVM) is a widely used algorithm for classification where f_j is the output activation that is not equal to the activation corresponding to the label [24]. Δ value is 1 for classification where a binary tensor is expected as an output.

Support Vector Machine loss is computed as in Equation 2.17

$$L_i = \sum_{j \neq y_i} \max\left(0, f_j - f_{y_i} + \Delta\right) \tag{2.17}$$

2.2.4.2. Cross Entropy Loss. Cross-entropy loss is also broadly used for classification tasks like image recognition. The function inside $-log$ is called *Softmax*. *Softmax function* exponentiates the output activations and compresses them between 0 and 1 by computing each class' probabilities as in Equation 2.18 where labels are one-hot vectors.

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \tag{2.18}$$

Cross-entropy loss interprets the scores in a more elaborate way by outputting a probability tensor. Hence it resolves a more complicated problem where all classes are optimized instead of one class.

2.2.5. Regularization

Regularization techniques are developed to prevent overfitting to the training data by adding a penalty to the known overfitting indications.

2.2.5.1. $L1$ Regularization. Each new parameter in the neural network increases the complexity of the function and makes it harder for the network to generalize to the unseen test data. λ is the regularization hyperparameter and specifies how much we should penalize overfitting. To overcome this problem, all of the weights are added to the loss function as seen in Equation 2.19.

$$L_{reg}(x) = L(x) + \lambda \sum_{i=1}^{k} |w_i| \tag{2.19}$$

2.2.5.2. *L2* Regularization. *L2* Regularization has the identical motives with the aforementioned *L1* Regularization. The 1/2 coefficient is put to inactivate the effect of the unnecessary multiplier 2 that will come from the gradient. *L2*'s decaying effect on the weight parameters is more stable than *L1* Regularization because it will decay it linearly proportional to the weight whereas *L1* will tend to make all the weights directly 0. The updated loss function with *L2* Regularization is given in Equation 2.20.

$$L_{reg}(x) = L(x) + (\lambda/2)\sum_{i=1}^{k} w_i^2 \tag{2.20}$$

2.2.5.3. Dropout. Srivastava *et al.* [25] proposed the *Dropout* method to avoid neurons to overfit to specific features of the data. During the training phase, a dropout layer deactivates each neuron in that layer by a probability p where p is $0 < p < 1$. Deep neural networks are universal function approximators thus this techniques intends to reduce overrepresentation of the training data in order ton increase the generalization capacity of the neural network.

2.2.6. Optimization

Optimizers are a set of algorithms used to update the parameters of a function approximator to minimize the loss function. In the deep neural networks setting, the parameters of the network are updated iteratively via a selected optimizer algorithm. Attaining the global minimum is the aim of the optimizer algorithms but occasionally they suffer from complications that arise from hyperparameter tuning, insufficient data or bad choice of optimizers.

Gradient descent optimization is performed by first randomly initializing and then iteratively updating the parameter vector. The update consists of the partial derivative vector of the loss function and the step size α as given in Equation 2.21. The gradient vector determines the direction and the magnitude of the update step. If the aim is to maximize a reward function, the update is added to the current parameter vector.

$$\theta := \theta - \alpha \frac{\delta}{\delta \theta} J(\theta) \tag{2.21}$$

Step size is one of the hyperparameters that affect the route of the descent immensely. Suboptimal convergence to a local minimum is a variant of the widely mentioned exploration problem. The step-sizes are thus larger in the initial stages of the optimization because the algorithm should explore the parameter space thoroughly. Similarly, the step-sizes should decay towards the final stages of the gradient descent to avoid overshooting of the minimum.

2.2.6.1. Stochastic Gradient Descent. In gradient descent, all data should forward propagate along the network to compute the new loss each time the parameters are updated. The derivative of current loss with respect to the parameters are computed and movement in the parameter space via backpropagation. One drawback of gradient descent is that it is computationally expensive. In contrast, its alternative *Stochastic Gradient Descent* (SGD) performs update over each data point. Although the direction of updates in gradient descent is more accurate than the SGD, it takes less time to compute the gradient.

2.2.6.2. Mini-Batch Gradient Descent. Vectorized representations of data allow the computation of gradient over a batch of inputs. Thus depending on the hardware restrictions data can be divided into mini-batches to attain a middle ground between the robustness of *Gradient Descent* and speed of *Stochastic Gradient Descent* in *Mini-Batch Gradient Descent*.

2.2.6.3. Momentum. A constant learning rate will cause oscillations once parameter values gets closer to the optimal point. *Standard Momentum Gradient Descent* makes use of the previous gradient values to gain momentum towards the minimum. SGD makes noisy updates but momentum is a method developed to find the optimal point with less variance faster. In Equation 2.22, the η value is the learning rate that scales the current step size of the gradient, μ is the increasing momentum coefficient between $[0, 1]$ that creates resistance

by including the previous iteration's resulting momentum gradient step [26].

$$\Delta v_t^{momentum} = \mu v_{t-1}^{momentum} - \eta \nabla_\theta f(\theta) \tag{2.22}$$

2.2.6.4. Nesterov's Accelerated Gradient. First, a momentum step is taken based on the previous gradients in the *Nesterov method* [27] and then the gradient is computed at the point after the momentum update [26]. Then, the parameter vector is updated using the derivative of loss at the momentum updated parameter and new momentum value as in Equation 2.23.

$$\Delta v_t^{Nesterov} = \mu v_{t-1}^{Nesterov} - \eta \nabla_\theta f(\theta + \mu v_{t-1}^{Nesterov}) \tag{2.23}$$

2.2.6.5. Root Mean Square Propagation. *Root Mean Square Propagation (RMSProp)* is the adapted version of gradient descent with momentum presented by Hinton [28]. In Equation 2.24, the proposed update to each parameter is given.

$$\begin{aligned} E\left[g^2\right]_t &= \rho E\left[g^2\right]_{t-1} + (1-\rho) * g_t^2 \\ \Delta\theta_t^{RMSProp} &= -\frac{\eta}{\sqrt{E\left[g^2\right]_t + \varepsilon_1}} * g_t \end{aligned} \tag{2.24}$$

The larger the exponential average of squared gradients the smaller the step size for the new gradient step will be. This will decrease oscillations and make the gradient step smoother. In addition to that as the values begin to converge *RMSProp* algorithm avoids the update to overshoot the minima. In contrast to the SGD with Momentum, *RMSProp* is directly estimating the best step size value for each parameter separately instead of globally adjusting the gradients.

2.2.6.6. Adam. *Adaptive Moment Estimation (Adam)* optimizer [29] is inspired from both adaptive optimization methods and SGD with Momentum to compute a more optimal step

size in several cases where the learning rate decreases when the gradients are too large but increases if they are consistently large. The Adam optimizer algorithm's suggested step size is given in Equation 2.25.

$$\begin{aligned} m_t &= \beta_1 * m_{t-1} + (1-\beta_1) * g_t \\ v_t &= \beta_2 * v_{t-1} + (1-\beta_2) * g_t^2 \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\ \Delta\theta_t &= -\eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon_1} \end{aligned} \tag{2.25}$$

Most of the time the decaying hyperparameter β_2 is larger than β_1. ε_1 is an arbitrarily chosen small number like 0.00001 [30].

2.3. Policy Gradient Methods

Policy in continuous environments refers to the parametrized probability distribution function over the action space that depends on the states. The policy $\pi_\theta(a|s)$ for time t given in Equation 2.26, is the probability of taking action a in state s with parameter θ .

$$\pi_\theta(a|s) = \Pr\{A_t = a | S_t = s, \theta_t = \theta\} \tag{2.26}$$

In policy gradient algorithms, the parameters of the policy network are updated via batches of trajectories collected from the environment. If the policy being updated is the policy used for sampling trajectories then it is called an on-policy gradient algorithm. *Policy Gradient* is defined as the partial derivative vector of the performance function. In this the-

sis, we will examine continuous action spaces where vectors of actions are sampled from the corresponding Multivariate Gaussian distributions. Each action element in the vector of actions is represented by a univariate Gaussian function with a mean and a standard deviation. Policy network used for continuous actions spaces is a neural network given in Equation 2.27 that takes the state as input and outputs the mean vector of a Multivariate Gaussian distribution. A seperate vector of standard deviations is optimized with the policy network at each iteration simultaneously.

$$\begin{aligned}\pi_\theta(a_t|s_t) &= \mathcal{N}(f_{\text{neural network}}(s_t);\Sigma) \\ &= \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}}\exp\left(-\frac{1}{2}(a_t - f(s_t))^T\Sigma^{-1}(a_t - f(s_t))\right)\end{aligned} \tag{2.27}$$

Step size hyperparameter has a substantial impact on the policy optimization. The policy parameter might end up on a flat region of the reward function and escaping from that location via state-of-art optimization algorithms might be hard. Even the small steps in the policy parameter space might result in drastic changes in the performance.

REINFORCE [19] is widely referred as the vanilla policy gradient algorithm that updates the policy parameter sequentially by sampling sets of trajectories under policy π_θ. Performance function, the expected reward of the trajectories under policy π_θ, and the corresponding vanilla policy gradient are given in Equation 2.28.

$$J(\theta) = E_{\tau \sim \pi_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right]$$

$$\theta^\star = \arg\max_\theta J(\theta)$$

$$\nabla_\theta J(\theta) = \begin{pmatrix} \frac{\delta J(\theta)}{\delta \theta_1} \\ \vdots \\ \frac{\delta J(\theta)}{\delta \theta_n} \end{pmatrix} \tag{2.28}$$

$$\nabla_\theta J(\theta) = E_{\tau \sim \pi_\theta(\tau)} [\nabla_\theta \log \pi_\theta(\tau) r(\tau)]$$

Trajectory is a set of sequential state and action tuples of length Horizon H given the policy π_θ. Probability that the sampled trajectory rolls out under parameter θ is given in Equation 2.29:

$$\begin{aligned} \pi_\theta(s_1,\, a_1, \ldots,\, s_T,\, a_T) &= p(\, s_1) \prod_{t=1}^{T} \pi_\theta(\, a_t |\, s_t)\, p(\, s_{t+1} |\, s_t,\, a_t) \\ \log \pi_\theta(\tau) &= \log p(\, s_1) + \sum_{t=1}^{T} \log \pi_\theta(\, a_t |\, s_t) + \log p(\, s_{t+1} |\, s_t,\, a_t) \end{aligned} \tag{2.29}$$

The probability of the initial state and the state transition probabilities are discarded while taking the derivative with respect to θ as in Equation 2.30. This implies that knowing the state transition probabilities of the environment and the initial state distribution is not necessary, we only need the reward function and policy for optimization. We perform mini-batch gradient descent utilizing each batch of data gathered from the experience with the environment thus we know the action taken in each step and the corresponding rewards.

$$
\begin{aligned}
J(\theta) &= E_{\tau\sim\pi_\theta(\tau)}[\sum_t r(s_t,a_t)] \approx \frac{1}{N}\sum_i\sum_t r(s_{i,t},a_{i,t}) \\
\nabla_\theta J(\theta) &= E_{\tau\sim\pi_\theta(\tau)}[(\sum_{t=1}^{T}\nabla_\theta \log\pi_\theta(a_t|s_t))(\sum_{t=1}^{T} r(s_t,a_t))]
\end{aligned}
\tag{2.30}
$$

Using Monte Carlo sampling the value of the policy gradient is estimated in Equation 2.31.

$$
\begin{aligned}
\nabla_\theta J(\theta) &\approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log\pi_\theta\left(\, a_{i,t}|\, s_{i,t}\right)\right)\left(\sum_{t=1}^{T} r\left(\, s_{i,t},\, a_{i,t}\right)\right) \\
\theta &\leftarrow \theta + \alpha\nabla_\theta J(\theta)
\end{aligned}
\tag{2.31}
$$

In the supervised learning setting, the aim is to find the parameters that best contruct a model by feeding data as input to the deep neural network. Given the sampled state-action pairs and rewards, to find the optimal parameters that best fit the data, first the probabilities of observing each state-action pair are multiplied. Then, gradient descent is performed to find the corresponding parameters that maximize the occurrence of rewarding data. In policy networks, an action vector is output by the network given the state. The objective is to find the parameter that maximizes the probabilities of the most rewarding action conditioned on the input state. In contradistinction to the idea of each sample having equal weight while fitting the parameters, in early policy gradient algorithms, the weight of sample is updated by the cumulative rewards after taking action a at state s from current time step until the end of the episode (Q-Value). Negative rewards assure the decrease in probability density of the corresponding action occurring at that state.

Vanilla policy gradient is a sample inefficient on-policy algorithm where samples collected from previous policies are discarded after the update is performed. By importance sampling, we can use the samples from the old policy while performing multiple iterations

of mini batch gradient descent. *Proximal Policy Optimization* (PPO) is a variant of these set of algorithms where the old policy is the constant denominator in the policy ratio and it stays the same until the last sampled trajectory is exhausted by the predetermined number of optimization iterations [31]. In these off-policy gradient methods like PPO and Trust Region Policy Optimization (TRPO) old policy is used to restrict the movement in the parameter space as well as importance correction. PPO algorithm will be the primary policy gradient algorithm we will use in our experiments thus further explanation will be provided in Section 2.5. Similarly Fujita *et al.* [32]introduced an algorithm similar to *Proximal Policy Optimization* named *Clipped Action Policy Gradients*. Instead of clipping with importance sampling and using the old policy's likelihood function they clipped the actions directly with constants α and β.

2.4. Actor Critic Methods

Policy gradient methods rely on the performance function to increase the probability of more rewarding actions. *Performance function* $J(\theta)$ has various definitions ranging from the summation of discounted rewards over all sampled trajectories to the advantage function. Various types of reward functions have been introduced for the policy gradient methods. Increasing interest in this area of research is due to the strong influence of them on the policy gradients.

When the policy and reward functions are unwrapped to timesteps from horizon-length trajectories, a better option of taking the reward from the current timestep onwards becomes prominent. Expected sum of rewards from time t to the terminal state conditioned on $state_t$ and $action_t$ tuple, denoted as the action-value function is still an unsophisticated choice because it doesn't provide information on the performance of that action relative to other possible actions. Hence a more suitable reward function which offers more information would be the advantage function which is derived as in Equation 2.32.

$$\begin{aligned} \hat{g} &= \frac{1}{N}\sum_{n=1}^{N}\sum_{t=0}^{\infty}\hat{A}_t^n \nabla_\theta \log \pi_\theta\left(a_t^n|s_t^n\right) \\ E_{s_{t+1}}[\delta_t^{V^{\pi,\gamma}}] &= E_{s_{t+1}}[r_t + \gamma V^{\pi,\gamma}(s_{t+1}) - V^{\pi,\gamma}(s_t)] \\ E_{s_{t+1}}[Q^{\pi,\gamma}(s_t,a_t) - V^{\pi,\gamma}(s_t)] &= A^{\pi,\gamma}(s_t,a_t) \end{aligned} \tag{2.32}$$

Using advantage function for policy gradient methods will result in a decrease in the probabilities of the less than average performing samples by the induction of negative rewards via advantage estimation. Correspondingly, positive rewards increase the probabilities of better performing samples.

In actor-critic methods [33], both policy network and value network approximators are optimized. The critic is the value-function which evaluates the policy and actor is the policy that takes actions according to the policy parameters. The actor takes actions based on the guidance of the critic. Policy gradient algorithms rely heavily on the advantage function during the backpropagation phase because the output of the critic value approximation network is used to calculate the advantage function during the policy optimization. The policy parameters aren't merely updated by using the advantages computed using the sample of that particular iteration but also by the information from previous iterations embedded in the output of the critic network. Advantage function estimation is a subject of bias-variance tradeoff. Decreasing the number of steps into the future reduces the variance at the cost of introducing the bias.

2.5. Proximal Policy Optimization with Generalized Advantage Estimation

Policy gradient methods have gained vast popularity against Deep Q-Networks (DQNs) [11] after the introduction of algorithms that constrain gradient movement in the policy parameter space such as Trust Region Policy Optimization(TRPO) and Proximal Policy Optimization (PPO) [31, 34]. In the open-source *OpenAI Baselines* framework [30], PPO [31] is used to optimize the actor policy network and the generalized advantage estimator (GAE) is

used to optimize the critic value function network simultaneously [35].

High bias suggests that the model underfits data and generalizes too much whereas high variance is an indication of overfitting to the data by taking the noise too much into account [36]. Value function approximation is a supervised learning problem where bias-variance tradeoff occurs. Schulman *et al.* [35] proposed a more sophisticated generalized advantage estimation technique that tackles this topic of bias-variance tradeoff. The γ hyperparameter first reduces the variance by decaying the rewards that are far away. The decaying telescoping sum of advantage function estimators δ_t^V with differing k-steps into the future is denoted as $\hat{A}_t^{(k)}$. Expansions of different k-step estimators are provided in Equation 2.33.

$$\begin{aligned}
\hat{A}_t^{(k=1)} &:= \delta_t^V && = -V(s_t) + r_t + \gamma V(s_{t+1}) \\
\hat{A}_t^{(k=2)} &:= \delta_t^V + \gamma\delta_{t+1}^V && = -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) \\
\hat{A}_t^{(k=3)} &:= \delta_t^V + \gamma\delta_{t+1}^V + \gamma^2\delta_{t+2}^V && = -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 V(s_{t+3}) \\
\hat{A}_t^{(k=\infty)} &= \textstyle\sum_{l=0}^{\infty} \gamma^l \delta_{t+l}^V && = -V(s_t) + \textstyle\sum_{l=0}^{\infty} \gamma^l r_{t+l}
\end{aligned} \tag{2.33}$$

Similar to the formula of $TD(\lambda)$ provided in Equation 2.8, the k-step advantage estimators are weighted exponentially to balance bias and variance. Bias is the smallest for $\hat{A}_t^{(\infty)}$ because the reward function is known for all future states of the trajectory. Although γ reduces variance by decaying the weights of the rewards far away, it increases the bias in an undiscounted estimation. As a result, the data might underfit by generalizing too much due to the high weights assigned to the most recent rewards. Determining a low λ would result in higher bias and lower variance if value function is an approximation. On the other hand, this GAE value will be further away the real advantage value is the value function is not completely accurate. The latter estimate that has lower bias has higher accuracy at the cost of variance. $\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)}$ derivation is provided in Equation 2.34 [35]. Schulman *et al.* [35] suggest that the optimal λ should be lower than the optimal γ value.

$$\begin{aligned}\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} &= (1-\lambda)\left(\hat{A}_t^{(k=1)} + \lambda\hat{A}_t^{(k=2)} + \lambda^2\hat{A}_t^{(k=3)} + \dots\right) \\ &= (1-\lambda)\left(\delta_t^V\left(\frac{1}{1-\lambda}\right) + \gamma\delta_{t+1}^V\left(\frac{\lambda}{1-\lambda}\right) + \gamma^2\delta_{t+2}^V\left(\frac{\lambda^2}{1-\lambda}\right) + \dots\right) \\ &= \sum_{l=0}^{\infty}(\gamma\lambda)^l\delta_{t+l}^V\end{aligned} \tag{2.34}$$

Practical calculations of advantage function estimators make use of finite horizon trajectories sampled from multiple environments. Advantage function is estimated by T-step-lookahead of immediate rewards and the value function in asynchronous advantage critic [37]. Trajectories of predetermined length are accumulated from multiple agents running simulataneously in different environments. Each gradient step is taken with respect to the current critic network parameters using the samples gathered from parallel runs. Inspired by this implementation, a truncated version of generalized advantage estimation is used in PPO, provided in Equation 2.35 [31] .

$$\begin{aligned}&\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \delta_t + (\gamma\lambda)\delta_{t+1} + \dots + \dots + (\gamma\lambda)^{T-t+1}\delta_{T-1} \\ &\text{where } \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)\end{aligned} \tag{2.35}$$

Value functions $V(s_t)$ in GAE formula are found by performing forward propagation on the value function approximator network (critic). Policy gradient's advantage function component is then updated by using the rewards sampled from parallel environments and the approximated value function output. An example of GAE usage in vanilla policy gradient algorithm is shown in Equation 2.36.

$$g^{\gamma} \approx E\left[\sum_{t=0}^{\infty}\nabla_\theta \log\pi_\theta(a_t|s_t)\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)}\right] = E\left[\sum_{t=0}^{\infty}\nabla_\theta \log\pi_\theta(a_t|s_t)\sum_{l=0}^{\infty}(\gamma\lambda)^l\delta_{t+l}^V\right] \tag{2.36}$$

Actor-Critic architectures are used to estimate the advantages [35] because we do not know the actual value of the state, as a matter of fact, we are only trying to approximate their value based on the samples we have seen so far. Thus, it is reasonable to train a separate critic network to predict the value of a given state.

Value function loss is the mean squared difference between the target value V_t^{targ} and the predicted critic network output. In the *OpenAI Baselines* framework, the target value is the sum of generalized advantage estimation and the output of the value function network as seen in Equation 2.37 [30]. Mini-batch gradient descent is performed over both actor and critic networks using the set of trajectories for multiple epochs so the samples are used for multiple times to compute updates. Target values are constant throughout the global optimization iteration.

$$\begin{aligned} V_t^{\text{targ}} &= \hat{A}_t^{\text{GAE}(\gamma,\lambda)} + V_t^{sampled} \\ L_t^{VF}(\theta) &= \left(V_\theta(s_t) - V_t^{\text{targ}}\right)^2 \end{aligned} \tag{2.37}$$

The *surrogate objective* $L^{CPI}(\theta)$ [31,34] shown in Equation 2.38 is used in both TRPO and PPO. The advantage estimates are computed by the samples gathered using the old policy $\pi_{\theta_{\text{old}}}$. Hence, in order to take multiple gradient steps utilizing the latest samples collected using π_θ , the importance sampling correction is necessary. At each iteration, the policy gradient step is computed by taking the derivative of the surrogate objective with respect to θ when performing mini batch gradient descent.

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t^{\text{GAE}(\gamma,\lambda)}\right] = \hat{\mathbb{E}}_t\left[r_t(\theta)\hat{A}_t^{\text{GAE}(\gamma,\lambda)}\right] \tag{2.38}$$

If the advantage is negative then all the ratios below $1-\varepsilon$ will be clipped because the optimistic scenario is to decrease the likelihood of sampling that action under the new policy. Likewise, if the advantage is positive, then all the ratios above $1+\varepsilon$ will be clipped and the gradient of clipped loss will be 0. In both cases, the clipping assures that the current location of the policy parameters in the parameter space is invariant since it is already sufficiently good. This protects against updates that would result in a catastrophically worse location.

$$L^{CLIP}(\theta) \quad = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\varepsilon, 1+\varepsilon\right)\hat{A}_t \right) \right] \tag{2.39}$$

Equation 2.40 consists of value-function-approximator loss L_t^{VF}, entropy reward L_t^S and the clipped surrogate objective L_t^{CLIP}.

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right] \tag{2.40}$$

At the very first iteration of surrogate loss calculation, the ratio of the old and new policy is 1 because the parameters are not yet updated. We will take the derivative with respect to the new parameters so the value of old policy is a constant importance correction. The output layer size is the same as the agent's action space. Concurrently, the logarithm of standard deviations vector is optimized that has the same dimensions as the action space of the agent. The output mean and the vector of logarithm of standard deviations are used to compute the diagonal multivariate probability distribution. It must be borne in mind that, policy is stochastic in this case, given the observation, it will sample randomly from the distribution with mean $f(s_t)$ and variance Σ.

The parallel implementation of the algorithm is provided in Figure 2.5 where d is a binary that denotes whether the episode terminated or not. Adam optimization is used to

Require α : step size hyperparameter

Require i : process

Require n : number of parallel processes

Initialize the fully connected layer weights using $\sigma = 0.01$ and the $log\sigma$ vector with 0 in θ ;

for $k = 1$ to $Iteratons$ **do**

Sample a trajectory $\tau = (s_1, a_1, r_1, d_1 ... s_H)$ by θ

Compute advantage estimates by θ_{critic} and τ, $\hat{A} = \hat{A}_{s_1,a_1} ... \hat{A}_{s_H,a_H}$ using Equation 2.35

$\theta_{\mathbf{old}} \Leftarrow \theta$;

for $e = 1$ to $Epochs$ **do**

for $batch = 1$ to $BatchSize$ **do**

Compute the $\mathscr{L}_{batch_i}$ using Equation 2.40;

$\theta \Leftarrow \theta - \alpha \nabla_\theta \frac{1}{n} \sum_{process_i} \mathscr{L}_{batch}$;

end for

end for

end for

Figure 2.4. PPO Algorithm.

compute the parameter updates according to the Equation 2.25. Because this algorithm will be used for the baseline policy gradient method in our experiments, further implementation details will be discussed in Chapter 5.

2.6. Transfer Learning

A human brain can easily transfer the learning attained from one task to another, for instance, babies use their knowledge of crawling to walk. Transfer Learning exploits the similarities between different but related tasks by utilizing the learning acquired in source task to the target task. Transfer learning is implemented in the cases where the environment dynamics or the tasks in the training and testing phases are different. Training a task from the beginning is time-consuming and most of the time computationally expensive. Consequently, it is widely used in the areas of image and speech recognition, natural language processing and reinforcement learning. As an illustration, given a learning curve of performance against timesteps, the transfer learning algorithms are designed to attain either higher slope, higher asymptote or higher start than the learning curve of non-transfer counterparts. [38]

In some settings, it is possible to achieve higher performance in all cases but according to the requirement of the design, sometimes the purpose can be just to achieve one of them. For instance, in computer vision, the network's accuracy is proportional to the amount of data until the overfitting point so if the data is insufficient, it is compelling to beat algorithms with transfer learning.

Time to threshold is also one of the aspirations to attain a successful transfer that focuses on decreasing the amount of time needed to achieve a certain value of goal [39]. In curriculum learning, it is used in Autonomous Task Sequencing for Customized Curriculum Design in Reinforcement Learning [40] to compare the performance of different curriculum methods.

Transfer Learning is widely used in the computer vision field. Most of the current research in the field use pretrained weights for the network parameter initialization of a new task. The pretrained weights usually dominate the earlier layer and the following layers have

parameters specific to the problem [41]. More precisely convolutional layers at the earlier layers trained on bigger datasets retain information on the shared features of the common image recognition tasks such as edge detectors [41]. Depending on the structure of the problem, fine-tuning the convolutional layers are allowed during training phase or the earlier layers are frozen where gradient flow through the convolutional layers are restricted and new layers are put on top of the network.

Continuous vectorized representations are used to represent observation and action space in deep reinforcement learning. However, they are not only used in the reinforcement learning domain. Words are mapped to a continuous meaning space in natural language processing. Subsequently, this suggests that the learning methods used in reinforcement learning might be applicable to NLP for future work. Likewise, parameters of word representations that are trained on vast data are used for initialization [42].

2.6.1. Forward Transfer Learning

Transferring the knowledge and representation gained from one task to another task is called *one task transfer learning*. Most applications include transferring the deep neural network parameters to the new task using modifications to the course of training or the architecture to begin learning from a higher start or to learn faster [1, 43]. The transferred neural network is expected to generalize to the new task and adapt to the new task's domain.

Development in computer vision has advanced the research in robotics because most complex robotics tasks process image data. Inspired by the fruitful implementations of transfer learning on computer vision field Rusu *et al.* [43] implemented fine-tuning and addition of new layers to an already trained network in the deep reinforcement learning in Atari domain. In their work, earlier convolutional layers are initialized with the parameters of a source task and frozen. The last layers of the network are allowed to specialize in the target task.

A transfer can occur from a large domain to a small domain and vice versa. In both cases, the possibility of negative transfer exists. If the algorithm is performing worse than

not using any transfer at all it is called *negative transfer learning*. For instance, if the algorithm is initialized from an optimal point for the source task but a suboptimal point for the target task, insufficient exploration might occur which would result in a negative transfer so random initialization would have been a better choice. When a transfer occurs from a larger domain to a smaller domain it is called *partial transfer learning*. One example of this in computer vision is tackled by Cao *et al.* [44] for the case where the source space is the subset of the target space. The transfer cases they have covered in the experiments section include performing transfer from *ImageNet 1000* [45] to *Caltech 84*, and *Caltech 256* [46] to *ImageNet 84* where the corresponding numbers stand for the number of classes. Solely utilizing the labels of the source domain for the associated labels belonging to the target domain does not prevent the occurrence of negative transfer. The labels not present in the target domain are called the *outlier labels*. Outlier label data present in the source domain reduce the classifier's accuracy thus to counteract this, a domain discriminator network and an opposing domain-invariant feature extractor network are assembled in an adversarial architecture. The loss function simultaneously maximizes the discriminator's and features extractor's accuracy [44]. Then, a domain discriminator is assigned for each class to obtain the weighted average of the domain loss where the outlier classes have less weight. In consequence, they have attempted to counteract the effects of negative transfer via discriminating the outlier classes from the source domain and maximizing the accuracy of the source and target distribution locations.

Transferring from simulation to the real world is often times a tedious task. Model-free algorithms rely on samples however the cost of sampling from a real-world environment is high in robotics settings. Tobin *et al.* [47] implemented the method of domain randomization in the physics simulator to accurately localize the objects in the real world for a manipulation task. Similarly Sadeghi *et al.* [48] used domain variation in 3D Modeling simulator Blender [49] by generating distinct pieces of furniture and hallway textures to train a simulated quadcopter. They later transferred the learned Q-value function network to a real-world quadcopter to avoid collisions while flying through a hallway.

Adversarial scenarios have also been widely used for robotics tasks involving vision. For instance, Bousmalis *et al.* [50], and Tzeng *et al.* [51] used adversarial networks where

one neural network is optimized to discriminate the image data of real world from the simulation and the other optimizes to generate simulator images that can fool the discriminator. The generator network will come up with better representations of the real image data as the discriminator approaches the minima.

2.6.1.1. Ensemble Policy Optimization [1]. Control tasks have benefitted immensely from the adversarial implementations. Increasing the robustness of policy leads to high performance in a multitude of target tasks. Rajeswaran *et al.* [1] suggested a method to increase robustness of the policy network in *Ensemble Policy Optimization*(EPOpt) algorithm by training an ensemble of different tasks. Their dual-step approach to perform transfer from one distribution of tasks to another distribution of tasks consists of *Robust Policy Search* where policy optimization is performed using samples from a batch of different tasks and *Model-Based Bayesian Reinforcement Learning* where the source task distribution parameters are updated via experience on the target task during training. In the policy search phase denoted by the Equation 2.41, the expected rewards of the trajectories from each task are computed.

$$J_{\mathscr{D}}(\theta) = \mathbb{E}_{p\sim\mathscr{P}}[\mathbb{E}_{\hat{\tau}}[\sum_{t=0}^{T-1}\gamma^t r_t(s_t,a_t)|p]] = \mathbb{E}_{\tau}[\sum_{t=0}^{T-1}\gamma^t r_t(s_t,a_t)] \tag{2.41}$$

The conditional value at risk algorithm presented by Tamar *et al.* [52], is used to form a subset of the worst performing ε-percentile of the distribution. Experiments in EPOpt further asses that performing batch policy optimization solely on the worst performing subset as seen in Equation 2.42 while discarding the higher performing trajectories leads to more robust policies.

$$\begin{aligned} \mathscr{F}(\theta) &= \{p|J_{\mathscr{M}}(\theta,p) \leq y\} \\ \max_{\theta,y} \int_{\mathscr{F}(\theta)} J_{\mathscr{M}}(\theta,p)\mathscr{P}(p)dp \quad &\text{s.t.} \quad \mathbb{P}(J_{\mathscr{M}}(\theta,P) \leq y) = \varepsilon \end{aligned} \tag{2.42}$$

In the source task update phase, the ensemble's distribution parameters are updated given the trajectories sampled from the target task to move the source distribution parameters closer to the target distribution parameters. In Equation 2.43, Bayesian inference is used to update the posterior distribution of the source task parameters given the trajectories sampled from the target environment. Initially, a set of parameters (p_i) are sampled from the target task distribution. Then the set of model parameters and the sampled trajectories are used to formulate the Bayesian equation. $P_p(p_i)$ is our initial belief (prior) of the target task parameters which is the probability of sampling target task parameters from the source task parameters. Using importance sampling we correct the $P_p(p_i)$ by dividing it to the probability of sampling the set (p_i) from the target task distribution to be able to use the likelihood function formed by the trajectories from the target task distribution given the set of (p_i). The update uses the probability of the next state instead of the probability of actions given the state because the target environment's response is more crucial to adapt the source task model parameters since we aim to learn the model of the target environment.

$$
\begin{aligned}
\mathbb{P}(P|\tau_k) &= \frac{1}{Z} \times \mathbb{P}(\tau_k|P) \times \mathbb{P}(P) = \frac{1}{Z} \times \prod_{t=0}^{T-1} \mathbb{P}(S_{t+1} = s_{t+1}^{(k)} | s_t^{(k)}, a_t^{(k)}, p) \times \mathbb{P}(P = p) \\
\mathbb{P}(S_{t+1}|s_t, a_t, p) &\equiv \mathscr{T}_p(s_t, a_t) \\
p_i &= [p_1, p_2, \ldots, p_M] \\
\mathbb{P}(p_i|\tau_k) &\propto \mathscr{L}(\tau_k|p_i) \times \frac{\mathbb{P}_P(p_i)}{\mathbb{P}_S(p_i)}
\end{aligned}
\tag{2.43}
$$

Two phases are repeated consecutively for a predetermined number of iterations. Although not experimented in the paper, this setting is applicable to problems where a limited number of trials are allowed in the real world target setting. The simulation would represent the source task distribution that is iteratively getting better at simulating the real world environment. Accordingly, our suggestions in Chapter 3 will be targeted for these use cases. All in all, they provide satisfactory results by comparing their results from *EPOpt* on 10^{th} percentile with Trust Region Policy Optimization (TRPO) trained on a single source task for each mass, *EPOpt* without the use of worst percentile subset extraction.

2.6.1.2. Adversarial and Competitive Environments. In Robust Adversarial Reinforcement Learning (RARL) [15], a separate adversarial network is created to destabilize the agent during training in a more intelligent way for increased robustness in the target environment. RARL algorithm will be used as the baseline algorithms in some of our experiments so detailed information will be presented in Section 3.1.3 . The proposed scenario is a two-player zero sum discounted game where the agent tries to maximize its own reward which is actively minimized by the adversary. Actions (A_1) are sampled from the agent's policy denoted as μ whereas the actions (A_2) are rolled out from the adversary's policy ν. Equation 2.44 refers to the reward function of the agent. Corresponding to that, the reward function of the adversary is $R^2 = -R^1$.

$$R^1 = E_{s_0 \sim \rho, a^1 \sim \theta^{pro}(s), a^2 \sim \theta^{adv}(s)} \left[\sum_{t=0}^{T-1} r^1 \left(s, a^1, a^2 \right) \right] \tag{2.44}$$

Due to computational constraints, instead of optimizing the minimax equation at each iteration they maximized the reward functions of the agent and the adversary consecutively. For j number of iterations, the agent's policy is optimized iteratively, while collecting samples using a fixed adversary policy. After that, the same number of rollouts are collected from the environment for the adversary's optimization while the agent's last policy's parameters are used. It is not required for the policy networks of the agent and the adversary to have the same output layer sizes. First, they have compared TRPO and RARL using the default environment hyperparameters without any disturbances for 500 iterations on tasks HalfCheetah, Swimmer, Hopper, and Walker2d in MuJoCo environments with 50 different seeds. Training with RARL achieved a better mean reward than the TRPO. In addition to that, RARL and TRPO are evaluated with a trained adversary in the test environment where RARL performed significantly better on all tasks compared to TRPO. They've also tested the aforementioned tasks by varying torso mass and friction coefficients which were not seen during the training phase and again RARL yielded better results than the TRPO baseline. An implementation of the algorithm using *rllab* framework and a single critic, simultaneous PPO variant using

OpenAI Baselines framework are open-sourced [10] [53] [54] [8]. We will compare double, single and shared double critic structure in Section 5.3.1 and Section 5.5.1 to evaluate how the critic effects the algorithm's generalization capability using *OpenAI Baselines* framework.

Separate critic networks are consecutively optimized with their policy network counterparts in RARL [15]. However, the reward functions of the protagonist and the antagonist are each others' negative in the algorithm thus a single shared critic network is also a relevant architecture. For instance, in Dong's implementation both of the networks are optimized redundantly but only the protagonist critic network's resulting advantage estimation is used in policy optimization resulting in a single shared critic network architecture [8]. Shared Critic Robust Adversarial Reinforcement Learning (SC-RARL) is the second type of architecture we will use for our experiments in Chapter 5.

Inspired by the fruitful results of RARL [15], Shioya *et al.* [17] proposed two extensions to the RARL algorithm by varying the adversary policies. Their first proposal is to add a penalty term as seen in Equation 2.45 to the adversary policy's reward function by sampling from the test domain to adapt the source task's transition function. This, however, is tailored robustness for each test task at hand which requires peeking into the test domain similar to the Bayesian update used in EPOpt [1].

$$\min_{\pi_{adv}} R + \lambda \frac{1}{N} \sum_{i}^{N} \left\| s_{t+1} - T_t\left(s_t, a_t\right) \right\|^2 \tag{2.45}$$

The second extension is inspired by Curriculum Learning to select the adversarial agents based on the progress of learning instead of naively taking the latest adversarial policy. The protagonist policies trained in harder environments doesn't guarantee a more robust performance, in fact, the randomization of adversaries during training was explored in some previous works [55] [17]. In both Shioya *et al.*'s [17] and Bansal *et al.*'s [55] experiments, using the latest and the hardest adversarial policy hinders the learning progression of the

protagonist. In [17], first, multiple adversaries are created and samples from the latest T iterations of the adversary policies are ranked according to the progress of learning using linear regression. The ranks are used to determine the probabilities of using the samples which are selected stochastically during training. Each adversary policy is maximized using the negative reward of the protagonist agent and the sum of KL Divergence from all the other adversary policies as seen in Equation 2.46 to encourage diversity between the multiple adversarial agents.

$$\min_{\pi_{adv}} R - \gamma \sum_{ij} KL\left(\pi_{adv_i}(a_t|s_t)\,\pi_{adv_j}(a_t|s_t)\right) \tag{2.46}$$

Bansal *et al.* [55] used Uniform $(\delta v, v)$ distribution for determining opponent humanoid's policy iteration where δ is the percentage of the constructed set's coverage from the last T adversary policy iterations. Shioya *et al.* [17] used multiple adversaries and ranked each sample's performance to determine the set of samples that should be used for optimization.

Experiments were done using Hopper and Walker2d environments in MuJoCo [18] to compare the results to the RARL Algorithm [17]. It is found that in the Hopper environment ranking the policies to adapt the probability of their selection performs better than RARL and uniform random selection but performed worse than the latter in Walker2d environment. In both of the environments using the trajectories which yielded fewer rewards performed worse than all the methods. As a contrast to this result, optimizing over the worst performing samples generated more robust policies in Hopper task when tested with different torso masses in EPOpt Algorithm [1] when an adversary policy is not present.

The adversarial algorithms can be considered as a dynamic way of creating different tasks for the agent at each iteration to encourage it to be more robust in unseen test conditions [56]. Challenging tasks allow agents to grasp complex latent features of the task thereby enable more robust policies to flourish.

2.6.2. Multi-Task Transfer Learning

In contrast to forward task transfer learning, in a multi-task transfer learning setting, the transferred knowledge, and the representation are based on multiple different tasks. In order to excel in multiple tasks in the test environment, these algorithms aim to attain an overall target environment performance [57]. Source tasks may have different loss functions or domains but should be relevant so that the extracted knowledge can be used in a new setting.

Transferring the best performing policy network parameters to a new task would oftentimes result in a negative transfer. Optimizing the policy network might wipe out a good initialization or may hinder the exploration of the new task. Initializing the neural network parameters in target task with the policy network trained on multiple tasks is one way of performing multitask transfer. Joint Proximal Policy Optimization (Joint PPO) [58] is a multi-task learning variant of PPO [31] which jointly trains all tasks from the distribution and performs a gradient step using a set of rollouts sampled from different tasks. Joint PPO should have a large batch size for each gradient update because small batches do not provide sufficient diversity to decrease the bias for an efficient gradient step. In PPO [31], a variant named *RoboschoolHumanoidFlagrun* where the agent is expected to run in various directions is also experimented.

Multi-task transfer learning heavily relies on the distribution and features of the tasks used in training. Excluding the tasks causing negative transfer thus distilling the relevant knowledge from the source tasks might solve the challenges encountered in multi-task transfer problems. If the exclusion is attempted in the testing phase it might reduce the learning speed. Nevertheless, recognizing the irrelevant source tasks during training is challenging without using any information from the target task.

2.6.2.1. Actor Mimic Network [2]. Multi-task transfer learning is utilized in both discrete and continuous environments. Parisotto *et al.* [2] suggested an actor mimic method to learn different games in the *Atari* domain . The first phase of the method aims to construct Deep

Q-Network (DQN) trained on a multitude of source tasks that is capable of performing each source task. The action space is discrete in the *Atari* domain and the Q-values in the output layer differ between source tasks. The output layer of the DQN network that consists Q-values is converted to a policy network using *Boltzmann distribution*. Due to the fact that the action spaces of the *Atari* games differ, multitask actor mimic network's outer layer consists of 18 units. Network architectures of DQNs of each source task and the Actor Mimic Network (AMN) are the same in some cases except the outer layers. The objective of the AMN for each source task is the summation of the cross-entropy between source task and the AMN and the feature regression loss. Feature regression loss intends to predict the last hidden layer of the source task DQN given the last hidden layer of AMN. The second phase involves transferring the multitask AMN to a new target task which is not seen in the training phase by initializing the DQN of the new target task via AMN without the softmax layer.

2.6.3. Multi-Task Meta-Learning

Meta-learning, scrutinizes the way different tasks that belong to the same distribution learn. Although the setting in multi-task transfer and meta-learning is the same, their main difference resides in the representation that is being transferred. Especially the essential focus of meta learning is the learning process. The objective is not to learn a certain task but to obtain the best initialization parameters for all the tasks in the distribution set. Finding the meta-policy parameters that directs the agent to receive higher task-specific reward in test tasks after performing a few gradient steps for adaptation is the ultimate goal for meta-learning in few-shot learning setting.

Meta-learning for few-shot classification has gained popularity in the field of computer vision [3, 59–61]. In the typical naming convention for *K-shot N-way classification* K stands for the number of examples from each given class and N represents the number of classes for each task. Moreover, in meta-training phase, each input task consists of both K examples from each N number of classes and a test set of novel samples from the N classes. During the meta-testing phase, the resulting representation from the training phase is used to classify the similarly structured task that involves novel data. The input training data can also be

structured as a recurrent neural network where each subtask in the training set gets input from the previous task. For instance Ravi *et al.* [59] and Al-Shedivat *et al.* [56] constructed an Long Short-Term Memory (LSTM) meta learner to exploit the sequence structure of the data for image classification and continuous control respectively. Duan *et al.* [62] also focused on the recurrent implementations on continuous control by representing trajectories as a Recurrent Neural Network (RNN) policy.

2.6.3.1. Model Agnostic Meta Learning [3]. Model Agnostic Meta Learning was introduced by Finn *et al.* [3] where the objective is to derive a meta-policy capable of adapting to a multitude of tasks after a few gradient steps in the target environment. Initially, a batch of tasks are sampled from the source task distribution. Then using the meta policy parameters we sample trajectories from each task from the batch. The following Equation 2.47 shows the loss function used to compute gradient descent with respect to the meta policy for each task.

$$\mathscr{L}_{\mathscr{T}_i}\left(f_\phi\right) = -\mathbb{E}_{\mathbf{s}_t,\mathbf{a}_t\sim f_\phi,q_{\mathscr{T}_i}}\left[\sum_{t=1}^{H} R_i\left(\mathbf{s}_t,\mathbf{a}_t\right)\right] \tag{2.47}$$

Meta policy parameters are adapted for each task by mini-batch gradient descent. Subsequently, trajectories are sampled from the corresponding tasks once more using the adapted parameters. Each sample from the sampled batch of tasks are used to update the meta policy parameter as seen in Equation 2.48.

$$\theta \leftarrow \theta - \beta\nabla_\theta \sum_{T\sim p(\mathscr{T})} \mathscr{L}_{\mathscr{T}_i}\left(f_{\theta-\alpha\nabla_\theta\mathscr{L}_{\mathscr{T}_i}(f_\theta)}\right) \tag{2.48}$$

In order to take a gradient step with respect to the meta policy, second order gradients should be computed to perform gradient descent on the samples generated by adapted policies. Practical implementations include stopping the second order gradient flow and treating the update on the meta policy as a constant during meta optimization due to concerns related to computation.

In the reinforcement learning setting, Finn *et al.* [3] carried out experiments using the Model-Agnostic Meta-Learning (MAML) model on a half cheetah and an ant in MuJoCo simulator [18]. Both agents are trained with tasks that required them to run in 2 opposite directions. In the test environment they are required to run at a direction seen in the training phase without knowing the direction the task is formulated for. The average returns yielded using the meta policy have high variance at the asymptote of the learning curve thus they've used the best performing meta policy parameters instead of the parameters yielded from the last iteration. After one gradient step in the test environment, the agent was able to adapt to the given task and improve. Similar to that, goal velocity environments are introduced where the velocity objective is sampled from a uniform distribution between $[0.0, 2.0]$ for the half cheetah and $[0.0, 3.0]$ for the ant. At test time, the agent is expected to run at a given velocity sampled from the distribution seen at training time. In both of these cases, the loss functions of the tasks are different but the domains and the essence of locomotion are alike.

2.6.3.2. Hierarchical Neural Network Structures. Benefits of hierarchical neural network structures are examined in Meta-Learning for Shared Hierarchies (MLSH) and Option Generative Adversarial Networks (OptionGAN) [63]. [64] The structure consists of a higher level master policy parametrized by θ and a set of subpolicies where each sub-policy parameter is denoted as ϕ_i. The master policy performs classification on the set of sub-policies at its output layer based on the observations from the environment. Right after a task is sampled from the source task distribution the warmup period begins where only the master policy is optimized. In the second phase, a joint update period occurs where master policy and sub-policy parameters are updated. Frans *et al.* [63] have conducted similar random direction experiments similar to MAML where the tasks is to make ant agent move either up or right. Furthermore, they experimented with a humanoid by designing a task that required

the humanoid to both walk and crawl. When the loss functions of the tasks differ a set of expert sub-policies and a higher level task classifier network that recognizes the context of the task is a promising neural network architecture. In fact this architecture is embedded in the algorithm architecture in Option Generative Adversarial Networks (OptionGAN). [64]

2.6.4. Meta Transfer Learning

Learning to transfer learn is a recent research area proposed by Wei *et al.* [57] with the objective of transfering what to transfer. They have parametrized the transfer learning algorithms used in source and target task pairs to deduce the maximizing transfer learning algorithm parameters for a novel source and target task pair. First, they learn a function mapping from the source, target task pairs and transfer learning algorithm parameters as input to performance improvement. At the testing phase, they intend to discover the transfer learning parameters that maximize the derived function.

2.7. Transfer Learning Evaluation Structure

Evaluation of generalization in deep reinforcement learning is a recently popularized research area that spurred from the need for accurate comparison of the novel algorithms [16] [13]. Cobbe *et al.* [13] created a CoinRun environment game with different test and training environments to construct a benchmark for generalization. In CoinRun the agent is trained on the different number of training levels for a 256M timesteps with PPO and the relation between level numbers and the overfitting are analyzed.

Concerns over the reproducibility and evaluation of deep reinforcement learning algorithms have also been brought up by Henderson *et al.* [65] and the results of poor hyperparameter selection and neural network architectures were analyzed. In transfer learning, these aspects are even trickier because the training performance is not informative on the generalization capability thus sampling in the target environment is required.

The testing and the training environment are usually exactly the same in the field of deep reinforcement learning. [16] In contrast, the data is composed of a training set, a vali-

dation set and a test set in supervised learning problems. Cross-validation is used for hyperparameter tuning on the training set. The training set is divided into a predetermined number of groups and for each hyperparameter search iteration, each group should be used once as the test environment while the rest are combined for training. After all the hyperparameters are used, the algorithm is ready to be tested on the unseen test set.

Recent research in the field of transfer reinforcement learning increasingly utilize the actor-critic policy gradient algorithms to build novel algorithms through the incorporation of a similar set of hyperparameters. [31] [55] [56] [66] [65] [65] [67] [3] However, when neural networks are involved skipping the hyperparameter tuning would bring about low proposed algorithm performance or even worse wrong comparisons thus hyperparameter tuning is the building block in transfer reinforcement learning. More importantly, source task performance is an inadequate assessment of the generalization capacity.

2.8. Bipedal Locomotion

Robust bipedal locomotion for a humanoid robot is still an open area for research mostly by the challenges imposed by the altering dynamics of the environment and the agent. Increasing the degrees of freedom of the humanoid robot allows the robot to perform more elaborate actions yet introduces more complexity to its anthropomorphic mechanism.

The bipedal locomotion problem without learning consists of a trajectory planning and an inverse kinematics phase [68]. First, the optimal stable trajectory should be determined for single and double support phases. Then, the inverse kinematics problem where the arrangement of the joints in the kinematic chain should be addressed to adjust the position of a rigid body to the targets imposed by the trajectory [69]. On frictional surfaces, *Zero Moment Point* criterion is a bottom-up approach to bipedal locomotion that enforces the net moment of horizontal components of inertial and gravitational forces to be zero to avoid slipping [70].

2.8.1. Bipedal Locomotion for Cassie

In this section, we will review two different approaches to bipedal locomotion on a bipedal robot named Cassie designed solely for walking and running by Agility Robotics [71].

2.8.1.1. Fast Online Trajectory Optimization for the Bipedal Robot Cassie [4]. Differential dynamic programming for unconstrained control and constrained optimization problems are techniques that are widely used to solve the trajectory optimization problem. Both of these methods have a high computational cost. Apgar *et al.* [4] suggest using a simplified model to reduce the computational costs. For instance, a spring-loaded inverted pendulum is a surrogate model that simulates external dynamics involved in the human bipedal locomotion namely ground reaction normal force. In fact, they apply the surrogate model in the multi-step forward motion case. As an example of this approach, utilizing the foot location to estimate the center of mass is given. Conversely, for the Operational Space Control (OSC) in order to output the motor torque commands, the system needs to be aware of the full dynamics model of the agent.

Their proposed system consists of two main algorithms namely the planner and the OSC. The goal of the robot is to reach the final pose while conforming to the given contact schedule. The step time and the duration of both feet on the ground phase constitute the contact schedule and there is a perfect oscillation between one foot and both feet on the ground phase. Planner gets the goal position, state of the robot and the contact schedule as the input and outputs body and foot trajectories. OSC processes the trajectory data and outputs motor torque commands for the robot to execute. The resulting state of the robot in the environment is provided as feedback to both planner and OSC. The computation of the actions that should be taken by the robot takes longer than the computation of the planner, therefore, the planner has more time to optimize the trajectory data which can be fed to the OSC system. Considering that the OSC part of the loop is constant by using the latest state of art techniques it is sensible to focus on ways of improving the planner for better optimization.

2.8.1.2. Feedback Control For Cassie With Deep Reinforcement Learning [5]. Model-free approaches tends to look more promising than the model-based approaches because all the constraints and non-linearities during the optimization phase cannot be taken into account during optimal control computations. Xie *et al.* [5] used MuJoCo simulator to implement the PPO Algorithm [31] with feedback control to initiate bipedal locomotion in Cassie.

Feedback mechanisms are model-based approaches designed to compute the trajectory of the robot. Usually, the eventual tradeoff of computation cost, the feedback mechanism does not reflect the system accurately, potentially yielding suboptimal results. The Cassie has fewer actuators than the degree of freedoms which makes this a harder problem for learning because it is an underactuated robot [5].

2.8.2. Center of Mass

The vertical and horizontal displacement of the center of mass during bipedal locomotion provides crucial insight into the anthropomorphic motion [72]. Dynamic balance and the optimal trajectory of a humanoid depend on the center of mass. Research on finding the route of the agent's center of mass during bipedal locomotion is prevailing. For instance, Carpentier *et al.* [72] analyzed the motion of a male subject walking on a flat surface to illustrate the center of mass's trajectory. They further prove that the center of mass's movement is characterized as a curtate cycloid while walking.

2.8.3. Friction

The friction between the feet of the robot and the surface are divided into two categories for bipedal locomotion: static and kinetic friction [73]. If the robot is slipping while walking kinetic friction coefficient is used whereas the static friction coefficient is used for non-slipping cases.

There are two main types of tangential friction when moving forward: landing your foot on the ground which is basically applying forward force to the ground by the heel and taking off your feet from the ground by the forefoot palm [73]. Friction acts in the opposite

direction by the ground in each of these cases.

When two surfaces contact, intermolecular forces occur which are named generally as *frictional forces*. Contact of uneven surfaces creates a higher frictional force because at the molecular level they have higher attraction among each other. In our case, the normal force is perpendicular to the contact surface and prevents the humanoid from sinking through the floor. The normal force is canceled out by the humanoid's weight.

Tangential forces are parallel to the contact surface and act in the opposite direction to the applied force [73]. Each force applied to the materials at the contact point cause the molecules at the surface to be disturbed and relocated. This decreases the intermolecular connection between the surfaces and allows the object to slide. The static friction force is calculated by multiplying the coefficient of static friction with the Normal force. If the force applied to the object is more than the static friction the object starts to move and the kinetic friction force starts acting on the object which calculated same as static friction force but has a different coefficient which is usually less than the static friction coefficient because the intermolecular attraction between moving surfaces is less.

3. METHODOLOGY AND IMPLEMENTATION

Deep reinforcement learning methods are phenomenal in learning the task at hand with sufficient training but they often fail to generalize to an ensemble of different tasks. More importantly, training a robot for each possible real-life scenario is impractical thus if we strive for a human level performance we should draw inspiration from how humans adapt to unforeseen challenges.

In this chapter, we elaborate on our propositions on the policy gradient, actor-critic and adversarial learning algorithms to illustrate the methods we will use to increase the robustness of the robots and introduce policy buffer evaluation. Subsequently, we present the set of environments that we designed to compare and evaluate the algorithms we propose. Particularly, inter-robot learning transfer, high tangential frictional ground and the delivery robot environments are applicable to real life scenarios.

3.1. Methodology

In this section, the algorithms used in Chapter 5 are discussed in detail to build a solid background for our design decisions. First, we introduce Policy Buffer in Section 3.1.1 for transfer learning evaluation. We provide intuition on the techniques we suggest to increase generalization capacity of the policy, namely Strict Clipping Proximal Policy Optimization (SC-PPO), Average Consecutive Critic Robust Adversarial Reinforcement Learning (ACC-RARL) and Asymmetric Entropy Bonus. Essentially, we suggest regularization techniques for PPO [31] and discuss the variants we suggest for RARL [15].

3.1.1. Policy Buffer

Walking in different environments may require different policies. Policies that are trained with different hyperparameters show different walking patterns at each optimization iteration. We propose a policy buffer to store these different policies trained on the same source task environment with the same loss function. We will show in our experiments

that it is possible to extract a comprehensive set of policies representative of distinct control patterns just from one source task environment.

Different snapshots of the policy network parameters taken during training in the source task environment perform dramatically different in the target task environment. Over-training in the source task environment causes overfitting which leads to a worse result on the target environments as the distance between the target environment and the source task environment increase in the environment parameter space. In order to discover the most suitable policy network for the target task, we first take snapshots of the policy network at each sampling iteration at predetermined intervals. The snapshots of the policies trained with different hyperparameters in the source task environment will be saved in the policy buffer.

We will further prove that the performance of the policy on the source task does not give sufficient insight into the average return in the test environment. Considering the striking difference between the policy iterations' ability to transfer, evaluations at different iterations are necessary to compare various methods. Taking the resulting parameters after a constant number of training iterations does not constitute a valid comparison because full scope of the algorithms' generalization capacity is omitted.

The iteration of the policy that will be used in the target environment is a hyperparameter that should not be ignored. Sampling from the target environment during training is used in several transfer learning algorithms to demonstrate the use cases where the real world is the target environment where the sample set is restricted and the simulator is the source environment where the sampling phase is only restricted by the computational resources [1] [17].

In scenarios where the difference between source task and target task can be parametrized, we propose that designing a *surrogate validation task* will prove to be informative in finding the interval of policy iterations with the highest adaptability to an alike target task. The *surrogate validation task* should have parameters closer to the target environment if possible and just a few samples from the *surrogate validation task* would be an adequate starting point for determining the policies that should be given priority during target environment sampling. *Surrogate validation task*'s purpose is analogous to hyperparameter tuning

using a validation set in supervised learning algorithms.

3.1.2. Regularization via PPO Hyperparameter Tuning

Policy gradient algorithms are the building blocks of the recent transfer learning and generalization algorithms. The choice of the clipping hyperparameter of Proximal Policy Optimization is crucial when using the algorithm as a transfer learning benchmark. *Open AI Baselines* framework and most of the literature use the clip parameter of 0.2 for continuous control tasks [30] [31] [55] [56] [66] [65]. In addition to that, the clipping parameter is discounted using a learning rate multiplier in *Open AI Baselines* framework to encourage swift reaching to the asymptote for all continuous control tasks in MuJoCo. In our experiments, we have found out that decaying the clipping parameter decreases the asymptotic performance of the algorithm in the Humanoid environment and submitted it as an issue. Decaying the clipping parameter linearly by the learning rate After our suggestion, the clipping hyperparameter annealing in the *ppo1* algorithm has been omitted in the *Open AI Baselines* framework [30].

We hypothesize that in a transfer learning setting, *strict clipping* can be used to discard the MDP tuples that lead to overfitting. In this thesis, we propose *strict clipping* as a regularization technique for PPO to decrease variance introduced by the source task-specific samples. In order to construct a fair comparison between state of art transfer learning algorithms, various lower values of clipping parameters are analyzed. *Strict clipping* is performed by decreasing the clipping parameter to unconventional values, for instance as low as 0.01. In Chapter 5 we prove that this method is a competitive benchmark for the transfer learning algorithms.

Increasing the batch size can also be used to increase the robustness of the resulting policy in PPO for transfer learning. More frequent gradient steps will be taken when the batch size is small and the resulting noise introduced will decrease the generalization capacity of the resulting policy. The average loss of the all the tuples in the batch is used to compute the gradient so the movement in a parameter so a larger batch size won't necessarily amount to a larger gradient step. Accordingly, each tuple's impact on the resulting gradient

step will decrease with increased batch size. Strict clipping to unconventional values discards the overly optimistic tuples directly whereas increasing batch size results in reduced variance and overfitting.

3.1.3. Robust Adversarial Reinforcement Learning Variations

Initially, the policy's generalization capacity should be maximized when transferring the learning from a source task to a target task. Training the policy aiming general robustness for a range of possible unknown scenarios is one way of achieving a successful initial test performance. Adversarial scenarios are inspired by the success of domain randomization during training. Introducing an adversary during training whose sole purpose is to destabilize the agent using multidimensional forces in Robust Adversarial Reinforcement Learning (RARL) have proven successful results in continuous control tasks. [15]

There is a continuous competition for exactly opposite rewards in RARL depending on the destabilizing capability of the adversary. For instance, an adversary policy with a two dimensional output has a restricted power due to low dimensional action space and might have a hard time destabilizing a Humanoid robot with 17-dimensional action space during training. However, an adversary with a 27-dimensional action space that applies a 3-dimensional force to each body component of the protagonist humanoid might even hinder the policy from reaching convergence. Overfitting is at utmost importance in finding the policy with the highest generalization capability considering the tidal nature of the algorithm. Thus we will compare and analyze the variants of RARL using the Transfer Learning Evaluation Structure we've proposed in the previous section and extract the best performing policies enhancing the capability of the algorithm. Because we will use the technique of policy buffer in our comparisons each RARL algorithm variant's real generalization capacity will be presented.

3.1.3.1. Critic Network Architectures. In order to analyze the effect of different critic architectures on the generalization capacity of the policy we will compare three different critic architectures: single critic networks in Shared Critic Robust Adversarial Reinforcement Learning (SC-RARL) [8],separate double critic networks that is used in RARL [10]

and our proposition Average Consecutive Critic Robust Adversarial Reinforcement Learning (ACC-RARL). Critic networks are inherently function approximators so they are vulnerable to overfitting as well as the actor networks.

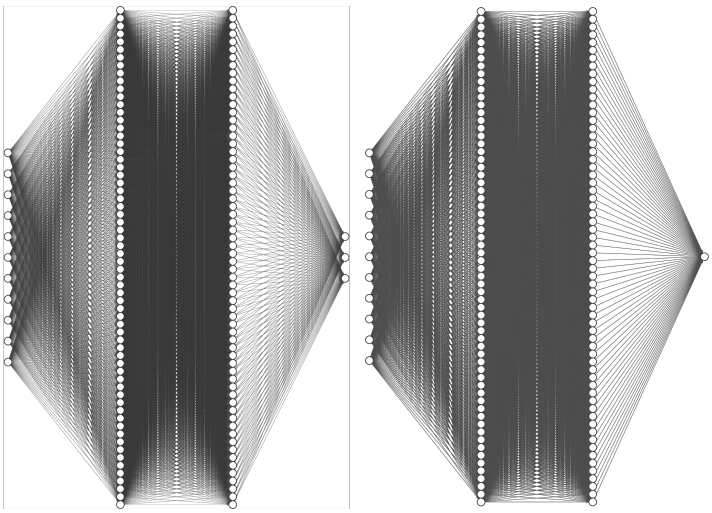

Figure 3.1. Protagonist policy and critic networks in Shared Critic Robust Adversarial Reinforcement Learning (SC-RARL) [8], neural network figures are generated using [9]

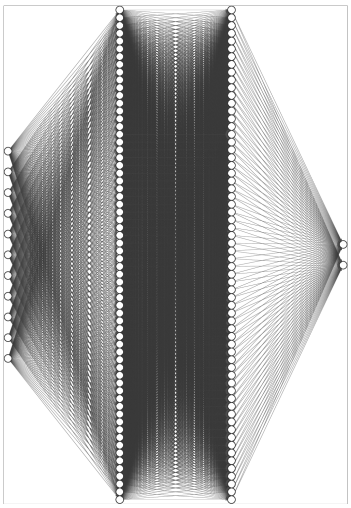

Figure 3.2. Adversary policy network in Shared Critic Robust Adversarial Reinforcement Learning (SC-RARL) [8], neural network figures are generated using [9]

Figure 3.1 shows the protagonist policy and critic network pairs for Shared Critic Robust Adversarial Reinforcement Learning (SC-RARL) where the policy network output layer size is 3 denoting the number of actions hopper can take. In Figure 3.2 the adversary

policy network is provided where each neuron in the output layer stands for one of the 2-dimensional force applied to the hopper's heel. The adversary does not have its own critic network and uses the negative of the protagonist's critic network output for target value computation. In SC-RARL, adversary policy is updated with $A^{GAE(\gamma,\lambda)}$ computed with the previously updated critic network of the protagonist but during adversary policy's optimization, no critic network is optimized. Since the advantage function computation for the adversary is exactly negative of the protagonist this architecture type is meaningful but the protagonist's critic network is only updated by the rewards gained in the protagonist's optimization phase. Fewer samples and optimization iterations might act as a regularization but we should also explore ways of using the rewards from other sampling iteration without overfitting.

In contrast to SC-RARL, in RARL, each critic network is separate from each other and at each global iteration protagonist and adversary policies are updated with the $A^{GAE(\gamma,\lambda)}$ computed using the rewards from different trajectories and separate randomly initialized critic network outputs. In SC-RARL, the shared critic is updated by the rewards gained during protagonist optimization phase and in RARL each critic is updated only by the rewards gained during its corresponding policy's optimization phase. The total number of samples used to update the critic networks are the same for RARL and SC-RARL.

In this thesis, we propose a third architecture named Average Consecutive Critic Robust Adversarial Reinforcement Learning (ACC-RARL) that computes advantage estimate using the mean of both critic's output but consecutively and separately optimizes each critic network along with their corresponding policies. Figure 3.5 shows the protagonist's policy, critic and adversary's critic network. The same protagonist's network is seen in the grouping made in Figure 3.6 that also includes the adversary's policy and critic. By this method, we aim to decrease overfitting via using double critic networks with different random initializations and restrict the catastrophic movement of the critic in parameter space by including the output of the previously updated critic in advantage estimation. The δ_t values for both adversary and protagonist in ACC-RARL algorithm are shown in Equation 3.1.

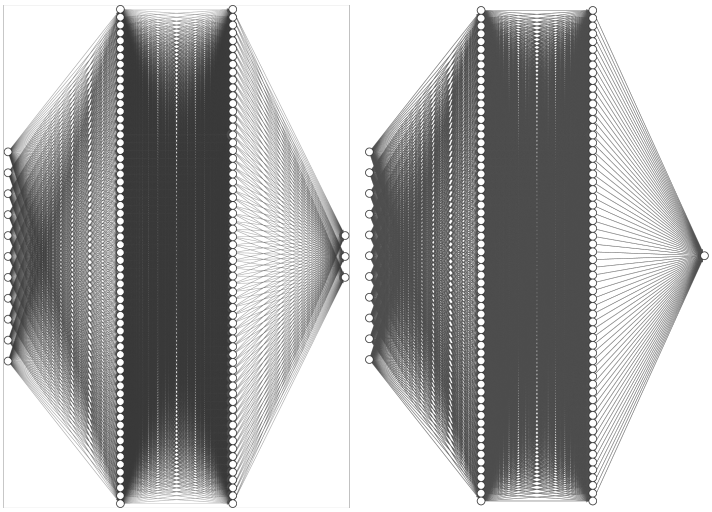

Figure 3.3. Protagonist policy and critic networks in Robust Adversarial Reinforcement Learning (RARL) [10] and Average Consecutive Critic Robust Adversarial Reinforcement Learning (ACC-RARL), neural network figures are generated using [9]

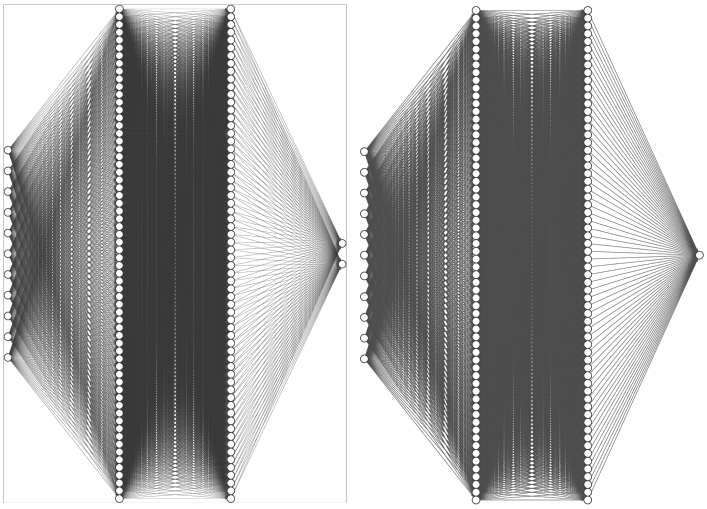

Figure 3.4. Adversary policy and critic networks in Robust Adversarial Reinforcement Learning (RARL) [10] and Average Consecutive Critic Robust Adversarial Reinforcement Learning (ACC-RARL), neural network figures are generated using [9]

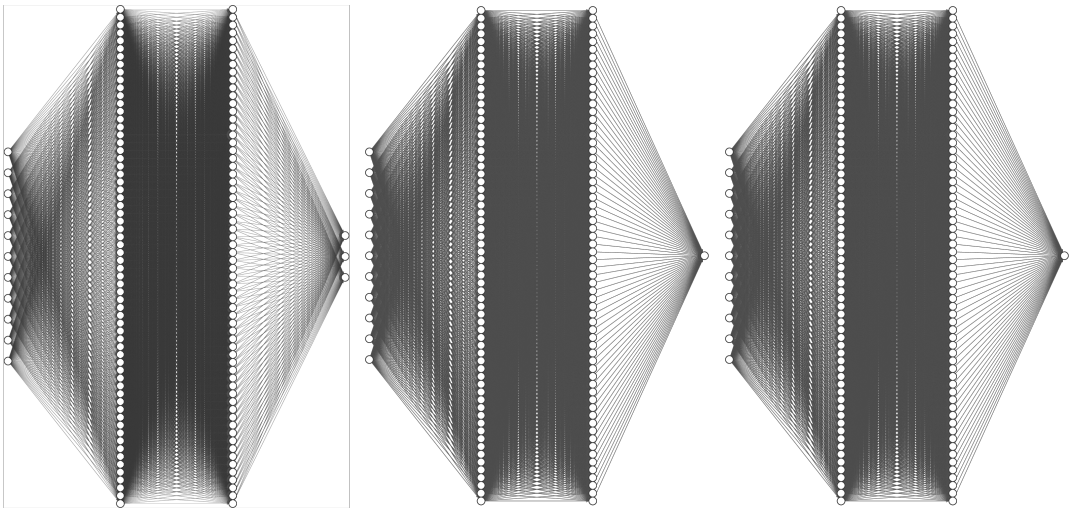

Figure 3.5. Protagonist policy, critic networks and adversary critic network in Average Consecutive Critic Robust Adversarial Reinforcement Learning (ACC-RARL), neural network figures are generated using [9]

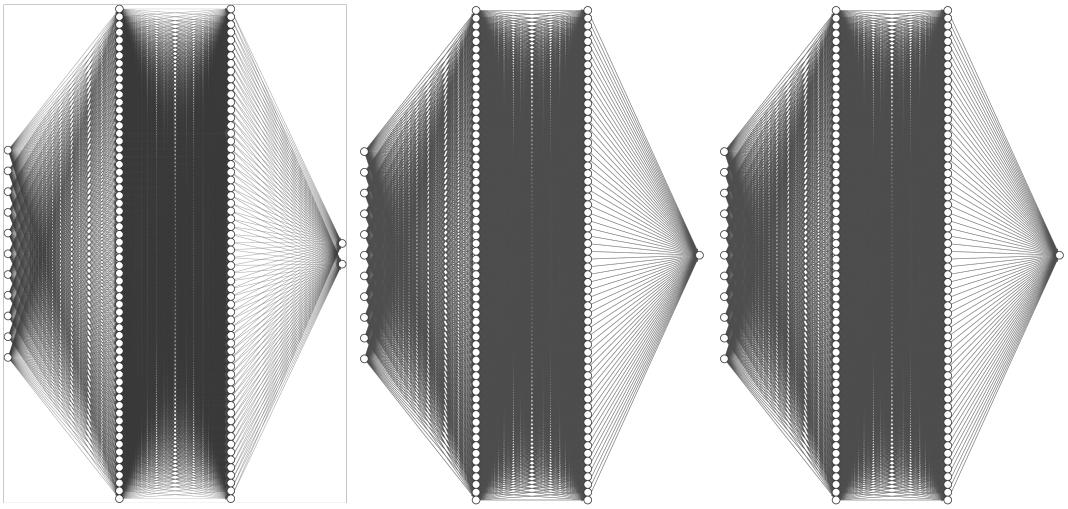

Figure 3.6. Adversary policy, critic networks and protagonist critic network in Average Consecutive Critic Robust Adversarial Reinforcement Learning (ACC-RARL), neural network figures are generated using [9]

$$\begin{aligned}\delta_t^{protagonist} = (-V_{protagonist}(s_t) + V_{adversary}(s_t))/2 + \\ r_t + \gamma(V_{protagonist}(s_{t+1}) - V_{adversary}(s_{t+1}))/2 \\ \delta_t^{adversary} = -\delta_t^{protagonist}\end{aligned} \tag{3.1}$$

The parallel implementation of the proposed ACC-RARL algorithm pseudocode is outlined in Figure 3.1.3.1.

In ACC-RARL, each critic is updated by the average of two sequentially updated critic outputs and the rewards gained during their corresponding policy's optimization phase. In RARL, policies are not informed of each other's critic output but they rely on the similarity of 2 consecutive sets of rewards accumulated by a different group of agents. On the other hand, in ACC-RARL, both critics are updated by each others' critic outputs with the rewards gained during their own optimization phase. Updating the critic function more frequently than the policies would increase overfitting so by ACC-RARL we aim to increase generalization capacity by encouraging them to optimize with different reward batches while considering each other's critic outputs. Thus, each critic network observes all the rewards sampled in the environment but assigns more weight to the rewards gained during its own optimization phase.

Figures 3.1,3.2,3.3,3.4, 3.5, 3.6 are generated using the neural network generator developed by LeNail [9]. All neural networks depicted have 2 hidden layers with size 64 and input layer of size 11. All policy networks have a seperate logarithm of standard deviations vector that has the same size as the corresponding policy network's output. These vectors are optimized simultaneously with the policy networks as described in the implementation detailed in Section 2.5.

3.1.3.2. Entropy Bonus. Entropy bonus is used to aid in exploration by rewarding the variance in the multivariate Gaussian distribution of action probabilities by a coefficient c_2 given in Equation 3.2.

Require α : step size hyperparameter ; i : process ; n : number of parallel processes

Initialize the fully connected layer weights using standard normal distribution with $\sigma = 1.0$, the final fully connected layer weights with $\sigma = 0.01$ in θ^{pro} and θ^{adv} ;

for $k = 1$ to $Iteratons$ **do**

  for $j = 1$ to $Iteratons_{pro}$ **do**

    Sample a trajectory $\tau = (s_1, a_1^{pro}, a_1^{adv}, r_1, d_1, ...s_H)$ using θ^{pro} and θ^{adv} for all i

    Compute generalized advantage estimates $\hat{A}^{pro} = \hat{A}^{pro}_{s_1,a_1}...\hat{A}^{pro}_{s_H,a_H}$ by θ^{pro}_{critic},θ^{adv}_{critic} and τ, using Equations 2.35,3.1

    $\theta^{pro}_{old} \Leftarrow \theta^{pro}$;

    for $e = 1$ to $Epochs$ **do**

      for $batch = 1$ to $BatchSize$ **do**

        Compute the $\mathscr{L}^{pro}_{batch_i}$ using Equation2.40;

        $\theta^{pro} \Leftarrow \theta^{\mathbf{pro}} - \alpha\nabla_{\theta^{\mathbf{pro}}}\frac{1}{n}\sum_{process_i}\mathscr{L}^{pro}_{batch}$;

      end for

    end for

  end for

  for $j = 1$ to $Iteratons_{adv}$ **do**

    Sample a trajectory $\tau = (s_1, a_1^{pro}, a_1^{adv}, r_1, d_1...s_H)$ using θ^{pro} and θ^{adv} for all i

    Compute generalized advantage estimates $\hat{A}^{adv} = \hat{A}^{adv}_{s_1,a_1}...\hat{A}^{adv}_{s_H,a_H}$ using θ^{pro}_{critic},θ^{adv}_{critic} and τ using Equations 2.35,3.1

    $\theta^{adv}_{old} \Leftarrow \theta^{adv}$;

    for $e = 1$ to $Epochs$ **do**

      for $batch = 1$ to $BatchSize$ **do**

        Compute the $\mathscr{L}^{adv}_{batch_i}$ using Equation 2.40;

        $\theta^{adv} \Leftarrow \theta^{\mathbf{adv}} - \alpha\nabla_{\theta^{\mathbf{adv}}}\frac{1}{n}\sum_{process_i}\mathscr{L}^{adv}_{batch}$;

      end for

    end for

  end for

end for

Figure 3.7. ACC-RARL Algorithm.

$$c_2 S[\pi_\theta](s_t) = c_2 * \sum(log(\sigma) + 0.5 * log(2\pi e)) \tag{3.2}$$

Although entropy bonus is a part of PPO total loss function we did not include it in the loss function because it decreased the performance of the policy. However, entropy bonus should be a part of our comparisons in adversarial training scenarios and will introduce asymmetry thus we will compare its inclusion in the protagonist and the adversary's loss function separately. Our hypothesis is motivated by the works on competitive and adversarial environments that suggest that hard environments might hinder learning [55] [17]. Subsequently impelling the protagonist to explore might prove to be beneficial in a randomized scenario although it decreases the performance in environments with no adversaries. Similarly, adding an entropy bonus to adversary's loss function might prepare the protagonist for a wider range of target tasks by extended domain randomization. Correspondingly the entropy bonus might also hinder adversary to reach peak performance and simplify the environment. The entropy coefficient we will use is 0.001 for adversary and protagonist loss function. All in all, two different trained policies will be added to the policy buffer for each critic architecture.

3.1.3.3. Curriculum Learning. Curriculum Learning is a recent branch in transfer learning that focuses on discovering the optimal arrangement of a group of source tasks to perform better on the target task. In order to excel in complex tasks, humans follow specifically designed curricula in higher education [74]. A more personalized curriculum will most certainly lead to a more successful result for humans so we hypothesize that it will prove beneficial for the learning of robots.

Similar to [55] [17], we will construct a random curriculum by randomizing the adversary policy iterations during training. In our experiments, we will compare the performance of protagonist policies trained with adversaries randomly chosen from different last T iterations. We will use uniform sampling from a restricted last T adversary policy set for this experiment because the Shioya *et al.*'s [17]'s proposition is computationally intensive and

the first method corresponds to Shioya *et al.*'s [17] "mean" method. In this thesis, we introduce RARL PS-Curriculum where we first train the policy with the adversary in the source task and record all the adversary policy snapshots at each iteration to policy storage. Then at each iteration based on the sampling from the Uniform $(\delta v, v)$ distribution, the adversary from the policy storage will be loaded during sampling. This method is included in our experiments because the adversaries loaded from and recorded to the buffer during curriculum training become less capable and more inconsistent as the training progress and δ decreases.

3.2. Implementation

In this thesis, in order to demonstrate the generalization capability of our resulting policy, we introduce a set of transfer learning benchmarks on the *Hopper-v2* and *Humanoid-v2* environment in *MuJoCo*. Reproducibility of reinforcement learning algorithms is challenging due to colossal dependability on hyperparameters and implementation details. Consequently, we will provide detailed information on experimental design and hyperparameters.

We will be using *tanh* activation functions for the hidden layers of the policy and the critic network. *tanh* is a much better alternative to the *sigmoid* function because its mean is centered around 0. The policy outputs the means of a multivariate Gaussian distribution. Simultaneously a vector of logarithm of standard deviations representing the diagonal covariance matrix will be optimized as a separate variable tensor. If the policy is stochastic then the action vector will either be the mean or a random value sampled from the resulting multivariate Gaussian distribution. We will use a stochastic policy for the training and a deterministic policy for the testing environment.

In the [30] implementation for humanoid, 16 parallel processes was used to sample from structurally same environments concurrently with different random seeds during training time. Considering the hardware limitations and for the sake of consistent comparison we will use 16 number of processes in our experiments. It is important to note that the number of parallel evironments for sampling is a hyperparameter in the implementation because each process uses different seeds for batch shuffling. The MuJoCo environments are stochastic and the initial state of the agent is sampled from a random distribution thus each environment

is initialized with a different initial seed. Similarly, the policy and critic network parameters are initialized using separate random seeds at each process but before training phase, all the policies are synchronized to take gradient with respect to the same parameters for an update so the initialization randomness doesn't effect the policies in PPO algorithm. Each MDP tuple in the horizon of 2048 will be shuffled randomly before the beginning of the optimization at each iteration. The resulting gradient step will be calculated for each batch by averaging the gradients from all processes. The process of shuffling and optimizing overall batches will be repeated for 10 epochs before sampling the next trajectory from the environment [30].

The adversarial architecture consists of a total of 2 actor-critic neural network pairs and 2 old policy networks. The inputs of each neural network are the same however the outputs of the adversary policy network differ from each other. Because *Hopper-v2* is a 2-dimensional environment 2 output neurons are specified to represent each force applied to the *geom* component of the robot. The environment simulates actions from both the protagonist and adversary at each iteration and outputs the reward based on the next derived position of the agent.

3.2.1. Environment Variation

Friction is one of the environmental dynamics that have a substantial effect on bipedal locomotion. In the *MuJoCo* simulator [18] the friction component of the floor consists of three parameters which are *tangential*, *torsional* and *rolling*. When two surfaces come into contact, the friction of the geometric surface that is higher in the hierarchy is used. If they share the same hierarchy the maximum of the friction parameters are used. The tangential coefficient of the friction is also valid across the surface.

Altering the gravity for each target task for the Humanoid and Hopper was one of the Henderson *et al.* [14] designed multi-task environments in line with *OpenAI*'s request for research [75]. For this implementation setting, we chose to implement one task transfer learning from an environment with ground tangential friction 1 to an environment with ground tangential friction 3 and 3.5. In [64] 4 target tasks are created with 0.25 increments of the environment gravity more specifically $0.50G, 0.75G, 1.25G$ and $1.5G$ using MuJoCo

simulator where G=-9.81. In our gravity experiments, we will use $0.50G, 1.5G$ and $1.75G$ as the target environment gravities to benchmark our propositions.

The environments are diverse enough to pose a challenging transfer learning problem because the representation of a humanoid trained for the source environment friction with parameters optimized for source task can't walk in the target environment. No modifications will be made to the loss functions for these environmental conditions.

3.2.2. Morphological Variation

Transferring among morphologically different robots with different limb sizes or torso masses have been a popular multi-task learning benchmark [14] [1] [15]. In the first section, we will test the generalization capability of our trained policy on a hopper with differing torso masses to compare our proposition to other algorithms. In Section 5.3 we will introduce two new target environments: a tall, heavy humanoid and a short, lightweight humanoid.

4. ENVIRONMENT SETUP

This chapter provides information on the software and hardware used to carry out our implementations. The same environmental setup is also used to reproduce results from [31].

4.1. Software Setup

Recent developments in the field of deep reinforcement learning commonly use the open source toolkit named *Gym* developed by OpenAI [3, 55, 76, 77]. *Gym* toolkit offers *Algorithms*, *Atari*, *Box2D*, *Classic Control*, *Toy Text* environments that are readily available within the toolkit as well as *MuJoCo* and *Robotics* environments that should be integrated with the *MuJoCo* physics engine [18].

MuJoCo is widely used by the research community due to its superior accuracy and stability among its counterparts [78]. The action spaces of the MuJoCo Humanoid and Hopper environments are shown in Table 4.1 and Table 4.2 [76]. Continuous action and state spaces are demonstrated as vectors where an element of the vector is assigned a value between a specified interval in the environment. The environment also allows restricting the action space to allow flexibility for researchers to demonstrate a malfunctioning joint. In particular this case was implemented as a benchmark for adaptability of meta learning in [77].

Tensorflow version 1.10 for CPU is used to construct the deep neural networks used in our experiments [79]. Learning curves are plotted using *Tensorboard* visualization toolkit that is integrated with the *Tensorflow* framework. We use the *Linux* version and have installed it in our *Python 3* virtual environment.

4.2. Hardware Setup

After experimenting with the continuous control tasks we have found out that using multiple processing with CPU provided faster results than the GPU alternative. Subsequently all of our experiments are carried out by using 16 parallel processes. In order to avoid

Table 4.1. Humanoid Actions

	Name	**Gear**
0	abdomen y	100
1	abdomen z	100
2	abdomen x	100
3	right hip x	100
4	right hip z	100
5	right hip y	300
6	right knee	200
7	left hip x	100
8	left hip z	100
9	left hip y	300
10	left knee	200
11	right shoulder1	25
12	right shoulder2	25
13	right elbow	25
14	left shoulder1	25
15	left shoulder2	25
16	left elbow	25

Table 4.2. Hopper Actions

	Name	**Gear**
0	thigh joint	200
1	leg joint	200
2	foot joint	200

overheating on our local computer we conduct the meta training phases of our research on a shared remote Ubuntu 16.04 server with 20 CPU cores.

5. EXPERIMENTS AND DISCUSSION

In this chapter we will first discuss the details of the experiments conducted in the source environments in Sections 5.1,5.2, then the target environment experiments will be presented and analyzed using the methodology detailed in Chapter 3. All of the experiments are performed in the *MuJoCo* simulator [18], the results are plotted in *Matplotlib* [80].

5.1. Humanoid Robot

As detailed in Equation 5.1, the reward function in the Humanoid environment consists of

- an alive bonus,
- linear forward velocity reward,
- quadratic impact cost with lower bound 10
- quadratic control cost.

$$r_{humanoid}(s,a) = 0.25 * r_{v_{fwd}} + min(5 \cdot 10^{-7} * c_{impact}(s,a), 10) + 0.1 * c_{control}(s,a) + C_{bonus} \tag{5.1}$$

If the z coordinate of the agent's root which lays at the center of the torso is not between the interval 1 and 2, the episode terminates. The alive bonus C_{bonus} is specified as +5 for the Humanoid task which is the default value for the Gym Humanoid Environment.

$$L_t^{CLIP+VF}(\theta) = \hat{E}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) \right] \tag{5.2}$$

In Equation 5.2, the total loss function of the PPO algorithm that we will use to update actor and critic networks is given. We do not use the entropy reward in the original algorithm for PPO implementation because although we have used small entropy coefficients like 0.005, we did not see any improvement in the learning curve. Moreover, the entropy bonus is not used in PPO implementation [30] [31]. In addition to that, in the *OpenAI Baselines* framework [30], the clipping hyperparameter decays by the learning rate multiplier, we also omit that because the learning curve tends to decrease after reaching the asymptote at the later stages of training for higher state-action space environments. In the latest version of PPO algorithm in *OpenAI Baselines* framework, clipping annealing in higher dimensional environments was omitted after we submitted an issue. The algorithm moves away from the minimum in the earlier implementation, as more samples are discarded because the clipping parameter approaches 0 due to the decay.

Running Mean Standard deviation allows the mean μ and standard deviation σ of the observations to update during training. It is also important to point out that at each iteration the parameters of the value function approximator network change along with the policy network. All of the observations are standardized and clipped to range $[-5,5]$ before being fed to the network as seen in Equation 5.3 [30].

$$s = \text{clip}((s-\mu(s))/\sigma(s), -\zeta, \zeta) \tag{5.3}$$

The hyperparameter space that will be used for the PPO policy gradient algorithm for the source and target environments in this chapter is shown in Table 5.1. We propose a constant clipping schedule and strict clipping parameters for action and state spaces with higher dimensions and transfer learning scenarios.

In Figure 5.1 average episode rewards of policies trained with 4 different sets of hyperparameters are shown. 16 different seeds are used in parallel with the *OpenAI Baselines* framework [30] implementation for each process and the policy parameters of each are saved

Table 5.1. Hyperparameters

Hyperparameters	**Symbol**	**Values**					
Clipping Parameter	ε	0.01	0.025	0.05	0.1	0.2	0.3
Batch size	b	64			512		
Step Size	α	0.0001			0.0003		
Curriculum Parameter	χ	0.3			0.5		
Learning Schedule		constant			linear		
Clipping Schedule		constant			linear		
Trajectory Size	H	2048					
Discount	γ	0.99					
GAE Parameter	λ	0.95					
Adam Optimizer	β_1	0.9					
Adam Optimizer	β_2	0.999					
Number of Epochs		10					
Entropy Coefficient		0.001					
Number of Hidden Layers		2					
Hidden Layer Size		64					
Activation Function		tanh					

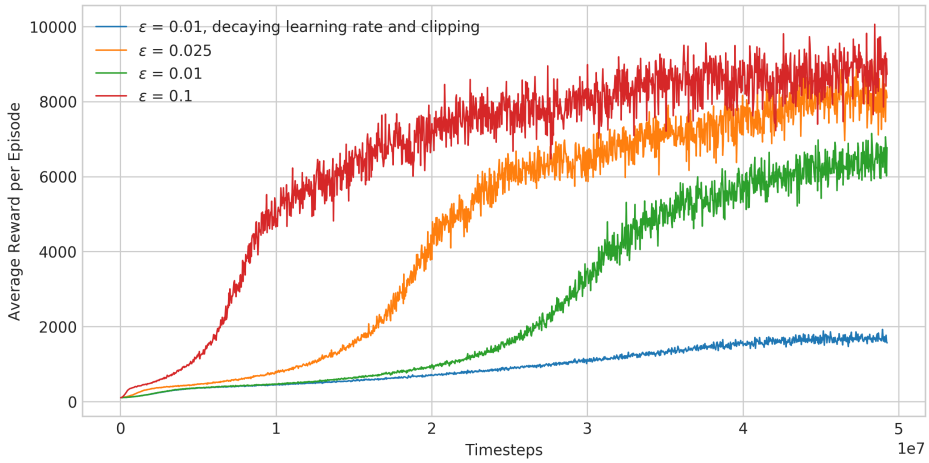

Figure 5.1. Learning curve of policies trained in the standard humanoid environment with different hyperparameters

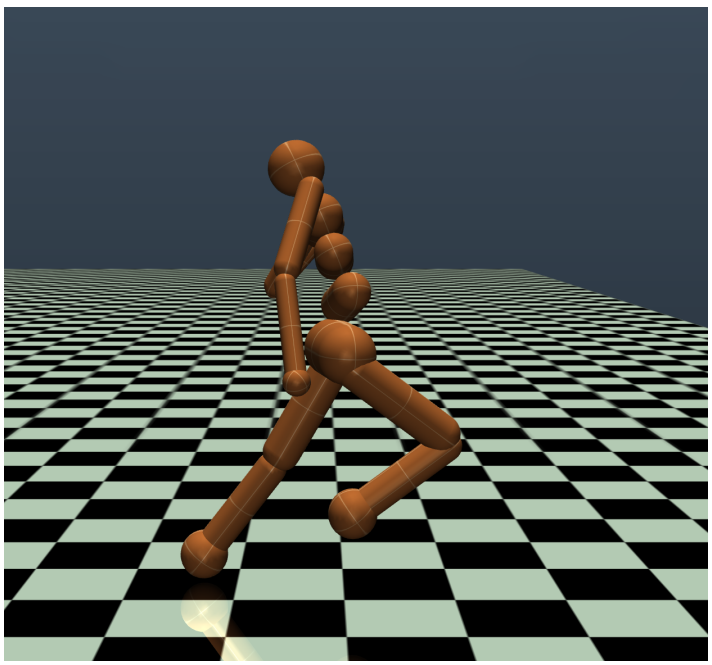

Figure 5.2. Humanoid running in source environment using the last policy trained with PPO $\varepsilon = 0.1$

at intervals of 50 iterations. Learning curve obtained with the latest PPO hyperparameters suggested in *OpenAI Baselines* framework [30] for the Humanoid environment are represented by the red curve. The learning curves for the strict clipping methods for generalization are shown with clipping hyperparameters 0.01, 0.025 and 0.01 decaying learning rate and clipping. Linearly decaying learning rate and clipping is a method used for environments with lower dimensions but we include strict clipping variations with clipping hyperparameters 0.01 and 0.025 of it in our benchmarks. In the testing phase, we will sample a trajectory of 2048 timesteps from 32 different seeded target task environments for each policy in the policy buffer.

5.2. Hopper Robot

Reward function of the hopper environment is given in Equation 5.4. Alive bonus in the Hopper environment is C_{bonus} is +1 and the termination constraints ensure that the angle between the hopper body and the ground and the hopper height are not below 0.2 and 0.7 respectively. Sum of squared actions could be regarded as a more primitive way of computing control cost in the humanoid environment.

- an alive bonus,
- linear forward velocity reward,
- sum of squared actions

$$r_{hopper}(s,a) = \ r_{v_{fwd}} - 0.001 * \sum a^2 + C_{bonus} \tag{5.4}$$

We choose to initiate 16 parallel processes with different random seeds for the Hopper environment and 1.875M timesteps of samples are collected from each uniquely seeded environment. Hyperparameters of the best performing PPO in this experimental setting are found via simple grid search. The input observations are standardized and clipped as in

Equation 5.3 before being fed into the neural networks.The best performing clipping parameter, step size and batch size in source task environment are $\alpha = 0.0003$,$\varepsilon = 0.3$ and $b = 512$. Average reward per episode of PPO and RARL with different critic architectures at source environment over a total of 30 million timesteps are shown in Figure 5.3.

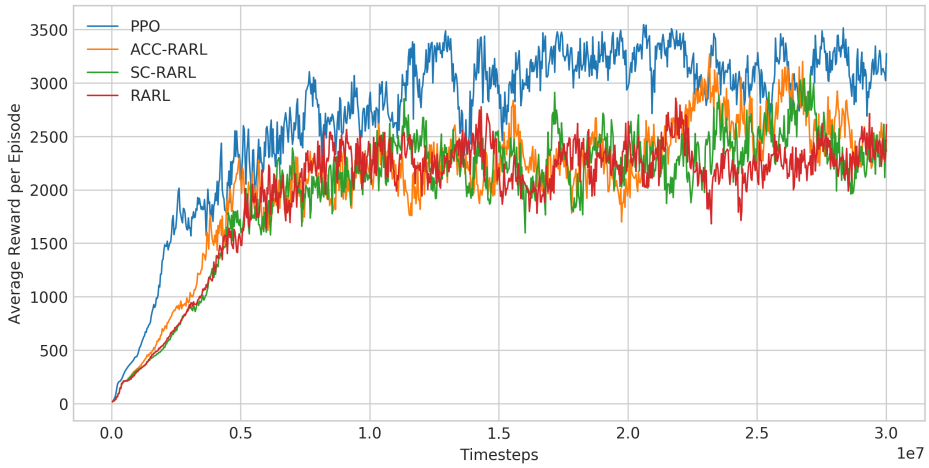

Figure 5.3. Learning curve of policies trained in the standard hopper environment

The policies trained with different variations of RARL perform worse in the source task environment, consistent with the comparison of the policies proposed in the Humanoid experiments. The protagonist policy gains fewer rewards due to domain randomization created by the adversary but a natural regularization occurs which counteracts to overfitting to the target environment. We will further prove our hypothesis by introducing a validation task in hopper morphology experiments. If an episode has not reached the termination in the horizon of 2048 timesteps it gets truncated. The shaded regions around average episode rewards as in Figure 5.4 reflect the standard deviation of the episode returns from 32 trajectories.

5.3. Morphology Experiments

5.3.1. Hopper

In these sets of morphology experiments, we will compare a total of 7 variants of adversarial training: 2 entropy bonus and 4 curriculum learning for each critic structure we have proposed in Section 3.1.3. Initially, the performance of different critic structures will

be compared: RARL [10], shared critic network (SC-RARL) [8], our proposition Average Consecutive Critic Robust Adversarial Reinforcement Learning (ACC-RARL). Next, the best performing variation of each critic structure will be analyzed. Figure 5.2 shows the morphological specifications of the standard Hopper.

Table 5.2. Source Environment

Body	**Unit Mass**
Torso	3.53429174
Thigh	3.92699082
Leg	2.71433605
Foot	5.0893801
Total Body	15.26499871

Satisfactory average reward per episode for Hopper environment in the original PPO benchmark has been found to be between 2000 and 2500 [31]. In Figure 5.3, best source task reward is found to be above 3000 where the hopper hops quickly and seamlessly. In RARL with TRPO, the torso mass range chosen for the experiments is $[2.5-4.75]$ [15], we will experiment with torso unit masses in the range$[1-8]$. $[1-6]$ unit masses will prove to be easier benchmarks thus the best performing policy for each critic structure comparison will be omitted. The right iterations of baseline policy PPO performed adequately between $[1-5]$ which suggests that domain randomization via adversary is unnecessary for some tasks and succeeding in the target environment might boil down to knowing where to stop during training or using the right iteration of policy from the policy buffer.

Algorithms trained with adversaries proposed in Section 3.1.3 are more unstable during the training phase. In our experiments, we have witnessed that wrong choice of hyperparameters destabilizes learning after convergence and the learning curve in the source environment becomes concave as the average reward per episode starts to decrease after convergence.

First, the policy buffer is created based on our suggestion in Section 3.1.1. The policy buffer will consist of snapshots of policies trained with PPO and 21 variants of RARL taken

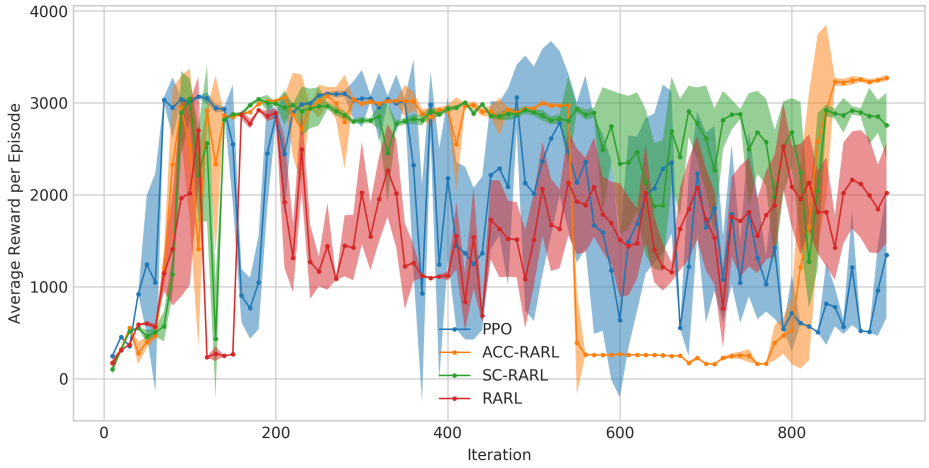

Figure 5.4. Average reward per episode at target environment with torso mass 1 of every 10 iterations from policy buffer

at intervals of 10. Hence, we will analyze the average reward per episode of 2002 different policies and comment on their generalization capacities for torso masses $[6-8]$. For instance, in Figure 5.4 the value of the last point on the ACC-RARL line represents the average reward gained in the target hopper environment with torso mass 1 following the last snapshot of the policy trained with ACC-RARL algorithm in the source environment where torso mass is 3.53429174 as given in Figure 5.2.

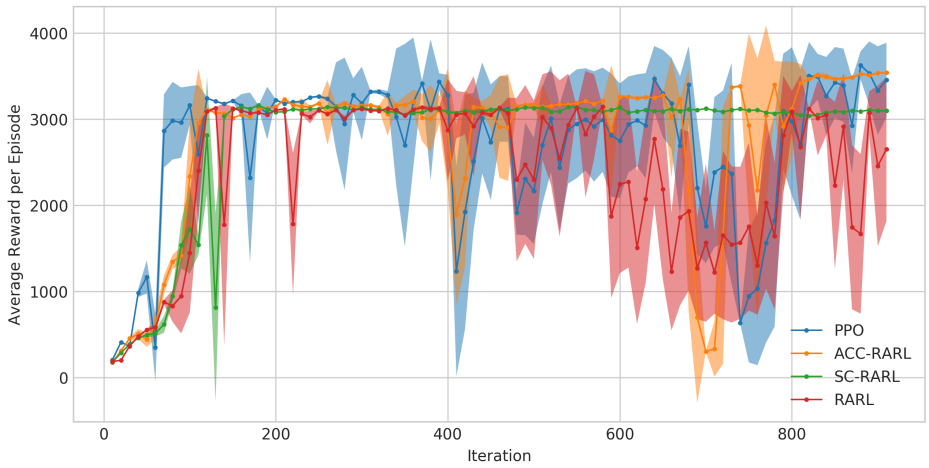

Figure 5.5. Average reward per episode at target environment with torso mass 2 of every 10 iterations from policy buffer

In Figure 5.5, it is seen that different iterations of each algorithm reach maximum performance in the target environment.

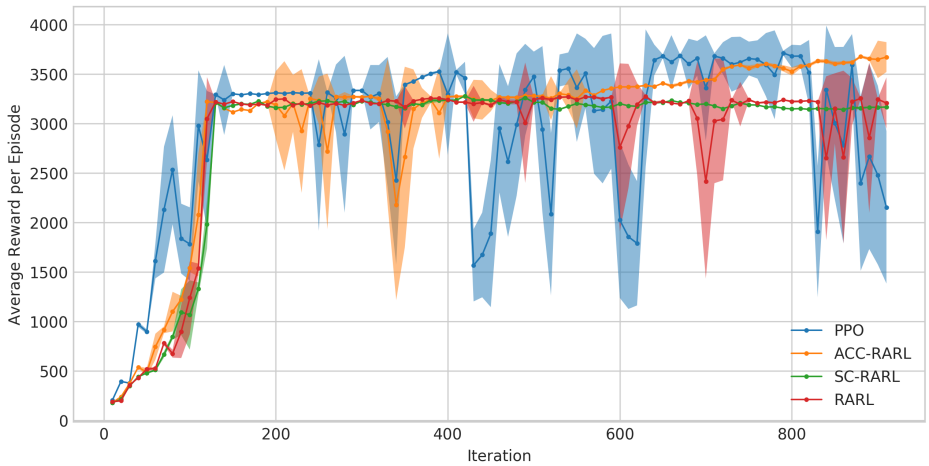

Figure 5.6. Average reward per episode at target environment with torso mass 3 of every 10 iterations from policy buffer

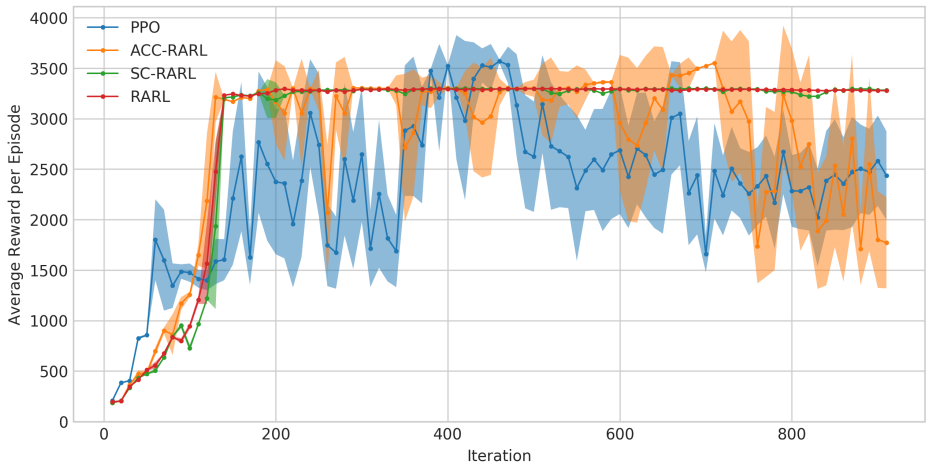

Figure 5.7. Average reward per episode at target environment with torso mass 4 of every 10 iterations from policy buffer

Figures 5.6 and 5.7 show that RARL and SC-RARL perform uniformly satisfactory for the corresponding benchmarks close to the source environment. Analyzing only these target environments might lead to failing to recognize policy iteration as a hyperparameter because it doesn't have much effect in these particular cases. Although both algorithms have different

critic initializations each critic is updated for the same number of iterations and using same structured loss functions thus it is understandable that In Figure 5.7 , the SC-RARL, and RARL perform similarly in the target environment with torso unit mass 4.

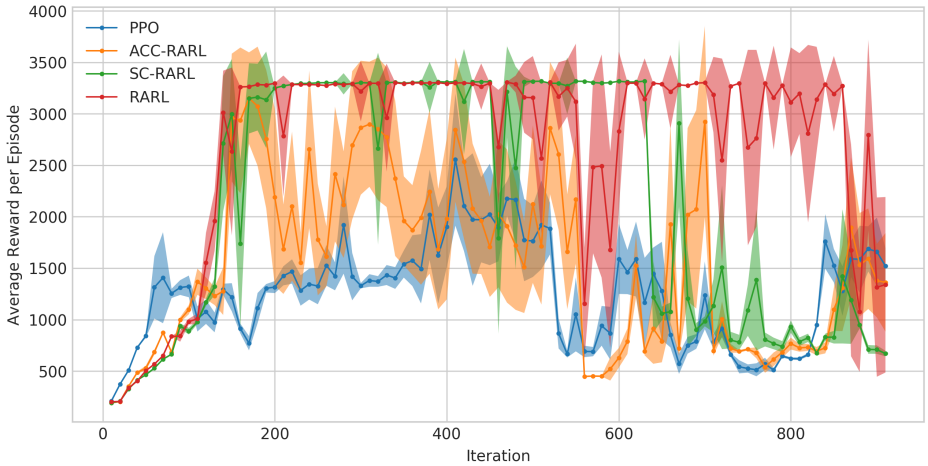

Figure 5.8. Average reward per episode at target environment with torso mass 5 of every 10 iterations from policy buffer

There is a considerable difference between the target environment performances of SC-RARL and RARL, especially in 800^{th} iteration, in Figure 5.8 which implies that training with the rewards of different trajectories sampled using different protagonist adversary pairs do indeed have a drastic effect on the type of control behavior learned.

The performance of the last iteration of RARL starts to decay in Figures 5.4 and 5.8. As a consequence, the agent should first resort to earlier snapshots of the policy intended for transfer to succeed in the harder target environments. Let us assume that the agent only has several snapshots of the policy in its policy buffer trained in the source environment with standard torso mass. Then the agent is put in a target environment with torso mass 6 which is analogous to an agent expected to carry weight while performing a control task. We propose that in cases like these, instead of training from the very beginning because the last iteration of each policy known by the agent performs below a certain threshold as in Figure 5.8, the agent should primarily resort to earlier policies at intervals suited for the context because the policies performing above 3000 are readily available in the agent's memory if the policy iterations are saved during training.

The policy buffer allows us to analyze all the different patterns of hopping learned during training. For instance, the type of hopping learned by the ACC-RARL between the iterations 550 and 800 is successful in the source task environment and environment where torso mass is 3 provided in Figure 5.6 but it is clearly unsuccessful in environments where torso mass is 1 and 2 as illustrated in Figures 5.4, 5.5. If we had not recognized the policy iteration as a hyperparameter then comparing the algorithms at arbitrarily selected iterations would not constitute a fair comparison. More importantly, the PPO that is generally used as a benchmark algorithm performs poorly if the last snapshot of it is used for comparison in a target task in Figure 5.4. However, for target environments with torso masses $[1-5]$, the right snapshots of PPO are capable of obtaining above 2500 average reward per episode from the environment as seen in Figures 5.4, 5.5, 5.6, 5.7, 5.8. Thus, transfer learning problem reduces to finding the most suitable snapshot of the policies from the policy buffer and constructing a meaningful policy buffer.

We should be aware that these target environment performance plots are unknown to us before sampling in the target environment. Consequently, we suggest that an alternative proper comparison of algorithms can be made by computing the area above a predetermined threshold for each line because that would suggest that the likelihood of choosing a robust policy from the snapshots of policies trained with the corresponding algorithm is higher if the area is bigger. Another evaluation method that we propose is to group the target environments and compare the algorithms based on their average performance of all tasks from the same group. However, in this thesis, we will plot the consecutive policy iterations of the algorithm that generated best performing policy in the target environment to prove that an expert level policy for a variety of environments has already been saved during training.

If target task with torso mass 3 is regarded as the validation task and a limited number of experiences were gathered to approximate generalization capacity of each policy, we might have had some idea of what the jumpstart performance of each policy might be but we cannot have anticipated that the policy iterations of ACC-RARL and RARL saved between 600 and 800 will perform worse for target task masses 1 and 2 as seen in Figures 5.4, 5.5. Similarly, this method does not work for the iterations of PPO saved near 800. The best performing policy iterations of PPO at validation environment yield fewer rewards than the

earlier iterations as given in Figure 5.4.

Assuming that the environments are parametrized and the distance between the source and target environments can be computed as a nonlinear function, it might be possible to predict performance of the policies residing in the policy buffer. Coinciding with the performances seen in Figures 5.4,5.9,5.11,5.13, it is anticipated that the earlier policy iterations trained with less samples perform better as the distance between target environment and the source environment increases in the parameter space. As seen in Figures 5.5, 5.6 and 5.7, the last iteration of the policy trained with RARL with PPO perform better or same compared to the original RARL experimentsn carried out with TRPO in [15]. For masses below hopper's torso mass the last iterations of ACC-RARL shown in Figures 5.4, 5.5, 5.6 performs superior to the last iteration of SC-RARL and RARL.

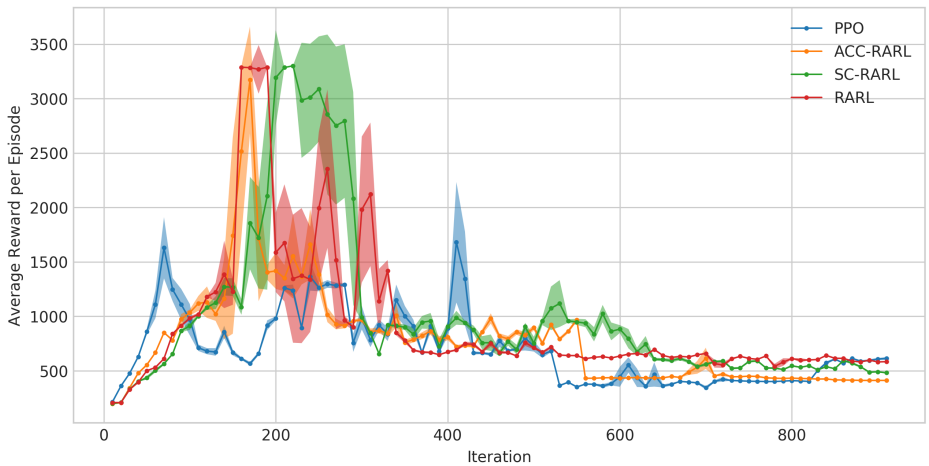

Figure 5.9. Average reward per episode at target environment with torso mass 6 of every 10 iterations from policy buffer

In Figure 5.9, a significant target environment performance drop occurs after approximately 300^{th} until the last policy iteration where all algorithms are affected. This implies that the type of hopping behavior learned after a certain point of training can't generalize to hopper environments with higher torso masses and all policy iterations trained via PPO algorithm with the given hyperparameters are inadequate. It should be pointed out that the best performing policy iterations of all algorithms start to get concentrated to the range $[150-300]$ thus a mapping between the target environment parameters and the policy iterations is highly

probable.

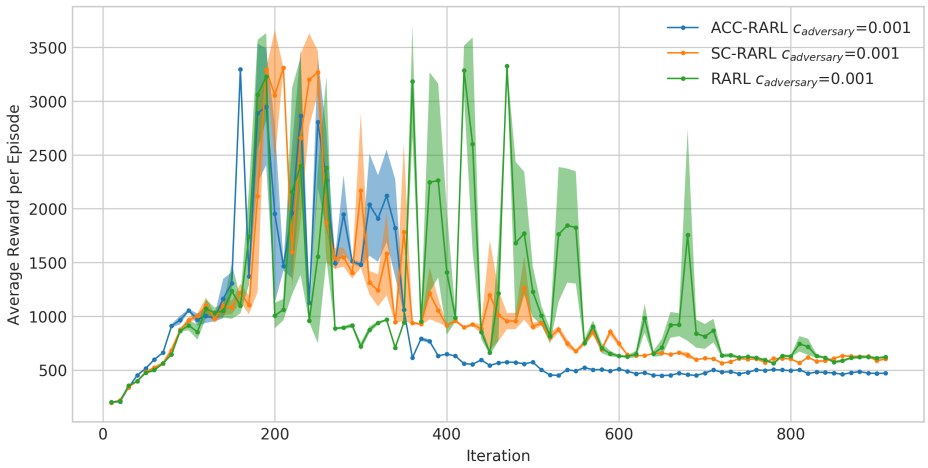

Figure 5.10. Average reward per episode at target environment with torso mass 6 of every 10 iterations from policy buffer

Addition of the entropy bonus increased the fluctuation of the average rewards for all algorithms as seen in Figure 5.10 but for the better. Because the standard deviations of adversary's output action probability distributions are increased during optimization, the adversary takes more randomized actions which often changes the protagonist's way of hopping by destabilizing the equilibrium.

In harder environments with torso mass 7 and 8, the earlier iterations of the policy trained with critic architecture ACC-RARL that we propose performs the best. Figure 5.11 shows that the range of the best performing policy iterations are contracted more and the performance similarities of RARL and SC-RARL are more apparent as illustrated in Figure 5.7. Moreover, it is observed in Figure 5.12 that the benefit of adding entropy to the adversary loss function in ACC-RARL and SC-RARL continues in the target environment with torso unit mass 7 whereas standard RARL is seen to perform better in this case. Due to this observation, it is deduced that the addition of entropy bonus is not guaranteed to increase the maximum average performance in the target environment.

The best average episode rewards for SC-RARL is trained with adversary loss function including the entropy bonus with entropy coefficient $c_{adversary} = 0.001$ as shown in Figures

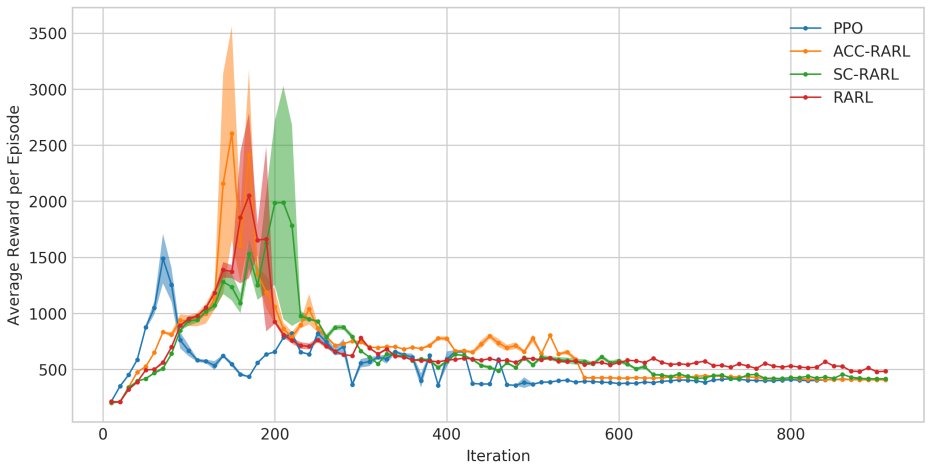

Figure 5.11. Average reward per episode at target environment with torso mass 7 of every 10 iterations from policy buffer

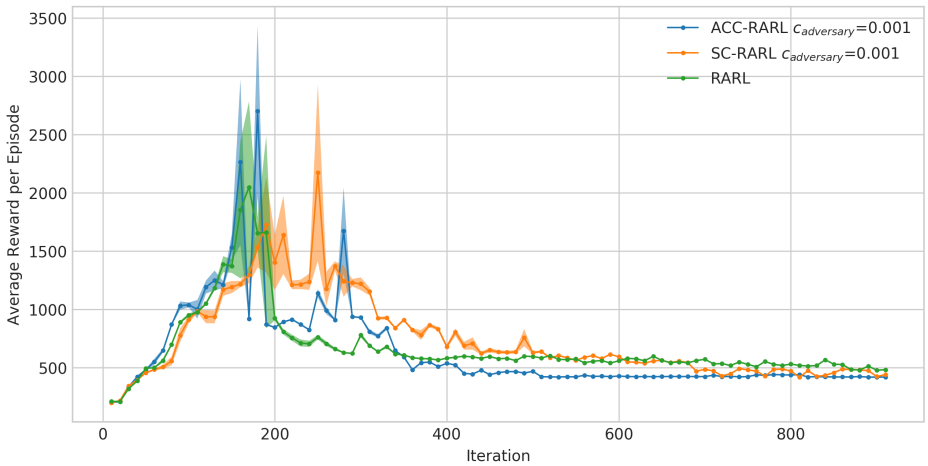

Figure 5.12. Average reward per episode at target environment with torso mass 7 of every 10 iterations from policy buffer

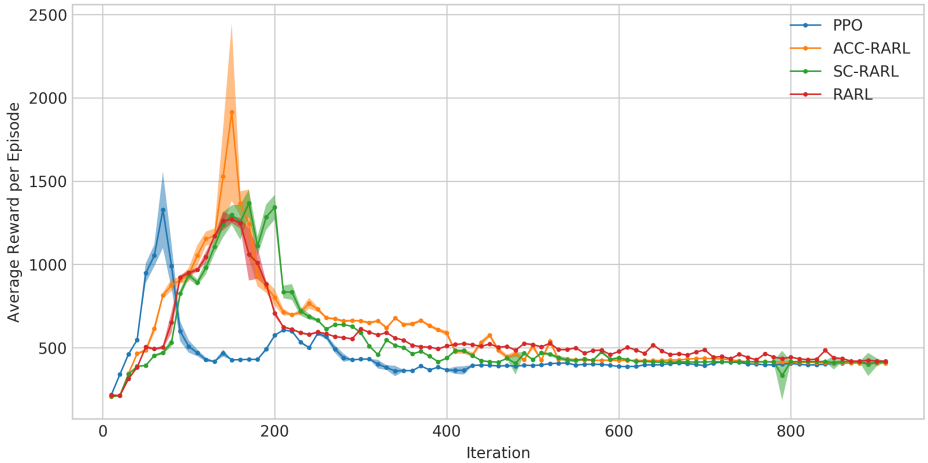

Figure 5.13. Average reward per episode at target environment with torso mass 8 of every 10 iterations from policy buffer

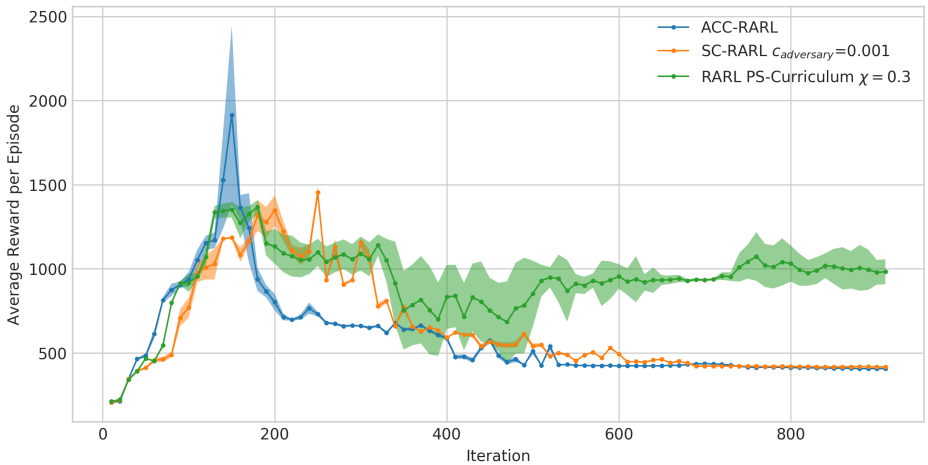

Figure 5.14. Average reward per episode at target environment with torso mass 8 of every 10 iterations from policy buffer

5.10, 5.12, 5.14. Training with adversary entropy shifts the best-performing policy iterations slightly to the right due to the increased domain randomization through the encouragement of adversary policy exploration via entropy bonus.

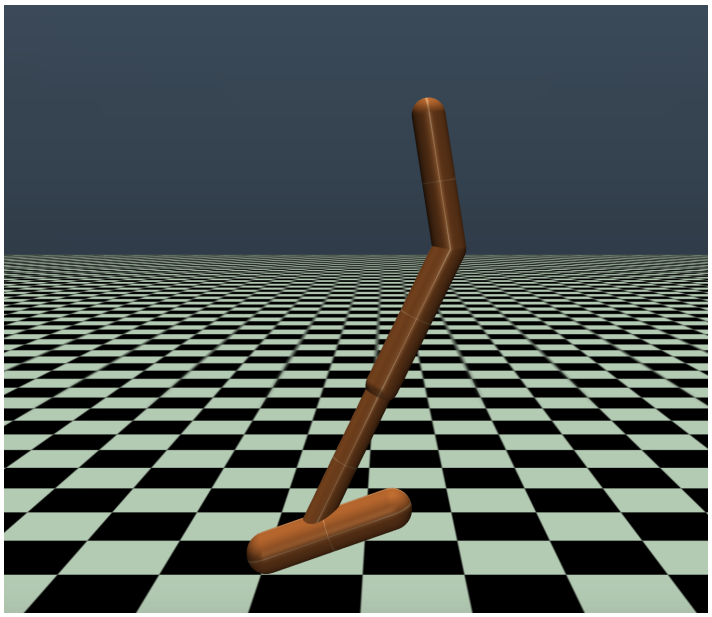

Figure 5.15. Hopper robot hopping with torso mass 8 in target environment using the 150th iteration of policy trained with ACC-RARL

When the mass of the torso is increased to 8, 150^{th} iteration of policy trained with ACC-RARL gains an average episode reward of 1914 ± 531.8 as plotted in Figure 5.13. The hopper hopping in this environment is shown in Figure 5.15. It is observed that the aforementioned policy is able to hop carefully until it loses balance and falls.

Although RARL trained with curriculum (RARL PS-Curriculum $\chi = 0.3$) still performed worse than ACC-RARL, it is observed that there is less variation among policy snapshots recorded after 100 iterations as shown in Figure 5.14. The increase in the performance indicates that training with the hardest adversary policies might not be beneficial. The robustness of a randomly chosen snapshot from the policies trained with RARL PS-Curriculum $\chi = 0.3$ has increased. The curriculum learning variants don't guarantee a better performance since the only case one of them performed better than the original algorithms

for this benchmark is in the target environment with torso unit mass 8 and the maximum performance increase is negligible.

Table 5.3. Average Reward per Episode of 2 policy iterations trained with ACC-RARL

Unit Mass	**Iteration**	**Average Reward per Episode**
1	170	2903 ± 11.74
2	170	3028 ± 9.902
3	170	3132 ± 3.32
4	170	3199 ± 5.411
5	170	3152 ± 444.4
6	170	3172 ± 489.8
7	150	2606 ± 952.3
8	150	1914 ± 531.8

If the target environments are grouped as all the target environments lower than the source environment's torso mass and as all the target environments higher than the source environment's torso mass we find that each target group requires a different policy if the highest possible target performance is intended. We show the performance of two policy iterations with high generalization capacity trained with the ACC-RARL algorithm in Table 5.3 to demonstrate that only two closely saved policy iterations are capable of performing forward locomotion when torso mass is in the range $[1-8]$. As the target environment gets harder it is seen that the earlier iterations perform better due to the regularization effect of early stopping.

5.3.2. Humanoid

Our aim in this section is to demonstrate the applicability of the method we proposed in Section 3.1.2 in real life scenarios by transferring learning among different robots and to accomplish a delivery task. We extend the morphological modification experiments to cases where the loss function differs by introducing transferring learning among robots with different morphologies. Since the termination criterion depends on the location of the center

of the torso, the loss functions of both tall and short humanoid environment are updated. For the tall humanoid, the range of the constraint shifts higher and for the short humanoid, it shifts lower by the height of the waist component variant. In addition to that, the total body weights of short and tall humanoids differ from the standard humanoid by the exclusion and inclusion of the upper waist respectively as seen in Figure 5.16.

One of the expected critical missions of a service robot is to carry out household chores. A large portion of these requirements involves tidying up the house and fetching the items required by the humans. Relocating non-homogenous objects is a part of the problems that occur in bipedal locomotion. In contrast to the tall and short humanoid environments, the loss function does not differ from the standard humanoid source environment for the delivery robot but the total body mass increases as much as the mass of the delivery box similar to the tall humanoid benchmark.

The method of clipping is primarily used to discourage catastrophic displacement in the parameter space. We observed that a higher clipping parameter causes a sudden drop in the learning curve and puts the policy in an unrecoverable location at the policy parameter space. We have found that when strict clipping with unconventional values like $\varepsilon = 0.01$ is used in a transfer learning setting, the MDP samples that lead to overfitting to the source task are discarded. Using strict clipping the trajectory that is used in the optimization process will be free of the variance introduced by the source task-specific samples.

Table 5.4. Delivery Environment

Body	**Unit Mass**
Delivery Box	5
Right Hand	1.19834313
Torso	8.32207894
Total Body without Delivery Box	39.64581713

Figures 5.16, 5.19 and 5.21 show the performance the of policies trained with clipping parameter of $\varepsilon = 0.1$ and $\varepsilon = 0.01$. The lines of the strictly clipped policy iterations tend

to be smooth thus saving the policy parameters every 50 iterations for all environments is sufficient for this experiment. Additionally, we've also tried RARL algorithm for these tasks but strict clipping performed superior with the hyperparameters we've used. Besides, strict clipping achieved remarkably well results in our benchmarks so the domain randomization enacted by the adversary is not needed. Strict clipping allows the humanoid to learn general characteristics of forward locomotion that can be transferred to various different environments by discarding samples that cause overfitting.

Figure 5.16 shows that all the policies trained in the standard humanoid environment with strict clipping $\varepsilon = 0.01$, saved after 950^{th} iteration perform exceptionally well in the target environment. 1100^{th} and 1250^{th} policy iterations not only gained a high average reward per episode but also performed consistently well with low standard deviation over all the trajectories sampled from 32 environments with different seeds. The policy with the highest average reward per episode 1250^{th} is used for the short humanoid environment simulation. The policy is directly transferred to the shorter humanoid and the short humanoid is able to run without the need for adaptation. In contrast, the policy with clipping parameter $\varepsilon = 0.1$ can't transfer the learning it attained in the source environment because the additional samples used during optimization caused overfitting to the source environment. It is seen in Figure 5.16, that even the earlier iterations of the policy trained with clipping parameter $\varepsilon = 0.1$ can't be transferred to a shorter robot.

Figure 5.18 is a snapshot from the tall humanoid environment simulation when the 1450^{th} policy iteration is used. We observed that the humanoid takes smalles steps to stay in balance with a larger upperbody and a higher $+z$ constraint. Although presumably taller humanoid would be situated at an approximately opposite location in a hypothetical environment parameter space, the Figure 5.19 proves that same policy iterations trained with strict clipping performs well in the tall humanoid environment. This suggests that the form of moving forward is applicable in both of these environments as well as the source environment. As a result, a tradeoff between generalization capacity and the performance arises as we prove that overfitting to the source task samples is a crucial issue. The tall humanoid environment is a harder environment than the short humanoid in our results as expected because it is harder to keep balance with heavier upper body mass. Similarly, we also observe

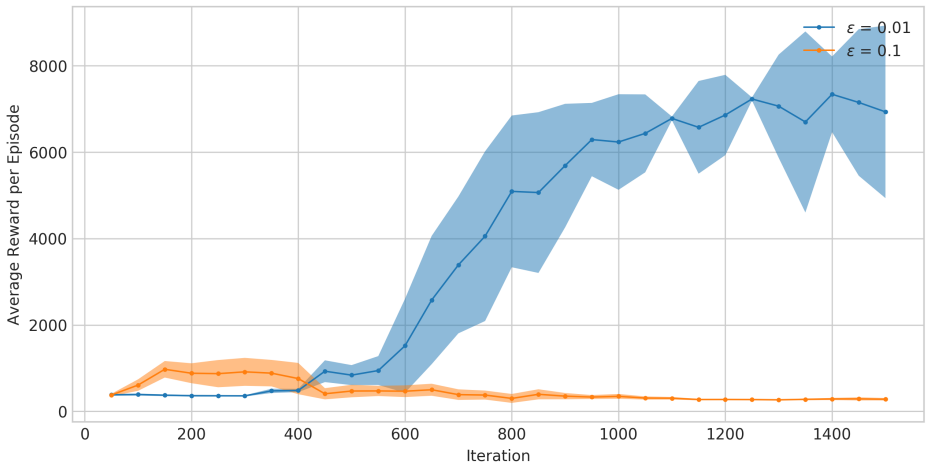

Figure 5.16. Average reward per episode of every 50 iterations from policy buffer for a shorter humanoid

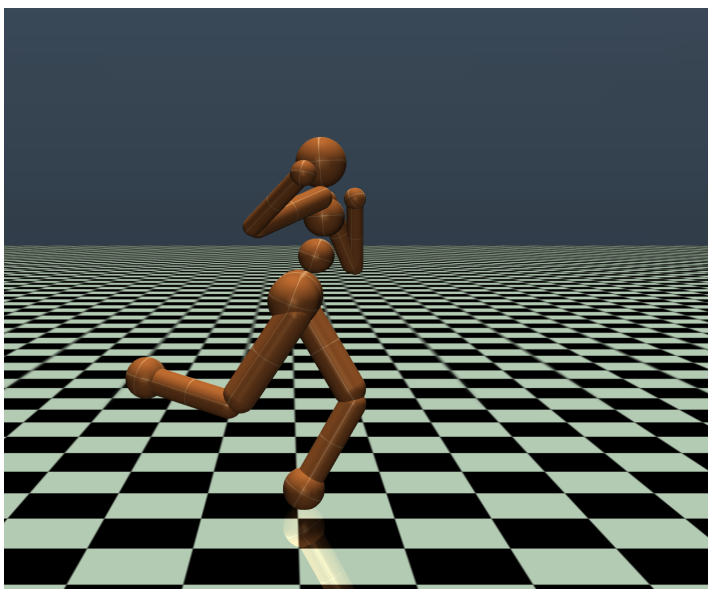

Figure 5.17. Short humanoid

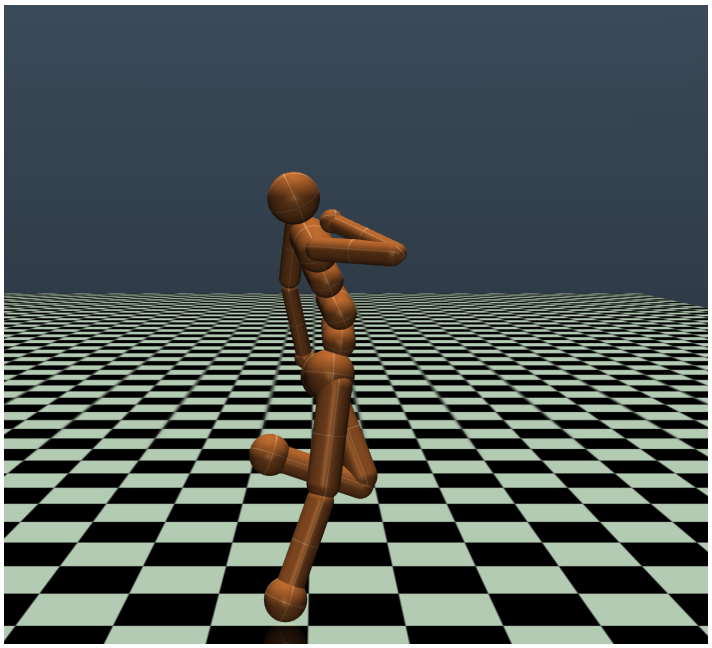

Figure 5.18. Tall humanoid

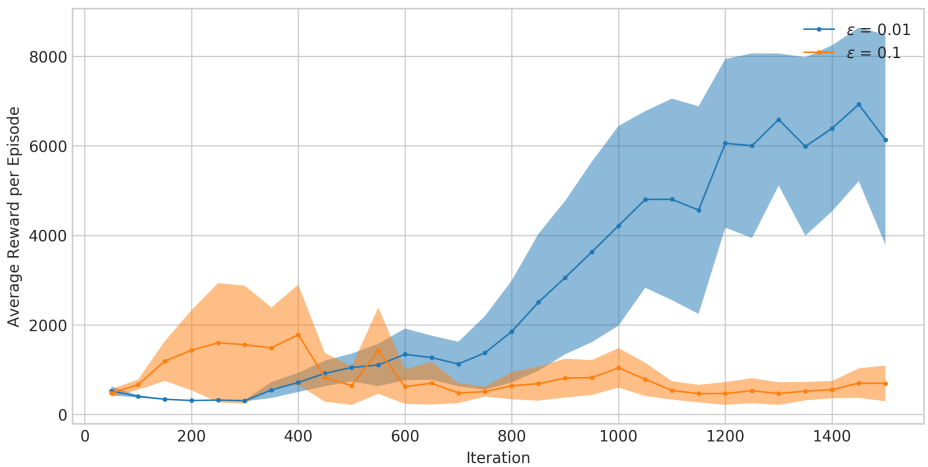

Figure 5.19. Average reward per episode of every 50 iterations from policy buffer for a taller humanoid

higher variance in the average rewards collected from the hopper environments with heavier unit torso mass discussed in Section 5.3.1.

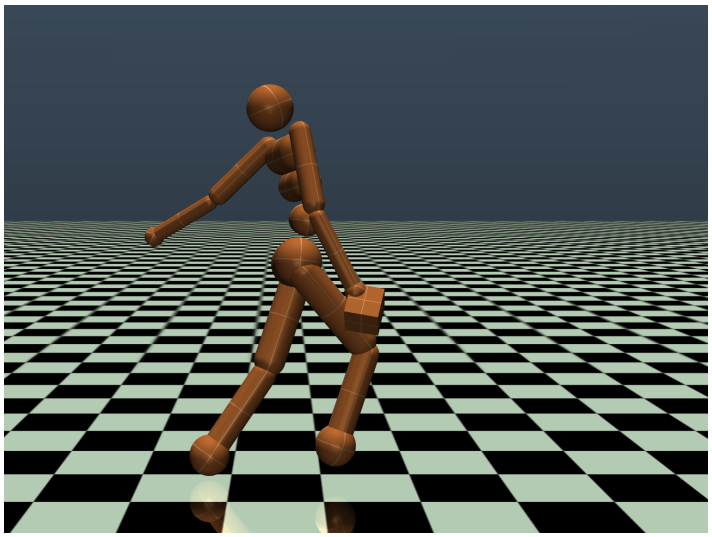

Figure 5.20. Standard delivery humanoid

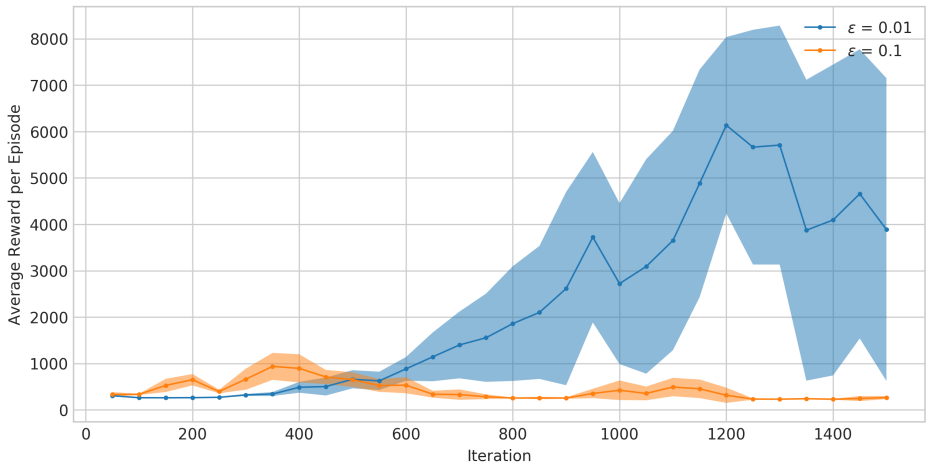

Figure 5.21. Average reward per episode of every 50 iterations from policy buffer at target environment where a delivery box of mass 5 unit is carried using right hand

Masses of relevant body parts for the delivery robot are given in Table 5.4. Taking into account the total body mass, a delivery box with a unit mass of 5 constitutes a challenging benchmark. The design decision was made to create imbalance by enforcing the humanoid to carry the box only by the right hand. Humanoid is able to carry the heavy box just like a

human using the 1200^{th} policy. 1200^{th} policy iteration that has the best target environment jumpstart performance is shown in the simulation snapshot in Figure 5.20. The simulation performance shows that the humanoid can generalize to this delivery task just by utilizing the learning attained from standard humanoid environment. The reduction in the performance after the 1200^{th} iteration in Figure 5.21, supports our method of resorting to the earlier policy iterations before discarding all the snapshots of the corresponding algorithm. This concavity is also observed in the heavier hopper environments in Section 5.3.1 and suggests that the experience and learning gained from the target environment is detrimental after a point based on the target environment.

The humanoids fall immediately in the target environments when the best-performing policy in the source environment is used thus the orange curves in Figures 5.16, 5.19 and 5.21 assess our claim that source environment performance isn't indicative of the generalization capacity. In contrast, when strict clipping is used as a regularization technique for PPO, the humanoid is able to run in the proposed target environments. In these sets of experiments, our policy buffer consisted of snapshots of policies trained with only $\varepsilon = 0.1$ and $\varepsilon = 0.01$. An alternative policy buffer might consist of snapshots of policies trained with different hyperparameters or training methods and a better performing snapshot trained solely in the source environment might be found for each benchmark. In order to find the best performing policy from the buffer when there is no surrogate validation environment, more sampling should be done in the target environment. Thus, the tradeoff between the number of trajectories rolled out and the performance in the target environment emerges. If this problem is not acknowledged the number of experiences gathered from the environment might even exceed the number of samples used for random initialization from the beginning. In consequence, the policy buffer should consist of the least amount of snapshots possible. In this thesis, we show how different policies perform in different target environments to provide insight on logical ways of constructing a policy buffer.

5.4. The Friction Environment

Frictional variation is one of the most common scenarios encountered in real life for bipedal locomotion of a humanoid. The environment we designed to benchmark this scenario

is shown in Figure 5.22. The humanoid is seen sinking to the ground due to high tangential friction but still is able to run using the policy trained with strict clipping unaware of the environment friction.

When transferring the last iteration of a policy trained in an environment with tangential friction of 1 to an environment with tangential friction 4, we found that transferring the learning from an easier task results in a higher asymptote and slope compared to jumpstarting the harder task from ground zero. The striking difference of asymptotic performance inspired us to look for better ways of designing the training environment.

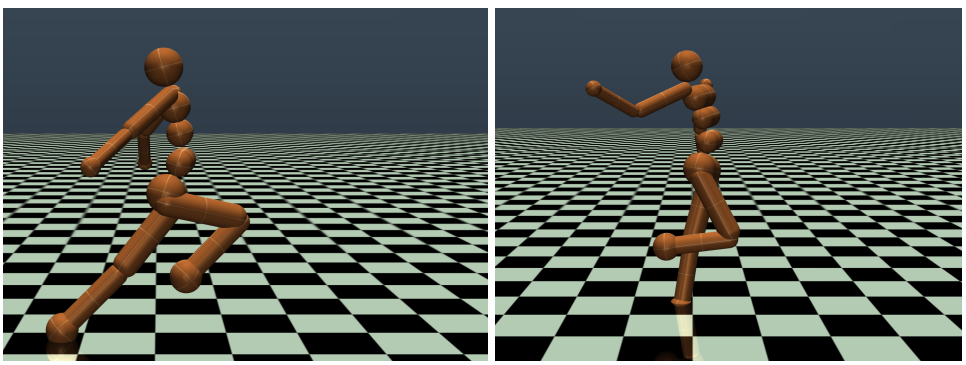

Figure 5.22. Humanoid runing in target environment with tangential friction 3.5 times the source environment

Instead of using multiple different policies for environments with different friction coefficients, choosing a policy with a higher generalization capacity is sufficient even for a target environment with 3.5 times the tangential friction of the source environment.

Table 5.5. Best Performing Iterations of Policies in Target Friction Environment

Clip	**Iteration**	**Average Reward per Episode**
0.01	1500	8283 ± 24.26
0.1	300	1078 ± 336.4

In Figure 5.23, the best jumpstart performances for each clipping parameter are given. Each policy is tested on 32 different target environments initialized with 32 different seeds

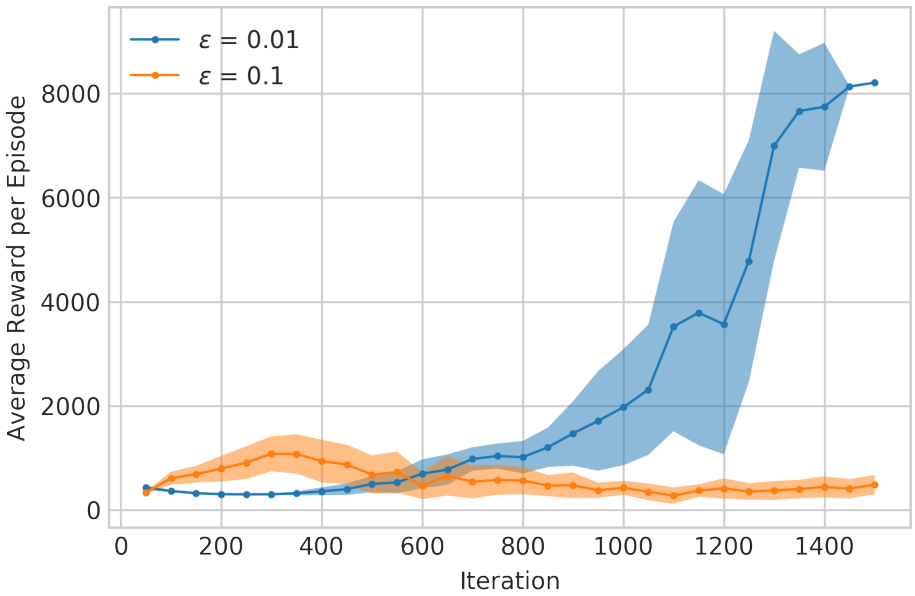

Figure 5.23. Comparison of average reward per episode of policies at target environment with tangential friction 3.5 times the source environment

but has the same friction coefficient of 3.5. For instance, as seen in Figure 5.5, the last iteration of the policy with a strict clipping $\varepsilon = 0.01$ trained in the source environment has an average reward of 8283 and a standard deviation of 24.26 across all target environments. In contrast, the best performing policy in the source environment has a low generalization capacity due to the fact that it overfits the source task environment. More samples are discarded and the movement in parameter space is restricted using strict clipping thus the agent learns more generalizable patterns of bipedal locomotion.

5.5. The Gravity Environment

During training in the source environment with earth's gravity, hopper and humanoid learn a variety of different techniques at different stages of training to perform target tasks. When put in a challenging environment, the robot should resort to the snapshots of memories engraved as the recorded saved policy parameters.

5.5.1. Hopper

In Learning Joint Reward Policy Options using Generative Adversarial Inverse Reinforcement Learning (OptionGAN) [64], the parameter space of gravity environment is between $0.5G_{earth}$ and $1.5G_{earth}$ for both humanoid and hopper. The policy over options converges to 2 different policies for Hopper tasks: one for lower and one for higher than the earth's gravity indicating that the tasks are complex enough to be solved with different policies. In these sets of experiments conducted with a larger range of gravity environments specifically $0.5G_{earth}$ and $1.75G_{earth}$, we will prove that by only choosing the right iteration of policies, gaining average episode reward greater than 3000 in the target environment is possible without expert demonstrations.

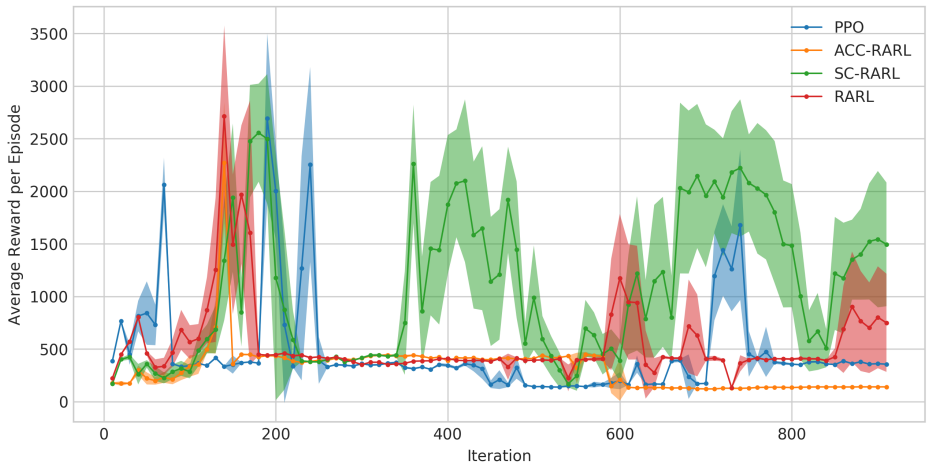

Figure 5.24. Average reward per episode at target environment with Gravity= -4.905 ($0.5G_{earth}$)

For these sets of tasks, we will use the same policy buffer we've created using PPO and different variations of RARL for the experiments in Section 5.3.1. The plots in Figures 5.24 and 5.25 prove that the probability of picking the right iteration of policies trained using curriculum from the policy buffer that conforms to expectations is higher.

In Figure 5.26 the baseline PPO's 420^{th} policy is shown where the average return is 3493 ± 294.2. Although it is the best performing policy among the policy iterations recorded at intervals of 10 it is harder to find this iteration than the policies trained with RARL. The

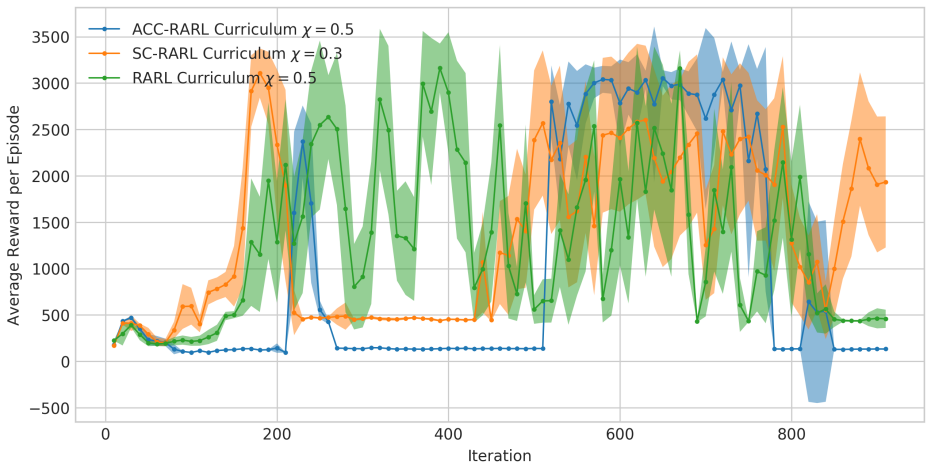

Figure 5.25. Average reward per episode at target environment with Gravity= - 4.905 ($0.5G_{earth}$)

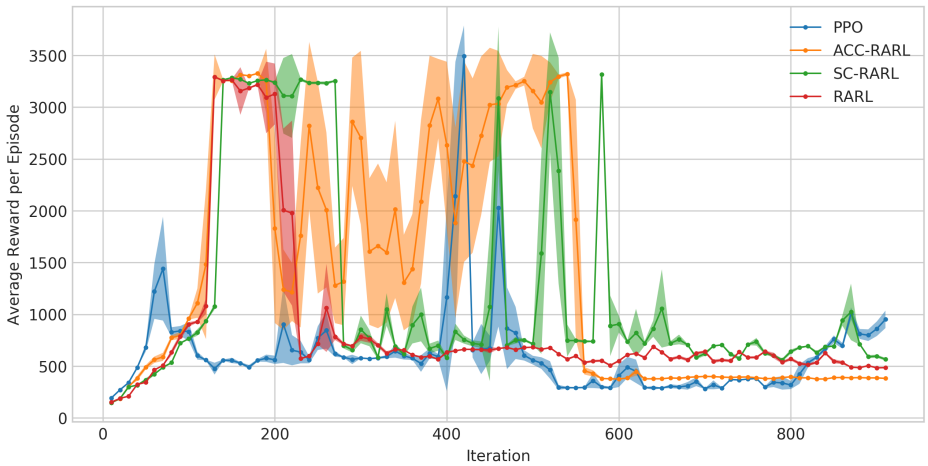

Figure 5.26. Average reward per episode at target environment with Gravity= - 14.715 ($1.5G_{earth}$)

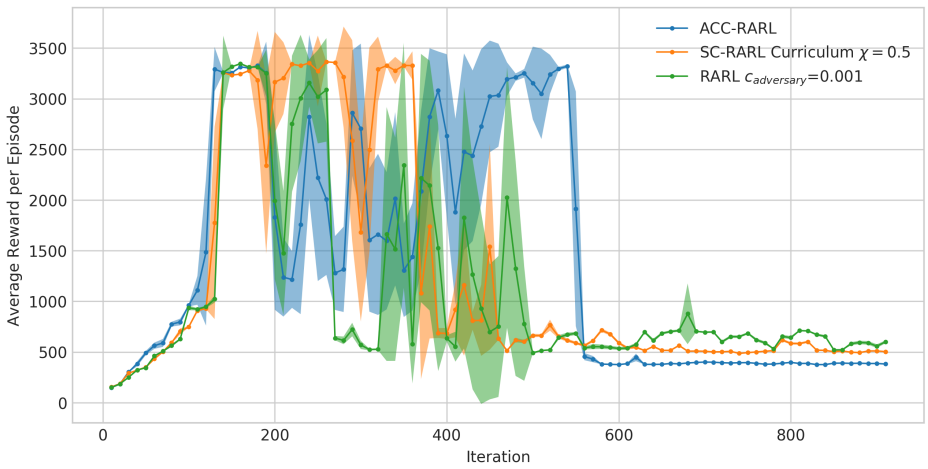

Figure 5.27. Average reward per episode at target environment with Gravity= - 14.715 ($1.5G_{earth}$)

Figure 5.27 shows that training SC-RARL with curriculum and inclusion of entropy bonus in adversary loss function not only increased the average reward per episode for the best performing policy iterations but also increased the number of policy snapshots that achieved an average episode reward above 2000.

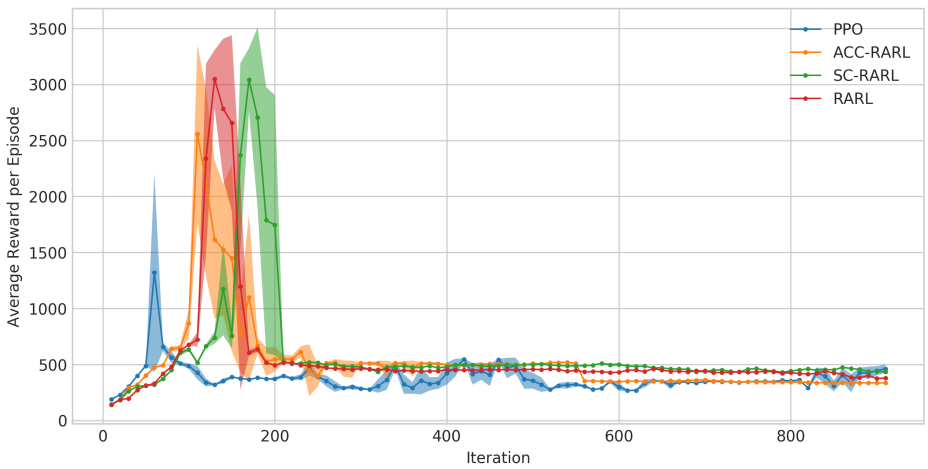

Figure 5.28. Average reward per episode at target environment with Gravity= - 17.1675 ($1.75G_{earth}$)

As the target environment gets harder thus further away from the source environment, the best performing iterations of policies are aggregated around earlier iterations. As seen

in Figure 5.28 the domain randomization in naive PPO doesn't suffice for generalization in harder target environments.

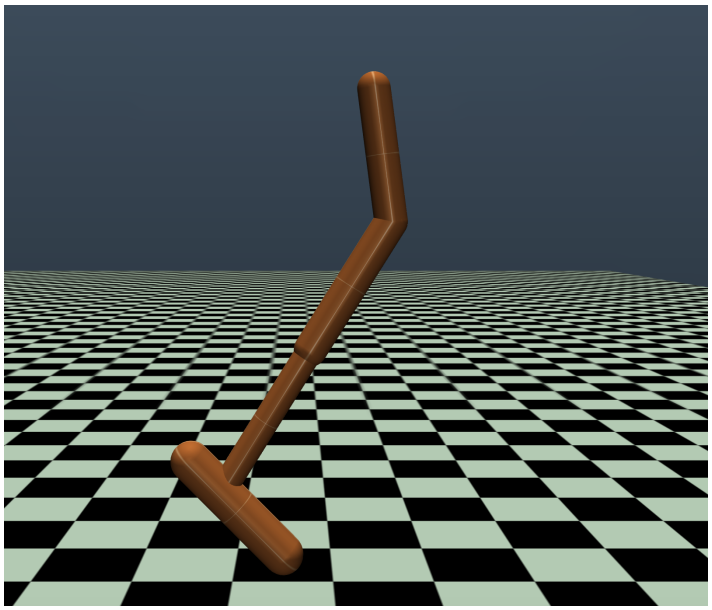

Figure 5.29. Hopper robot hopping at target environment with Gravity= - 17.1675 ($1.75G_{earth}$)

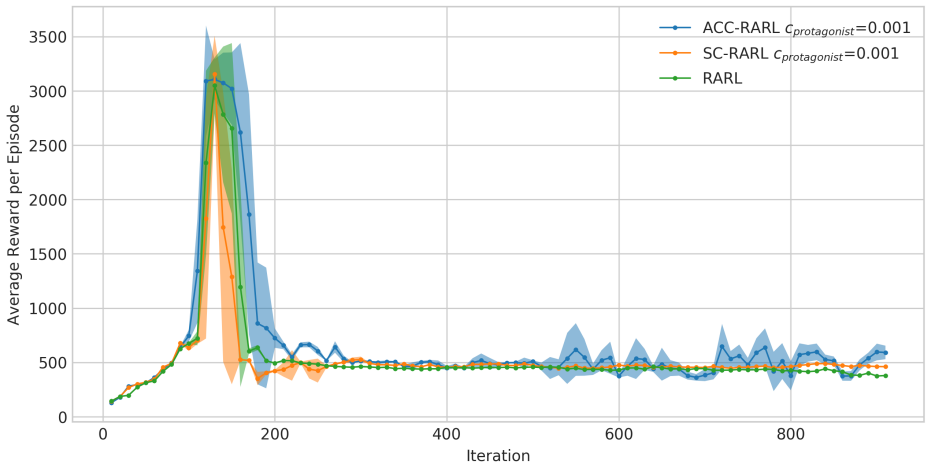

Figure 5.30. Average reward per episode at target environment with Gravity= - 17.1675 ($1.75G_{earth}$)

In Figure 5.30 we see that encouraging the exploration of the protagonist policy through the inclusion of entropy bonus increases performance for policies trained with ACC-RARL

and SC-RARL. Although entropy doesn't guarantee an increase in all harder target environments, in Section 5.3.1 we've also demonstrated that entropy increases the performance for ACC-RARL and SC-RARL if included in the loss function of the adversary.

Figure 5.29 shows the hopper hopping in target environment with Gravity= - 17.1675 ($1.75G_{earth}$) following the policy trained via SC-RARL with protagonist entropy bonus $c_{protagonist} = 0.001$. In Figures 5.28 and 5.30 we see the same concavity encountered in the performance plots of heavier torso mass target environments in Section 5.3.1 and delivery environment in Section 5.3.2. As the difference between the target environment and the source environment increases, the range of the better performing policy iterations reduces similar to the observations for the heavier torso mass environments discussed in Section 5.3.1. This curve is analogous to the convexity of the test error curve in supervised learning problems where movement towards the earlier training cycles along the training error curve leads to underfitting and the later training cycles results in overfitting. The optimum point on the curve has high generalization capacity and performance in the target environment. Additionally, just as the training error curve stays the same the source task performance is the invariant whereas the target environment structure is the variable affecting the learning curve of the test error.

The higher gravitational force also acts similar to heavier torso mass so there might be some correlations between the proposed environments but the regularization effect of early stopping does contribute to the increased generalization capacity. Above all, the gravity environments performed in line with morphological modifications and resorting to earlier policy parameters is shown to be a befitting method to succeed in harder target environments.

5.5.2. Humanoid

In order to stay in balance under harsh circumstances, the policy that is being transferred should be robust to unknown environmental dynamics. Walking uninterruptedly in gravities lower and higher than the earth's environment requires different patterns of forward locomotion unlike the morphological humanoid benchmarks in 5.3.2 where the 150^th iteration of each policy trained with the same clipping parameter gained above 4000 average

rewards per episode for all environments.

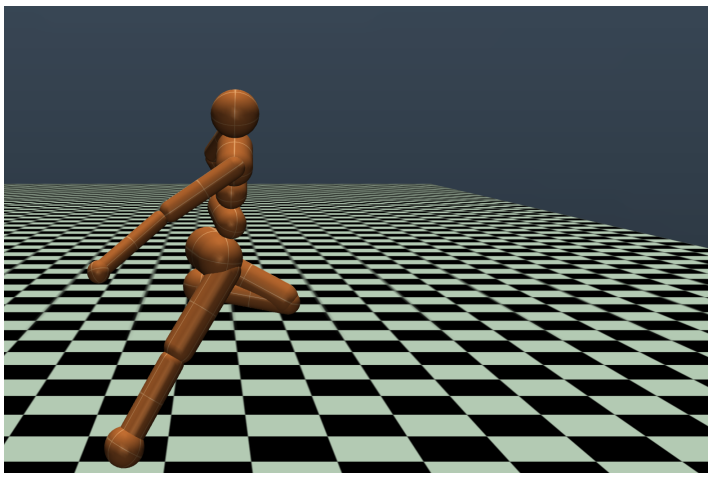

Figure 5.31. Humanoid in target environment with gravity= -4.905 ($0.5G_{earth}$)

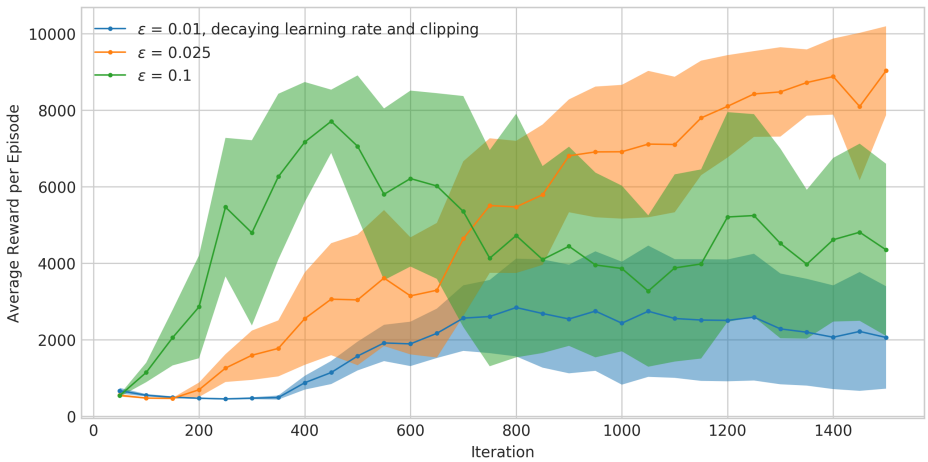

Figure 5.32. Average reward per episode at target environment with gravity= -4.905 ($0.5G_{earth}$)

Figure 5.31 shows that when the last iteration of the policy trained with strict clipping $\varepsilon = 0.025$ is used in the target environment with $gravity = -4.905$ ($0.5G_{earth}$) the humanoid is able to run. Although earlier snapshots of the policy that shows the best training performance yields less average rewards than the last policy iteration of PPO trained with $\varepsilon = 0.025$ in the target environment, the performance is still remarkably well. Both regularization techniques namely early stopping and strict clipping show the same performance and generalization to this target environment.

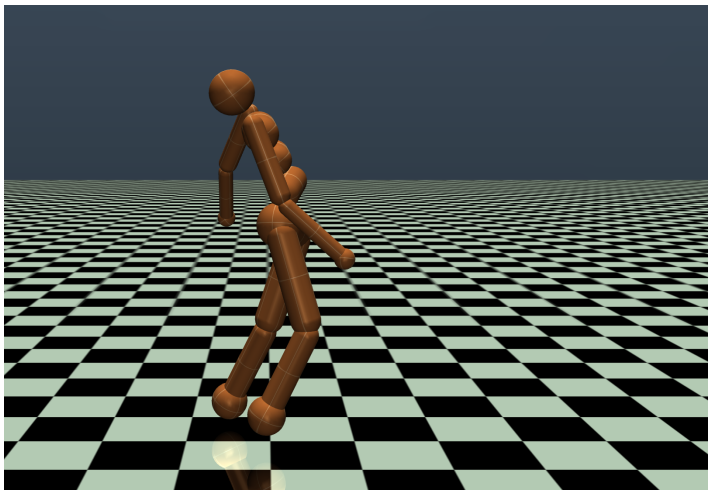

Figure 5.33. Humanoid in target environment with gravity= - 14.715 ($1.5G_{earth}$)

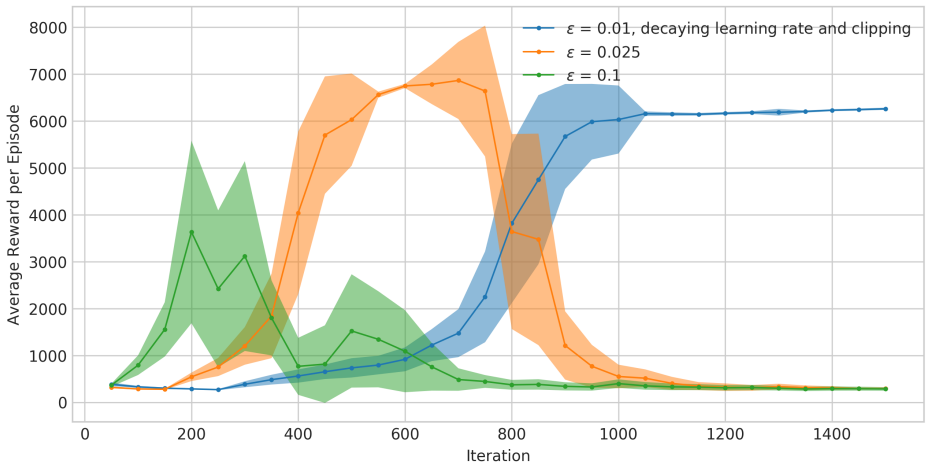

Figure 5.34. Average reward per episode at target environment with gravity= - 14.715 ($1.5G_{earth}$)

Utilizing both strict clipping and early stopping was used together in harder environments like the delivery environment in Section 5.3.2. Similarly, in the target environment with gravity= - 14.715 ($1.5G_{earth}$) the humanoid needs to resort to the previous snapshots of the policy trained with strict clipping $\varepsilon = 0.025$ as plotted in the Figure 5.34. The bipedal locomotion pattern in the simulated target environment with gravity= - 14.715 ($1.5G_{earth}$) when the humanoid jumpstarts with the 600^{th} policy trained with clipping parameter $\varepsilon = 0.025$ is shown in Figure 5.33.

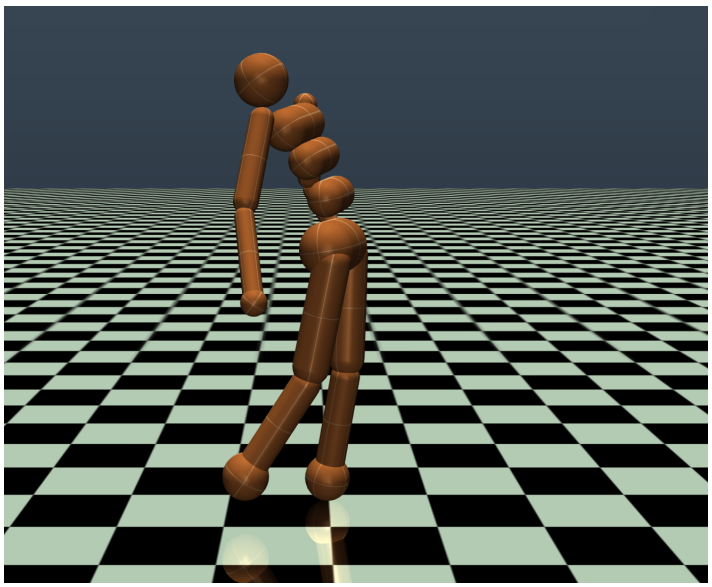

Figure 5.35. Humanoid in target environment with gravity= - 17.1675 ($1.75G_{earth}$)

Gravity benchmarks for the humanoid indicate that snapshots of different policies should be used for the target environment with gravity= - 17.1675 ($1.75G_{earth}$).The policy iterations trained with hyperparameters "$\varepsilon = 0.01$, and *decaying learning rate and clipping*" performed poorly in the source environment and target environment with lower gravities given in Figures 5.1 and 5.32 respectively. However, the last iterations of them perform consistently well in environments with higher gravities. Figures 5.34 and 5.36 reveal that decaying clipping during training might have hindered the exploration and restricted the humanoid to stick to a more careful way of stepping forward under the high gravitational force which pulls the humanoid to the ground as in Figure 5.35.

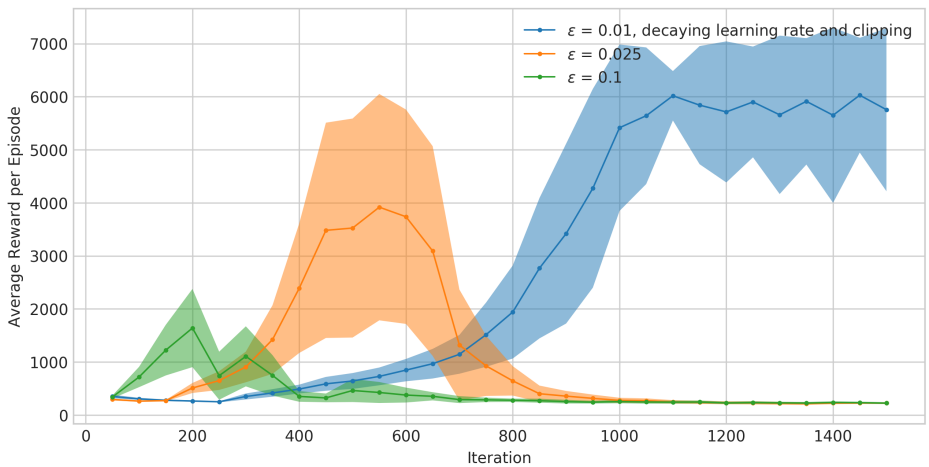

Figure 5.36. Average reward per episode at target environment with gravity= - 17.1675 ($1.75G_{earth}$)

6. CONCLUSION

Humans are capable of analyzing continuously arriving data from five senses and accumulate vast knowledge throughout their existence. In the quest of building machines with higher capabilities, transferring the accumulated knowledge among tasks and starting an unknown task with a strong prior understanding are essential. On account of this, we address the crucial problems in transfer learning that should not be neglected to develop meaningful novel algorithms and suggest new methods to increase the ability of generalization.

Transfer learning via deep reinforcement learning techniques is shown to be tricky because the generalization capacity is inherently dependent on the hyperparameters. The results we show in Chapter 5 are in line with our hypothesis that overtraining is, in fact, a major issue in transfer deep reinforcement learning.

In this thesis, we first proved that source task environment performance isn't indicative of generalization capacity and target task performance in deep transfer reinforcement learning. An agent tries a substantial amount of different action combinations depending on the hyperparameters in the training phase of deep reinforcement learning algorithms. With each environment interaction, the agent's strategy of solving that particular task is aimed to advance globally. At some point, the agent becomes an expert at performing that task but doesn't remember the strategies it has acquired during the earlier phases of learning. We proved that these overridden strategies or robust strategies with poorer training performance yield higher performance in our transfer learning benchmarks. Provided that forward locomotion is an integral problem in continuous control, we altered the gravity and the tangential friction of the environment and the morphology of the agent in our benchmarks.

In our work, we propose keeping a policy buffer analogous to human memory to capture different strategies because training performance doesn't determine test performance in supervised learning problems. Accordingly, transferring the best performing policy at source task environment to the target environment will prove to be an inadequate evaluation technique as the difference between the source and target environment increases. Consequently,

this methodology allowed us to compare the extended scope of each algorithm and transfer the learning attained in a source environment to harder target environments. Hence we reduced the problem to choosing the best performing policy for the target environment from the buffer. In addition to that, we suggest the use of surrogate validation environment if possible, to tune the hyperparameters via choosing the best fitting policy from the buffer.

In supervised learning for deep neural networks, early stopping [81] is used when the algorithm's generalization capacity starts to decrease. To the best of our knowledge, this thesis is the first study that extends the use of early stopping as a regularization technique to deep reinforcement learning. Knowing where to stop depends on the difference between target and the source environment thus since we're not given the context of the target environment during training in the source environment we keep snapshots of policies in the policy buffer. By recognizing the iteration of training as a hyperparameter in our experiments we've managed to retrieve the overridden strategies that yield high rewards in the target environments due to their generalization capacity. For instance, we proved that a hopper robot is capable of performing forward locomotion in an unknown environment 1.75 times the source task's gravity using the policies saved at earlier iterations.

We provided comparisons of RARL algorithms trained with different critic structures, curriculum learning, and entropy bonus and showed how the choice of training affect the generalization capacity for Hopper task. We proposed the ACC-RARL algorithm as a new critic structure and showed that it increased performance significantly in harder torso mass tasks.

Furthermore, we introduce strict clipping for Proximal Policy Optimization (PPO) [31] as a regularization technique. Using an unconventionally low clipping parameter we discarded the samples that overfit the source task namely the standard humanoid environment. We observed higher jumpstart performance in humanoid environments with higher tangential friction, a larger range of gravity and morphological modifications using the robust policies saved during training. Decreasing the Kullback–Leibler divergence constraint for Trust Region Policy Optimization (TRPO) is a future research direction we would like to explore.

Although outside the scope of transfer learning, we've discovered that decaying the clipping parameter decreases final policy performance for the humanoid environment that has higher state, action space. The transfer learning algorithms we use are based on the state-of-art algorithms built for continuous control thus this finding had a substantial effect on our experiments.

We believe that the first step of determining the most promising policy parameters lies in the accurate parametrization of the environment. We would like to investigate the relationship between the parametrized distance between environments and the policies residing in the buffer. This unknown mapping might be depicted as a nonlinear function approximator and should be estimated with the least amount of data possible.

In this thesis, we showed the necessity of hyperparameter tuning to increase the generalization capacity of the transferred policy. However, in some cases we proved that a median task between the source task and the target task might not always give us an idea of source task performance. Accordingly, we believe designing a better surrogate validation task is a fruitful future research direction.

REFERENCES

1. Rajeswaran, A., S. Ghotra, B. Ravindran and S. Levine, "Epopt: Learning robust neural network policies using model ensembles", *arXiv preprint arXiv:1610.01283*, 2016.

2. Parisotto, E., J. L. Ba and R. Salakhutdinov, "Actor-mimic: Deep multitask and transfer reinforcement learning", *arXiv preprint arXiv:1511.06342*, 2015.

3. Finn, C., P. Abbeel and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks", *arXiv preprint arXiv:1703.03400*, 2017.

4. Apgar, T., P. Clary, K. Green, A. Fern and J. Hurst, "Fast Online Trajectory Optimization for the Bipedal Robot Cassie", *Robotics:ScienceandSystems2018 Pittsburgh,PA,USA,June26-30,2018*, Vol. 14, p. 54, 2018.

5. Xie, Z., G. Berseth, P. Clary, J. Hurst and M. van de Panne, "Feedback Control For Cassie With Deep Reinforcement Learning", *arXiv preprint arXiv:1803.05580*, 2018.

6. Sutton, R. S. and A. G. Barto, *Reinforcement learning: An introduction*, MIT Press, 2018.

7. Ng, A., "Lecture 1.3—Recurrent Neural Networks: Recurrent Neural Network Model", COURSERA: Sequence Models, 2018.

8. Dong, Z., "*Tensorflow implementation for Robust Adversarial Reinforcement Learning*", 2018, `https://github.com/Jekyll1021/RARL`, accessed in March 2019.

9. LeNail, A., "NN-SVG: Publication-Ready Neural Network Architecture Schematics", `https://doi.org/10.21105/joss.00747`, 2019.

10. Pinto, L., "*Rllab implementation for Robust Adversarial Reinforcement Learning*", 2017, `https://github.com/lerrel/rllab-adv`, accessed in December 2018.

11. Mnih, V., K. Kavukcuoglu, D. Silver *et al.*, "Human-level control through deep reinforcement learning", *Nature*, Vol. 518, No. 7540, pp. 529–533, Feb. 2015, `http://dx.doi.org/10.1038/nature14236`.

12. Abbeel, P. and A. Y. Ng, "Apprenticeship learning via inverse reinforcement learning", *Proceedings of the Twenty-first international Conference on Machine learning*, p. 1, ACM, 2004.

13. Cobbe, K., O. Klimov, C. Hesse, T. Kim and J. Schulman, "Quantifying generalization in reinforcement learning", *arXiv preprint arXiv:1812.02341*, 2018.

14. Henderson, P., W.-D. Chang, F. Shkurti, J. Hansen, D. Meger and G. Dudek, "Benchmark environments for multitask learning in continuous domains", *arXiv preprint arXiv:1708.04352*, 2017.

15. Pinto, L., J. Davidson, R. Sukthankar and A. Gupta, "Robust adversarial reinforcement learning", *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 2817–2826, JMLR. org, 2017.

16. Zhang, C., O. Vinyals, R. Munos and S. Bengio, "A study on overfitting in deep reinforcement learning", *arXiv preprint arXiv:1804.06893*, 2018.

17. Shioya, H., Y. Iwasawa and Y. Matsuo, "Extending Robust Adversarial Reinforcement Learning Considering Adaptation and Diversity", *International Conference on Learning Representations*, 2018, `https://openreview.net/forum?id=BJ7Upwyvf`.

18. Todorov, E., T. Erez and Y. Tassa, "Mujoco: A physics engine for model-based control", *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pp. 5026–5033, IEEE, 2012.

19. Williams, R. J., "Simple statistical gradient-following algorithms for connectionist reinforcement learning", *Machine Learning*, pp. 229–256, 1992.

20. Minsky, M., "Steps toward artificial intelligence", *Proceedings of the IRE*, Vol. 49, No. 1, pp. 8–30, 1961.

21. He, K., X. Zhang, S. Ren and J. Sun, "Deep residual learning for image recognition", *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

22. Graves, A., A.-r. Mohamed and G. Hinton, "Speech recognition with deep recurrent neural networks", *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pp. 6645–6649, IEEE, 2013.

23. Zupan, J., "Introduction to artificial neural network (ANN) methods: what they are and how to use them", *Acta Chimica Slovenica*, Vol. 41, pp. 327–327, 1994.

24. Weston, J. and C. Watkins, "Support vector machines for multiclass pattern recognition", *Proceedings of the Seventh European Symposium On Artificial Neural Networks*, 4 1999.

25. Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting", *The Journal of Machine Learning Research*, Vol. 15, No. 1, pp. 1929–1958, 2014.

26. Sutskever, I., J. Martens, G. Dahl and G. Hinton, "On the importance of initialization and momentum in deep learning", *International conference on machine learning*, pp. 1139–1147, 2013.

27. Nesterov, Y. E., "A method for solving the convex programming problem with convergence rate O (1/kˆ 2)", *Dokl. akad. nauk Sssr*, Vol. 269, pp. 543–547, 1983.

28. Tieleman, T. and G. Hinton, "Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude", COURSERA: Neural Networks for Machine Learning, 2012.

29. Kingma, D. P. and J. L. Ba, "Adam: Amethod for stochastic optimization", *Proc. 3rd*

Int. Conf. Learn. Representations, 2014.

30. Dhariwal, P., C. Hesse, O. Klimov *et al.*, "*OpenAI Baselines*", 2017, `https://github.com/openai/baselines`, accessed in May 2018.

31. Schulman, J., F. Wolski, P. Dhariwal, A. Radford and O. Klimov, "Proximal policy optimization algorithms", *arXiv preprint arXiv:1707.06347*, 2017.

32. Fujita, Y. and S.-i. Maeda, "Clipped Action Policy Gradient", *arXiv preprint arXiv:1802.07564*, 2018.

33. Konda, V. and J. Tsitsiklis, "Actor-Critic Algorithms", *Advances in Neural Information Processing Systems 12*, pp. 1008–1014, 2000.

34. Schulman, J., S. Levine, P. Abbeel, M. Jordan and P. Moritz, "Trust region policy optimization", *International Conference on Machine Learning*, pp. 1889–1897, 2015.

35. Schulman, J., P. Moritz, S. Levine, M. Jordan and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation", *arXiv preprint arXiv:1506.02438*, 2015.

36. Alpaydın, E., *Yapay öğrenme*, Boğaziçi Üniversitesi Yayınevi, 2011.

37. Mnih, V., A. P. Badia, M. Mirza *et al.*, "Asynchronous methods for deep reinforcement learning", *International conference on machine learning*, pp. 1928–1937, 2016.

38. Torrey, L. and J. Shavlik, "Transfer learning", *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, pp. 242–264, IGI Global, 2010.

39. Taylor, M. E. and P. Stone, "Transfer learning for reinforcement learning domains: A survey", *Journal of Machine Learning Research*, Vol. 10, No. Jul, pp. 1633–1685, 2009.

40. Narvekar, S., J. Sinapov and P. Stone, "Autonomous Task Sequencing for Customized

Curriculum Design in Reinforcement Learning.", *IJCAI*, pp. 2536–2542, 2017.

41. Karpathy, A., F. Li and J. Johnson, "CS231n Convolutional Neural Network for Visual Recognition,"", *Online Course*, 2016.

42. Goldberg, Y., "Neural network methods for natural language processing", *Synthesis Lectures on Human Language Technologies*, Vol. 10, No. 1, pp. 1–309, 2017.

43. Rusu, A. A., N. C. Rabinowitz, G. Desjardins *et al.*, "Progressive neural networks", *arXiv preprint arXiv:1606.04671*, 2016.

44. Cao, Z., M. Long, J. Wang and M. I. Jordan, "Partial Transfer Learning With Selective Adversarial Networks", *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

45. Russakovsky, O., J. Deng, H. Su *et al.*, "Imagenet large scale visual recognition challenge", *International Journal of Computer Vision*, Vol. 115, No. 3, pp. 211–252, 2015.

46. Griffin, G., A. Holub and P. Perona, *Caltech-256 Object Category Dataset*, Tech. Rep. 7694, California Institute of Technology, 2007, `http://authors.library.caltech.edu/7694`.

47. Tobin, J., R. Fong, A. Ray, J. Schneider, W. Zaremba and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world", *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, pp. 23–30, IEEE, 2017.

48. Sadeghi, F. and S. Levine, "CAD2RL: Real single-image flight without a single real image", *arXiv preprint arXiv:1611.04201*, 2016.

49. Blender Online Community, *Blender - a 3D modelling and rendering package*, Blender Foundation, Blender Institute, Amsterdam, 2002, `http://www.blender.org`.

50. Bousmalis, K., A. Irpan, P. Wohlhart *et al.*, "Using simulation and domain adaptation to

improve efficiency of deep robotic grasping", *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4243–4250, IEEE, 2018.

51. Tzeng, E., C. Devin, J. Hoffman *et al.*, "Adapting deep visuomotor representations with weak pairwise constraints", *arXiv preprint arXiv:1511.07111*, 2015.

52. Tamar, A., Y. Glassner and S. Mannor, "Optimizing the CVaR via Sampling.", *AAAI*, pp. 2993–2999, 2015.

53. Pinto, L., "*Gym environments with adversarial disturbance agents*", 2017, `https://github.com/lerrel/gym-adv`, accessed in March 2019.

54. Duan, R., P. Chen, Houthooft, Rein, J. Schulman and P. Abbeel, "*rllab*", 2016, `https://github.com/rll/rllab`, accessed in December 2018.

55. Bansal, T., J. Pachocki, S. Sidor, I. Sutskever and I. Mordatch, "Emergent complexity via multi-agent competition", *arXiv preprint arXiv:1710.03748*, 2017.

56. Al-Shedivat, M., T. Bansal, Y. Burda, I. Sutskever, I. Mordatch and P. Abbeel, "Continuous adaptation via meta-learning in nonstationary and competitive environments", *arXiv preprint arXiv:1710.03641*, 2017.

57. WEI, Y., Y. Zhang, J. Huang and Q. Yang, "Transfer Learning via Learning to Transfer", J. Dy and A. Krause (Editors), *Proceedings of the 35th International Conference on Machine Learning*, Vol. 80 of *Proceedings of Machine Learning Research*, pp. 5085–5094, PMLR, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018, `http://proceedings.mlr.press/v80/wei18a.html`.

58. Nichol, A., V. Pfau, C. Hesse, O. Klimov and J. Schulman, "Gotta Learn Fast: A New Benchmark for Generalization in RL", *arXiv preprint arXiv:1804.03720*, 2018.

59. Ravi, S. and H. Larochelle, "Optimization as a model for few-shot learning", *International Conference on Learning Representations*, 2017.

60. Mishra, N., M. Rohaninejad, X. Chen and P. Abbeel, "A Simple Neural Attentive Meta-Learner", *International Conference on Learning Representations*, 2018, https://openreview.net/forum?id=B1DmUzWAW.

61. Nichol, A. and J. Schulman, "Reptile: a Scalable Metalearning Algorithm", *arXiv preprint arXiv:1803.02999*, 2018.

62. Duan, Y., J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever and P. Abbeel, "RLˆ 2: Fast Reinforcement Learning via Slow Reinforcement Learning", *https://arxiv.org/pdf/1611.02779.pdf*, 2016.

63. Frans, K., J. Ho, X. Chen, P. Abbeel and J. Schulman, "Meta learning shared hierarchies", *arXiv preprint arXiv:1710.09767*, 2017.

64. Henderson, P., W.-D. Chang, P.-L. Bacon, D. Meger, J. Pineau and D. Precup, "Optiongan: Learning joint reward-policy options using generative adversarial inverse reinforcement learning", *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

65. Henderson, P., R. Islam, P. Bachman, J. Pineau, D. Precup and D. Meger, "Deep reinforcement learning that matters", *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

66. Huang, Y., C. Gu, K. Wu and X. Guan, "Reinforcement Learning Policy with Proportional-Integral Control", *International Conference on Neural Information Processing*, pp. 253–264, Springer, 2018.

67. Peng, X. B., P. Abbeel, S. Levine and M. van de Panne, "DeepMimic: Example-guided Deep Reinforcement Learning of Physics-based Character Skills", *ACM Trans. Graph.*, Vol. 37, No. 4, pp. 143:1–143:14, Jul. 2018, http://doi.acm.org/10.1145/3197517.3201311.

68. Bajrami, X., A. Dermaku, R. Likaj, N. Demaku, A. Kikaj, S. Maloku and D. Kikaj, "Trajectory planning and inverse kinematics solver for real biped robot with 10 DOF-s",

IFAC-PapersOnLine, Vol. 49, No. 29, pp. 88–93, 2016.

69. Thavai, R., "Inverse Kinematics Solution for Biped Robot", *IOSR Journal of Mechanical and Civil Engineering (IOSR-JMCE)*, Vol. 12, pp. 57–62, 2015.

70. Vukobratović, M. and B. Borovac, "Zero-moment point—thirty five years of its life", *International journal of humanoid robotics*, Vol. 1, No. 01, pp. 157–173, 2004.

71. Robotics, A., "*Cassie*", `http://www.agilityrobotics.com/robots#cassie`, accessed in September 2018.

72. Carpentier, J., M. Benallegue and J.-P. Laumond, "On the centre of mass motion in human walking", *International Journal of Automation and Computing*, Vol. 14, No. 5, pp. 542–551, 2017.

73. Zatsiorsky, V. M. and V. M. Zaciorskij, *Kinetics of human motion*, Human Kinetics, 2002.

74. Bengio, Y., J. Louradour, R. Collobert and J. Weston, "Curriculum learning", *Proceedings of the 26th annual international conference on machine learning*, pp. 41–48, ACM, 2009.

75. OpenAI, "*Request for Research: Multitask RL with Continuous Actions*", `https://openai.com/requests-for-research/#multitask-rl-with-continuous-actions`, accessed in November 2018.

76. Brockman, G., V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang and W. Zaremba, "OpenAI Gym", *arXiv preprint arXiv:1606.01540*, 2016.

77. Clavera, I., A. Nagabandi, R. S. Fearing, P. Abbeel, S. Levine and C. Finn, "Learning to Adapt: Meta-Learning for Model-Based Control", *arXiv preprint arXiv:1803.11347*, 2018.

78. Erez, T., Y. Tassa and E. Todorov, "Simulation tools for model-based robotics: Compar-

ison of bullet, havok, mujoco, ode and physx", *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pp. 4397–4404, IEEE, 2015.

79. Abadi, M., A. Agarwal, P. Barham *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems", *arXiv preprint arXiv:1603.04467*, 2016.

80. Hunter, J. D., "Matplotlib: A 2D graphics environment", *Computing In Science & Engineering*, Vol. 9, No. 3, pp. 90–95, 2007.

81. Prechelt, L., "Early stopping-but when?", *Neural Networks: Tricks of the trade*, pp. 55–69, Springer, 1998.