Multi Domain and Multi Task Deep Reinforcement Learning for Continuous Control

David Camilo Alvarez Charris

Email: david13.ing@gmail.com

Master of Science

Artificial Intelligence

School of Informatics

University of Edinburgh

2018

Abstract

Despite the success of recent advances in Deep Reinforcement Learning (DRL) in areas of robotic manipulation, game playing and control, most DRL agents are trained to maximize a unique reward function that solves a single task, or are trained within a environment with a single setting (a unique gravity, game configuration or sensor noise). This type of training is not only limited to solving one task at the time, but also ignores the potential benefit that training across parallel tasks can have in improving learning speed and sample efficiency.

In this project we examine the problem of how an agent learning from experience - with Reinforcement Learning - can achieve high performance across multiple tasks or domains learnt in parallel. We also analyze if this type of training poses any benefit when compared to single task learning agents. To assess our hypothesis a set of Neural Networks which serve as Value Function and Policy approximators were implement [1] as single task (STL) learning agents. We also propose a hard parameter sharing architecture - called Multi Headed Network - capable of being trainned for muti task (MTL) learning problems. State of the art methods such as Proximal Policy Optimization and Generalized Advantage Estimators are used to train the functions approximators through policy gradient methods.

Our results demonstrate that the Multi Headed Network was cable to achieve a high performance across all domains and tasks in the Bipedal Walker and Lunar Lander training environments. For some domains, the proposed multi domain learning architecture produced a 4.981% improvement on the asymptotic reward and a 13.940 % improvement on the sample efficiency when compared to single domain learning agents.

[1]The code for this project can be found in: https://github.com/david1309/Multi_Task_RL. For the videos of the trainned agents go to: https://www.youtube.com/playlist?list=PLe5MBbUvqqF_Bb7eiwLHhzE6qVeY5P3-X

Acknowledgements

As this dissertation marks the end of a year full of academic, cultural, and ”self understanding” growth, I would first like to thank my parents Marlen and Leonardo as they play an essential role in this project - always being so involved in my life, always supportive of my dreams and crazy endeavours, and always making me feel so loved and happy. I am also grateful for the love, support, and joy given by my partner, Camila, she has been an important part of my personal and academic projects and a backbone to my life for half a decade :).

I am thankful with professor Timothy Hospedales and Chenyang Zhao for kindly supporting my self-proposed project, and for their advice on which research directions to focus my project in, help with technical difficulties, and all the great creative ideas that have given me new insights.

I would like to thank professors Iain Murray and Michael Gutmann , despite not being involved in this project they did have a important impact on how I viewed and understood my dissertation, and they have greatly influenced the future directions I intend to focus my research in.

Last but not least (and this is true!), I am extremely grateful to all the kind persons who donated to my Crowd funding campaign; without their trust and support this whole experience would not have been possible. I am specially thankful to Gloria Frisneda, who has also inspired me with her words, kindness and supported me with her donations, and with Fanny Lopez for her compassionate support. Special thanks to Haitham Baomarhaith, my friend Emmanuel Cordoba, and to Keziah Conroy - as she was the one who inspired be to start my crowd funding campaign, and thus, a key player in making this MSc happen. I am also grateful with COLFUTURO and The University of Edinburgh for awarding me the Colfuturo-Edinburgh Scholarship; without this financial support all of this would had not been feasible.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*David Camilo Alvarez Charris*
Email: david13.ing@gmail.com)

Table of Contents

Chapter 1

Introduction

Animals with higher cognitive skills are able to exhibit impressive decision making and learning capabilities, in which by interacting with the world they learn how to achieve goals and maximize their utility. For example, a bird learns how to use sticks as tools to reach the insects it wants to eat, a baby gradually learns how to walk in order to move towards its parents or food, or a adult human becomes addicted to slot machines because of the potential reward (money) it could obtained. All of this examples can bee seen as learning from experience to maximize some notion of reward or utility.

Reinforcement Learning (RL) is a mathematical and algorithmic frameworks to describe the process of interacting with the world (or a simulation), and learning how to act optimally in it. By modeling the world in a RL framework a machine interprets it as as a series of states and action, in which by interacting with each state it can potentially learn the optimal mapping between which action to choose for a given state while maximizing the rewards or utility it obtains. As stated in my previous work on the IPP report (Alvarez-Charris, 2018a), Deep Reinforcement Learning (DRL) in which such mapping is parametrized by a Artificial Neural Network (ANN) with many hidden layers, has successfully been used for a variety of robotic , video and board games, and control problems (Mnih et al., 2015; Levine et al., 2016). Using DRL researches have successfully enabled robots to learn control policies for screwing caps on bottles or folding clothes, and it has also been used to enable algorithms to achieve human level performance in games like Dota, ATARI and GO. Despite the success of DRL on the above mentioned examples, DRL is used for achieving high performance in a single task or domain. This approach falls short in more challenging scenarios where an agent must learn how to achieve a good performance across several tasks in parallel.

To illustrate this more challenging problem of learning multiple tasks, lets continue with the example mentioned before. A baby not only learns to walk in a smooth and perfectly flat floor, it also develops motor skills that allow him to walk in diverse situations - inclined terrains, slippery surfaces, and even walking when exposed to different forces such as a rucksack on his back or a heavy wind pushing him backward. By learning in multiple domains, the baby is not only able to solve multiple problems but might also learn faster given that it is exposed to multiple experiences. This type of learning falls under the set of problems of Multi Domain (MDL) and Multi Task Learning (MTL) - a set of problems in which most agents trained with standard DRL are incapable of achieving this parallel task/domain learning.

1.1 Motivation and Objectives

In this project we intend to explore the research question: *"Can agents trained through Reinforcement Learning jointly achieve high performance across multiple continuous control tasks/domains ?"* - and - *Does the parallel training across domains / task pose any benefit as measured by some performance metric ?*. Based on this the main objective of this project is to compare if training a single MDL/MTL agent in parallel across multiple domains / tasks poses any benefit - in terms of the mean episode total reward, final (asymptotic) reward or sample efficiency - when compared to the performance exhibited by a SDL/STL agent individually trained on each task.

Our hypothesis is that *"By using alternative function approximation models with shared parameters and training them in parallel across different domains / tasks, a Reinforcement Learning agent can learn appropriate policy(s) for achieving high performance throughout all tasks"*.

During this project several pre existing reinforcement learning methods and models were implemented in Python, along with our proposed architecture capable of exhibiting MTL.The contributions of this projects are the following:

- Implementation of baseline single domain/task agents (STL) using non-linear functions approximators and state of the art policy optimization algorithms.
- Proposal of a non-linear function approximator with the capacity of being trained in parallel across multiple domains/task (MTL), and optimized by policy search algorithms.

- Training of STL and MTL agents across multiple environments of the Open AI Gym [1], along with the modification of such environments for domain changes and creation of new tasks.
- Thorough comparison of the performance between MTL and STL agents , as well as the benefits and problems that arise during parallel task training.

1.2 Document Outline and Results

Our results indicate that the proposed MDL/MTL architecture is capable of learning a policy able to achieve a high mean reward across multiple domains when learning motor policies for walking with a simulated robot and learning multiple tasks for flying and landing a spacecraft. In some experiments, the parallel task training did increase the agents total reward compared to a single task agent and it also improved the agents learning speed / sample efficiency. Even though the MDL and MTL agents were indeed able to learn in all domains and tasks, we did not see the additional benefits in some of the tasks we explored and in most of the multi domain experiments.

This documents is organized as follows: In **Chapter. 2** we introduce general concepts of Reinforcement learning and previous works related to MTL. **Chapter. 3** is the main chapter of this project extensively describing the methodology used to develop the single and multi domain function approximators and policy optimization algorithms, our proposed architecture, and the results of the experiments. In **Chapter. 4** we extend the architecture for multi tasks learning problems and analyze the results obtained. Finally in **Chapter. 5** we summarize the work done in this dissertation, analyzing the short comings of our proposals and plausible future directions.

How to read this document:

Readers not familiar with Reinforcement Learning and gradient based optimization should start from the initial **Chapter. 2** . For readers already familiar with such concepts but not with the use of Neural Networks as policies, and Proximal and Trust Region policy optimization methods, please refer to **section. 3.2.1** and **section. 3.2.3** . For advanced readers, **section. 3.2.2** introduces the proposed architecture and all sections following **section. 3.3.1** show the multi domain and multi task results.

[1] `https://gym.openai.com/`

Chapter 2

Background

2.1 Reinforcement Learning

Reinforcement Learning is a framework in which the world is modeled as a Markov Decision Process (MDP) made out of a set of states $\boldsymbol{S}$ and actions $\boldsymbol{A}$. It is "Markov" because we assume that the world complies with the Markov Property, which means that future states are independent of the past - the complete history of previously visited states- given the present. This implies that the present state s_t is a sufficient statistic, a summary, of all the previous history. In mathematical terms, *"The conditional probability distribution of future states of the process only depends upon the present state, not on the sequence of events that preceded"*; which implies the conditional independence given by $P(s_{t+1}|s_t, s_{t-1}, ..., s_{t-N}) = P(s_{t+1}|s_t)$. It is a "Decision Process" given that in this framework a agent receives as input a state s_t and needs to *decide* which action $a_t \in \boldsymbol{A}$ to take. Such actions are usually taken in order to maximize a scalar reward signal given by r_t. The interaction between the states and the agents creates the agent-environment loop **fig. 2.1**, in which the agent receives an observation s_t, acts with action a selected according to a policy denoted by π, and then the environment returns a new observation s_{t+1} and the reward signal associated with taking action a at state s_t.

The agents objective is usually not to maximize the reward at a particular time step, but rather maximizing the expected sum of (discounted) rewards obtained by acting in the environment throughout a complete trajectory τ of length T. This sum is referred to as the return denoted by $R(\tau)$ (**2.1**), and the discount factor γ controls how important are the rewards received in each future time step. The overall objective of RL is to find

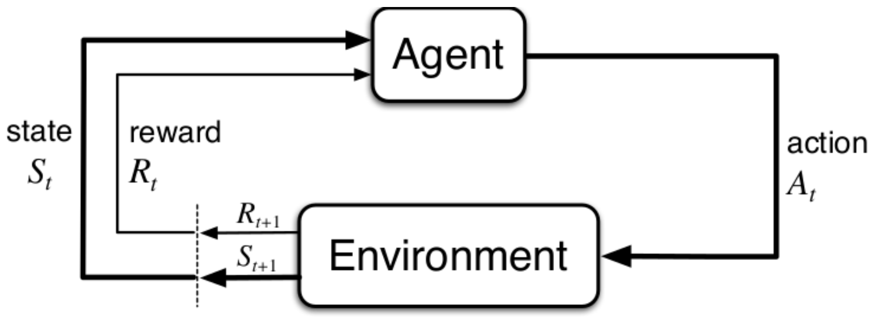

Figure 2.1: *Agent-Environment loop depicting the interaction between the two entities.*

the best policy π that can achieve this objective; to do so we can make use the following relevant functions of RL.

$$R(\tau) = \sum_{t=0}^{T-1} \gamma^t r_t \tag{2.1}$$

$$V_\pi(s) = \mathbb{E}_\pi\Big[R_t | S_t = s\Big] = \sum_a \pi(s,a) \sum_{s'} P^a_{s,s'}\Big[R^a_{s,s'} + \gamma . V_\pi(s')\Big] \tag{2.2}$$

$$Q_\pi(s,a) = \mathbb{E}_\pi\Big[R_t | S_t = s, A_t = a\Big] = \sum_{s'} P^a_{s,s'}\Big[R^a_{s,s'} + \gamma . \sum_{a'} \pi(s',a') . Q_\pi(s',a')\Big] \tag{2.3}$$

The State Value Function $V_\pi(s)$ (**2.2**) maps a state to a numerical value describing the "long-term" value or "goodness" of a state under a certain policy. With this function we can further compute the State-Action Value function $Q_\pi(s,a)$ (**2.3**) which measures how good is to take a particular action a in a given state s under a certain policy. In the equations for this functions $P^a_{s,s'}$ is a transition function of the environment determining the probability of, through action a transitioning from one state to another, and $R^a_{s,s'}$ is a function determining the reward r_t obtained by the agent through this transition (i.e $p(s' \mid s,a)$ and $p(r_t \mid s,s',a)$) . Assuming that we have computed the Q-function for a given environment, we can obtained a greedy policy by simply selecting the best action at each state $\pi(s) = \underset{a}{\operatorname{argmax}}\, Q_\pi(s,a)$.

Although computing the Q-function can be challenging given that $P^a_{s,s'}$ and $R^a_{s,s'}$ are part of the environment and unknown; and for large state-actions dimensions computing the value functions might not be feasible, there are several methods for computing $V_\pi(s)$ and/or $Q_\pi(s,a)$. However, in this project we will focus on methods which explicitly find a policy $\pi(s,a)$ *directly* mapping state to actions without the need of using intermediate value or action-value functions. Such policy can be obtained with the models described in **section 3.2.1** .

2.2 Multi Domain and Multi Task Learning

As the entirety of this projects is concerned with Multi Domain Learning (MDL) and Multi Task Learning (MTL) and there is no clear consensus concerning what is a different domain or a different task, here we present how we define each.

In general, Multi ”X” Learning refers to the problem of learning to solve multiple problems at the same time - in parallel - in order to exploit the commonalities and difference across this tasks to obtain some benefit. This benefit is usually manifested as improvements on the classifications accuracy, learning efficiency, or in our case the average sum of rewards. In Multi ”X” Learning we are equally interested in achieving high performance in all problems without distinction, there is no particular preference for performance in a particular problem (Zhang and Yang, 2017)

We define that a environment has a different *Domain* when the structure of the MDP has changed regarding differences in the state transitions probabilities or changes in the observations / inputs / sensors that the agent receives. For example, increasing the gravity in a environment makes the transition probability of going from the state ”standing” to the state ”falling” vary; or changing the type of camera a robot uses to observe the world would also change the nature of the MDP's states and thus constitute a change in domain. Under this definition one can view MDL as an agent trying to solve the same task but under different conditions.

Different *Tasks* refer to when the reward functions describing the desirability of each state is changed. For example, learning to walk or jump or balance in one leg constitute different tasks given that different rewards are being maximized. Because of this, in MTL the agent is trying to solve significantly different tasks which have some commonalities.

2.3 Gradient Based Optimization

In this project we extensively make use of optimization methods for finding good policies π_θ that enable agents to act in the simulation environments. Although not essential for understanding RL, MDL or MTL; I believe its important to have a basic understanding of the class of optimization methods used in this project. This sections is devoted for such purpose.

The problem of fitting a function or model $f(x)$ to data (x,y) is encountered in a wide variety of scenarios. For example, x could be data describing the amount of rain of the last five days, y how sunny it has been for the last five days, and fitting $f(x)$ to produce a forecast of how sunny it will be for the following two days, or x being images from different human faces, y a binary variable describing the gender of each individual, and $f(x)$ classifying each human face as women or men. In the context of RL x usually corresponds to the set of all possible (or sampled) states of the environment $\boldsymbol{S}$ and we want to find some $f(\boldsymbol{S})$ which estimates as close as possible the true value of each state $V(s)$, the state-action value $Q(s)$ or which performs a direct mapping to an action $\pi(a \mid s)$.

A function or model is composed of a set of parameters, for example $f(x;m,b) = m.x + b$ is parametrized by the set $\theta = \{m,b\}$, or $f(x;\mu,\sigma) = e^{\frac{-(x-\mu)^2}{\sigma^2}}$ is parametrized by the set $\theta = \{\mu,\sigma\}$. From here on the notation $f(x;\theta)$ or $f_\theta(x)$ will denote that function $f(x)$ is parametrized by the set of parameters θ. In function fitting we attempt to find the best set θ that makes $f(x;\theta)$ as close as possible to the true data y. Although there exist a wide variety of methods for function fitting, when we can compute the partial derivatives of $f(x;\theta)$ with respect to θ for all possible $x \in X$, gradient based methods can be used and have the potential to produce excellent results.

As mentioned before we want to make our function ”as close as possible” to the true data - the notion of ”closeness” is measured by a objective function that is a dependant of both f_θ and y, such function is denoted as $J(\theta)$.

We can iteratively find good parameters in the following way: Initialize all parameters $\theta_{t=0}$ (either randomly or following some model specific procedure such as that described at the end of **section 3.2.1**), compute the prediction made by $f_{\theta_{t=0}}$ either for a single input data (in which case it is called Stochastic Gradient Descent (SGD)), a small subset of the data (Mini-batch Gradient Descent) or for all the data X (Batch Gradient Descent). Given predictions $f_{\theta_{t=0}}$ and target values y we compute the value of the objective function $J(\theta_{t=0})$ for the current set of parameters **(2.4)** . At this point what we wish is to modify *each* parameter in the direction that improves our models performance as measured by $J(\theta)$. Such direction is naturally given by the gradient **(2.5)** of the objective function - the gradient is the vector of partial derivatives of the objective function with respect to each parameter and it indicates the direction in which the functions increases. If a bigger J corresponds to a better set of parameters we want to maximize this function and thus shift the parameters in the gradients direction (Gradient

Ascent). In contrast if smaller J indicate better parameters we intend to minimize J and will move in the direction of the negative gradient (Gradient Descent, described by (2.6) and depicted in **fig.** 2.2). We will repeat this update procedure until we reach the global optimum of $J(\theta)$ or for a number of steps T.

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}(y_{(i)} - f_{\theta}(x_{(i)}))^2 \tag{2.4}$$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \left[\frac{\partial J\boldsymbol{\theta}}{\partial \theta_1}, \frac{\partial J\boldsymbol{\theta}}{\partial \theta_2}, \ldots, \frac{\partial J\boldsymbol{\theta}}{\partial \theta_D}\right] \tag{2.5}$$

$$\boldsymbol{\theta}_{t+1} \leftarrow \theta_t - \alpha.\nabla_{\boldsymbol{\theta_t}} J(\boldsymbol{\theta_t}) \tag{2.6}$$

Given that the estimate of the gradient can be noisy (as in SGD) and that our model has a limited capacity preventing it from exactly achieving the maximum/ minimum of the objective function, we only make a small step α in the direction of greatest ascent/ descent (2.6). If α is sufficiently small then $J(\theta_t + 1) \leq J(\theta_t)$ in the case of gradient descent. The equation shown above describes the gradient descent (GD) procedure for fitting $f_{\boldsymbol{\theta}}$, parametrized by a a D-dimensional set of parameters $\boldsymbol{\theta}$ to data (X, Y) which corresponds to N - data points. With $n = 1$ we would have a SGD procedure, for $n < N$ a Mini-batch GD and for $n = N$ Batch GD. The objective function in this example is the mean square error (MSE) measuring the the average squared deviation between the predicted and ground truth values.

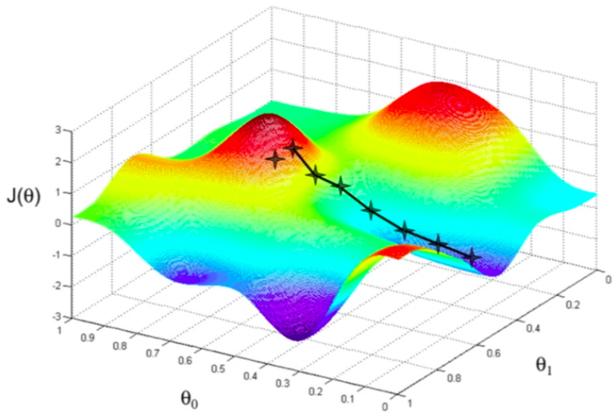

Figure 2.2: *Steps of a gradient descent optimization algorithm for fitting a function $f_{\boldsymbol{\theta}}$ with two parameters $\boldsymbol{\theta} = \{\theta_0, \theta_1\}$. The coloured surface represents the objective function $J(\boldsymbol{\theta})$ and the black stars and line illustrate each of the steps updating the parameters until a local optimum is reached (taken from CS229 Machine Learning Ng (2018))*

2.4 Weight Sharing Across Networks

As for my IPP report I extensively described the previous work related to this project, the following sections are taken verbatim from my IPP (Alvarez-Charris, 2018a)

One approach to achieve MTL in Neral Networks (NN) has been using weight sharing across multiple NN or a single large NN. For instance, Yu et al. (2017) introduced a method in which initially a single NN is jointly trained across multiple tasks in order for it to obtained a common representation across tasks. After the joint training, for each weight in the NN a ”specialization metric” is computed - this metrics determines if the weight should be shared across the different tasks or if it should undertake a second training phase (specialization training) for it to focus on a single task. The metric is based on the gradient of the Proximal Policy Optimization (PPO) loss function with respect to each parameter; this gradient is computed for each tasks and if the variance across all tasks is low this will indicate that most tasks agree on the value of the parameter and it can thus be shared. If the variance is high, the weight is separated to a new NN structure and is further trained for specialization. The final NN architecture consist N - sub networks which have common weights (the jointly trained ones) and differ on the weights that undertook the specialization (see bottom left of **fig.2.3**)

Yang and Hospedales (2015) introduce a NN architecture capable of exhibiting MTL via the use of semantic descriptors. The descriptors $z^{(i)}$ for task i are multivariate representations (a vector or a tuple) containing relevant information useful to indicate which task/domain is currently being trained or tested. The authors are able to combine each training example feature vector and semantic descriptor by constructing a two-sided neural network (see top of **fig.2.3**) that minimizes **(2.7)**.

$$\frac{1}{M}\sum_{i}^{M}\frac{1}{N}\sum_{j}^{N_i}\mathcal{L}\Big(\hat{y}_j^{(i)}, y_j^{(i)}\Big) \quad ; \quad \hat{y}_j^{(i)} = f_P(x_j^{(i)}).g_Q(z^{(i)}) \tag{2.7}$$

One NN labeled P operates in the usual way by receiving as input the feature vector x_j and outputting a prediction $f_P(x_j^{(i)})$ for task i. A second NN labeled Q performs another prediction $g_Q(z^{(i)})$ based on the semantic descriptor of the training example. The final prediction is computed by the matrix product of the two NN. In this way, the two-sided NN minimizes the loss across alk M tasks and given the semantic descriptor it is able to achieve high performance across multiple tasks.

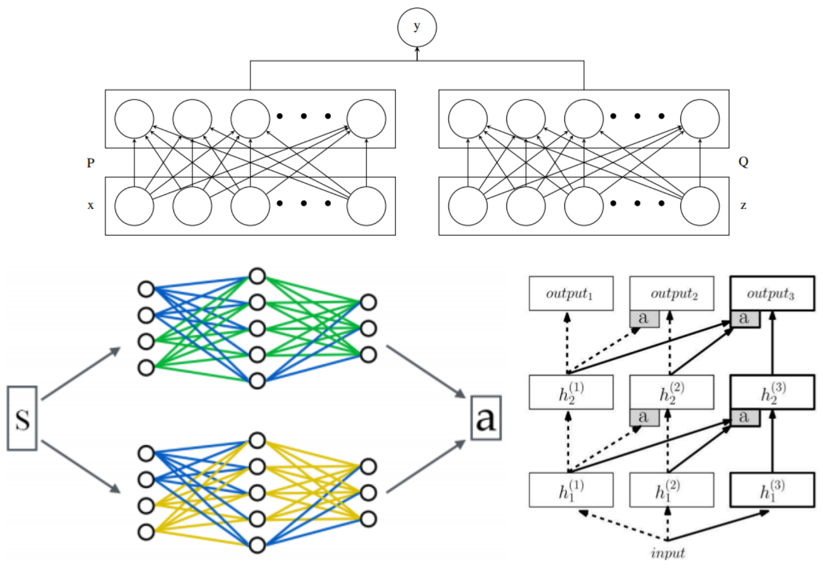

Figure 2.3: *MTL Neural Network Architectures. Top: Two-sided Neural Network combining feature vector predictions P with predictions made by a semantic description network Q which is specific for each task. Bottom Left: Guided policy NN illustrating an architecture in which N networks are created with shared weights (in Blue) and task specific specialized weights (in Green and Yellow). Bottom Right: Progressive neural network in which each column specializes in a single tasks, and lateral layer wise connections across columns enable transfer learning. (taken from Yang and Hospedales (2015); Yu et al. (2017); Rusu et al. (2016))*

2.5 Progressive Learning

Note that it is outside the scope of this project to assess problems raised when multiple tasks are learnt in sequence (Live Long Learning) or when learning a new task can be accelerated by knowledge transfer from a pre-trained agent (Transfer Learning). However, the work done in this areas is partially related to the MTL problem and because of this we include some background about it.

Rusu et al. (2016) tackle both the problem of MTL and transfer learning by using a progressive NN. In this paper MTL is achieved by creating a new NN (a ”column”, see bottom right of **fig.2.3**) for each new task and keeping intact already trained columns by ”freezing” its parameters. Using this approach the authors avoid the common problem of catastrophic interference, in which previously learnt knowledge is overwritten and forgotten when new tasks are learnt. Transfer learning is achieved in this columnar architecture by creating layer wise lateral connections between each column. The lateral connections are achieved through (**2.8**), in which the activation’s at layer *i* of column k are computed based on the sum between: the previous layer activation's multiplied by the weights of the *ith* layer in column k (as usually done: $W_i^{(k)} h_{i-1}^{(k)}$) and a weighted sum (weighted by U) across the previous layer activation's of all other previous columns indexed by j. In this way, the knowledge of all previous j columns can be used, modified, or totally ignored via learning different values for U (adapted from my previous coursework Alvarez-Charris (2018b)).

$$h_i^{(k)} = f\left(W_i^{(k)} h_{i-1}^{(k)} + \sum_{j<k} U^{(kj)} h_{i-1}^{(j)}\right) \tag{2.8}$$

2.6 Tensor Factorization and Tensor Norms

A different approach for achieving MTL is that of Tensor Factorization, which enables a system to *automatically* discover which representations to share across NN structures - avoiding the problem of manually hard-coding the features that must be shared across tasks. The idea exposed by Yang and Hospedales (2017) is to use the principles of matrix-based knowledge sharing but applied to tensors. In matrix knowledge sharing, we have a collection of models W composed of T linear models each one with D dimensions. Using matrix decomposition we can split W into $W = L.S$, where L is the shared knowledge across tasks and S are the task-specific knowledge. Using Tensor factorization the authors extended this idea to the tensors describing a Deep Neural Network (DNN) parameters. The authors approach is to train a DNN for each task, where the parameters for each layer of the DNN are trained using tensor decomposition. Under this approach when the whole system composed by all DNN is trained, it can automatically learn shared knowledge structures at the level of every layer.

An alternative approach to Tensor Factorization is to include a Tensor norm as a "task regularizer" within the loss functions (Yang and Hospedales, 2016). This regularizer enforces the weight updates to be done in a direction such that there exist a coupling between the different learning problems (task).mIn this paper the authors create a NN with layers $l_1^{(m)}, l_2^{(m)}, ..., l_N^{(m)}$ for each of the M tasks. Then, they create layer collections by grouping the layers across all NN i.e. a collection at the *ith* layer level is composed by $\{l_i^{(1)}, l_i^{(2)}, ..., l_i^{(M)}\}$. By applying a trace tensor norm on each collection, the training procedure makes each NN parameters to exhibit a shared knowledge structure with all other NN (all other tasks).

Chapter 3

Bipedal Robot: Learning to Walk under varying forces

Legged robots have become largely popular in the field of robotics because of their capability to traverse rough terrains, jump over obstacles and their use in studying animal locomotion. A class of legged robots called Bipedal robots have a large potential for engineering and commercial applications because of their anthropomorphic appearance - which is socially more appealing than other morphology's. Bipedal robots can be used as replacement for humans in task carried in hazardous environments and they can serve as lower limb prostheses and rehabilitation devices (Westervelt et al., 2007). A main challenge with bipedal robots is that of creating stability and motion control methods, for which many "classic" optimal control strategies already exist.

Although we acknowledge our project's scope is bounded to experiments within a simulation and that there is a huge simulation-to-reality gap in the context of robotic learning - we are motivated to use the Bipedal agent scenarios because of the potential of using reinforcement learning for learning good stability and motion control; given that many optimal control and path planning methods become brittle and unfeasible as the dimensionality of the system increases and the complexity of the environment changes (Ichter and Pavone, 2018). Moreover, bipedal motion control provides a simple way of testing the performance of a single agent under a range of domains.

In this chapter we describe the details of the Bipedal Walker simulation environment, along with the neural network architectures used as a function approximator and the optimization algorithm used for policy search. We do this for both a Single Domain

Learner (SDL) and a Multi Domain Learner (MDL), where each domain consist of learning how to walk under different magnitudes of a horizontal force ("wind"). At the end of this chapter we present the different single and multi domain experiments we performed.

3.1 Bipedal Environment

The Bipedal Walker environment [1] consist of a 2D bipedal robot with a main body (hull) and two legs, each one with a hip and knee joint **fig.** 3.1. The robot is equipped with a LIDAR sensor measuring the distance between the agent and the terrain.The scenario is situated in a irregular terrain full of grass where the robots tasks is to walk forward across a irregular terrain for as many times steps as possible or until the terrain ends. Each time a episode is instantiated , terrain irregularities and slopes are randomly changed.

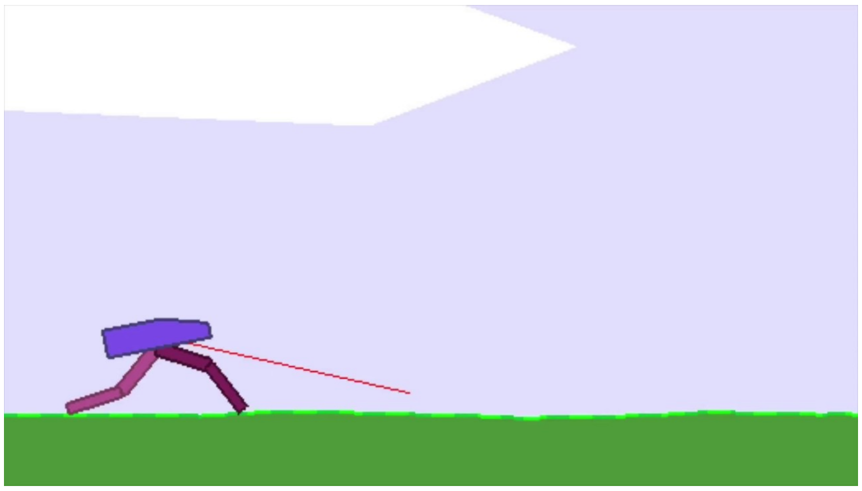

Figure 3.1: *Bipedal Walker Environment - Agent in purple and terrain in green*

Observation Space

Each state in the Bipedal Walker environment consist of a 24-dimensional sensory feedback. Such feedback is composed by a 4-D vector describing the angle, angular velocity, x velocity and y velocity of the hull, a 8-D vector measuring the angle and speed of the hip and knee joints for each of the legs, a 2-D binary haptic feedback indicating if the robots feet are in contact with the terrain, and a 10-D vector of measurements made

[1] `https://gym.openai.com/envs/BipedalWalker-v2/`

by the LIDAR sensor. The complete state space is summarized in **table.** 3.1. Note that the coordinates of the robot along the terrain are not part of the observation vector.

Table 3.1: *Bipedal Walker Observation and Action Space*

Observation	Min. Value	Max. Value
Hull Angle	0	2π
Hull Angular Vel.	$-\infty$	$+\infty$
Hull Vel X	-1	+1
Hull Vel Y	-1	+1
Hip 1 Angle	$-\infty$	$+\infty$
Hip 1 Speed	$-\infty$	$+\infty$
Knee 1 Angle	$-\infty$	$+\infty$
Knee 1 Speed	$-\infty$	$+\infty$
Leg 1 Contact	0	1
Hip 2 Angle	$-\infty$	$+\infty$
Hip 2 Speed	$-\infty$	$+\infty$
Knee 2 Angle	$-\infty$	$+\infty$
Knee 2 Speed	$-\infty$	$+\infty$
Leg 2 Contact	0	1
10 LIDAR readings	$-\infty$	$+\infty$
Action	**Min. Value**	**Max. Value**
Hip 1 Torque	-1	+1
Knee 1 Torque	-1	+1
Hip 2 Torque	-1	+1
Knee 2 Torque	-1	+1

Action Space

The robot has 4 degrees of freedom, two on each leg. The agent can decided over four continuous variable controlling the torque applied to the hip and knee joints in each leg. The direction of movement of each joint is based on the sign of the variable (negative $\rightarrow$ counter clock wise, positive $\rightarrow$ clock wise) and the torque applied to each joint is proportional to the magnitude of each variable. See **table.** 3.1 for a summary of the action space.

Reward Function

The reward function is based on the position of the robot along the X-axis X_{pos} (not part of the observation) and on the angle of the robots hull θ_{hull} (first entry of observation vector). Moving forward is a way of receiving high rewards and the reward functions encourages the agent to keep its head (hull) straight; any non-zero angle of the hull is penalized. Moreover, applying motor torque to each joint (we denote the torques' magnitude for the complete set of joints as $||\boldsymbol{A}||$) has a small penalty, encouraging optimal agents to be more efficient by minimizing the magnitudes of the torques. If the agents hull touches the floor or the agents falls of the cliff on the left end of the terrain in receives a penalty of -100. If the agent successfully gets to the right end of the terrain it does not receive any penalty.

$$r_t \propto +130.X_{pos} - 5 \mid \theta_{hull} \mid -0.00035.||\boldsymbol{A}|| \tag{3.1}$$

In our Multi Domain experiments each domain consist of changing the magnitude of a constant negative horizontal force that is analogous to ”wind” pushing the agent in the opposite direction to the direction of motion. The different ”wind” conditions take values of 0, 1 and 2.

3.2 Methods

From **section 2.1** , we can conclude that one of the main objectives of Reinforcement Learning (RL) is to perform a mapping between states and actions in order to maximize the expected sum of future discounted rewards. In environments where the number of states and actions is relatively small and known (approximate) dynamic programming methods (Sutton et al., 1998) can be used in order to find the optimal policy mapping states to actions [2]. This methods become unsuitable as the state and actions space increases, which are notoriously large spaces in continuous control scenarios.

A solution is to avoid using a tabular representation of the value and policy functions and rather to approximate such functions through a *function approximator* - a model with parameters θ mapping states to values ($V(s;\theta)$) or directly mapping states to actions ($\pi(a|s;\theta)$). Such techniques allows the values and/or policies to be summarized in low dimensional representations which drastically reduced the number of parameters that

[2]such as any class of method following the Generalized Policy Iteration criterion for improving the value estimations of states and eventually converging to a optimal policy

need to be learnt (Geramifard et al., 2013). This efficient solution presents to main issues: which model to use to parametrize the functions of interest, and how to learn the parameters θ of this model. The following sections describe the rationale behind the type of function approximator we used and the algorithms used to optimized its parameters. This is done for Single Domain (SDL) and Multi Domain (MDL) agents.

3.2.1 SDL/STL Function Approximator Architecture

There exist a wide variety of methods that can serve as ideal candidates for parametrizing the policy and state-value function [3], for example linear regression models, Nearest Neighbors algorithms, Gaussian Procceses, decision trees (Pyeatt et al., 2001) and many other statistical and machine learning models. However, to facilitate learning the parameters of the models we only consider differentiable function approximators (FA) to which gradient based optimization techniques can be applied.

One of the simplest differentiable FA models is a linear function approximator as that described by **(3.2)**, in which a D-dimensional feature vector ϕ (that transforms raw states into relevant features) is linearly mapped to the value of the state s or the action to perform is such state through a set of weights contained in $\boldsymbol{\theta}$. Although performing stochastic gradient descent (SGD) to learn the weights of this model has a simple and highly tractable form, in a more complex environment such as that of the Bipedal Walker using this simple model does not produce good results. In fact, preliminary experiments that we carried on showed that a binary policy made out of a simple one parameter linear regression (what some would call a ”Perceptron”) easily solved simple environments such as Cart Pole. However, this good results did not extend neither to the Bipedal Walker scenario nor to other much simpler ”classic” scenarios such as continuous control Pendulum and Mountain Car [4].

The poor results in complex scenarios are mainly caused by a key limitation of linear FA - they do not take into account the interactions between different features (Sutton et al., 1998) and thus are prone to fail in environments with complex dynamics i.e. if the hip angle is large, a high hip velocity is not ideal since it could indicate that the agent is about to fall. Even though we could had included such interactions by

[3]We are only interested in functions performing *direct* mapping form states to actions or values (policy or value function) as this is the approach we selected. We acknowledge there are other methods that instead approximate $Q(s,a;\theta)$ but this were not used in this project

[4]Classifc Environment: https://gym.openai.com/envs/MountainCarContinuous-v0/

engineering higher order features through polynomial or radial basis functions, hard coding manually picked features is a difficult task and in the Bipedal walker scenarios it is not obvious which features would be useful for learning how to walk.

$$V(S;\theta) = \boldsymbol{\phi(S)}.\boldsymbol{\theta} = \sum_{i}^{D} \phi_i(S).\theta_i \tag{3.2}$$

Having explored the the linear FA, the natural next step to overcome its limitations was to use a model capable of automatically discovering useful features - such as Artificial Neural Networks (ANN). This types of models can be understood as hierarchical linear regression where we apply the same principle exposed in (**3.2**) with two notables differences: *1)* The process of having a linear combination of the features $\boldsymbol{\phi(S)}$ weighted by $\boldsymbol{\theta}$ is sequentially repeated several times, *2)* After each repetition the result of the product $\boldsymbol{\phi(S)}.\boldsymbol{\theta}$ is passed through a non linearity [5]. The sequential repetitions of the linear combination is what are called layers of the network, and the non linearity's are called activation functions. By combining these two elements we can have L sequential layers in which the output of layer $l-1$ is passed through a non linearity and serves as the input of layer l. This process allows to compute a hierarchy of useful features with increasing levels of complexity - going from the raw inputs (denoted by $\boldsymbol{S}$) to higher level abstractions useful to approximate the function of interest (Bengio et al., 2009).

Equations (**3.3**) and **fig.3.2** describe a 2 layered ANN: The first layer (hidden layer) consist of K- different units mapping the raw state $\boldsymbol{S}$ into various features using a set of weights $\boldsymbol{B}$ and sigmoidal non linearity's $\sigma(.)$. The output tensor of the first layer denoted by $\boldsymbol{\phi(S)}$ is the input to the second layer (output layer) which then maps such features into the final value of the state $V(S)$ using weights $\boldsymbol{W}$ and another sigmoid function. We would jointly optimize all parameters contained in the vector $\boldsymbol{\theta}$ using techniques such as SGD. [6]

$$\phi_k = \sigma(\boldsymbol{B}^{(k)}.\boldsymbol{S}) \qquad \boldsymbol{\phi(S)} = [\phi_1, \phi_2, \ldots, \phi_K] \tag{3.3}$$

$$V(S;\theta) = \sigma(\boldsymbol{W}.\boldsymbol{\phi(S)}) \qquad \boldsymbol{\theta} = \{\boldsymbol{B}, \boldsymbol{W}\} \tag{3.4}$$

[5] I enjoy this less "shiny" and clearer perspective of ANN. More details found in Iain's Murray MLPR class: `http://www.inf.ed.ac.uk/teaching/courses/mlpr/2017/notes/w4b_neural_net_intro.html`

[6] For simplicity we did not include bias terms in this model. In practice this should be included for approximating functions with a non zero y-intercept

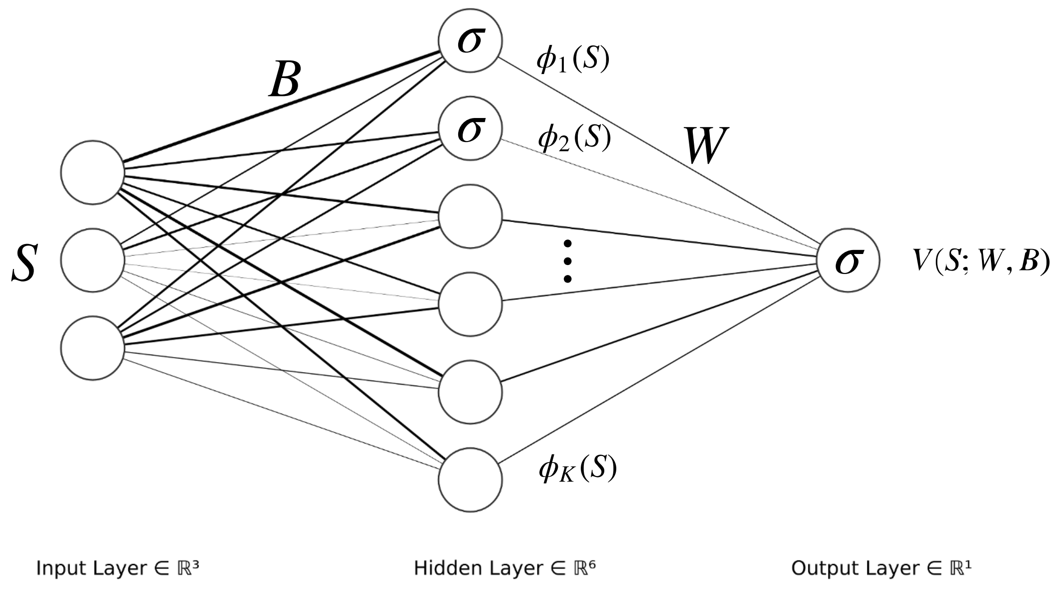

Figure 3.2: *Artificial Neural Network diagram describing how the model performs regression (function approximation) by estimating the relationship between input raw states S and output states value $V(S)$ through parameters $\boldsymbol{\theta} = \{\boldsymbol{B}, \boldsymbol{W}\}$*

Once we have established a ANN model as the FA of our choice we inherently solve the limitations of linear models and can then perform End-to-End Differentiable Model Fitting - in which by using optimization techniques (discussed in **section 3.2.3**) we can automatically find the weights that transform raw inputs into a more useful hierarchy of features.

As illustrating the mapping from states to scalar values is easier, the equations and figures discussed until now have exemplified how we can approximate the value function. However, the reader should keep in mind that our ultimate goal is to find a good policy π_θ (recall sub index and ";" denote "parametrized by") maximizing the agents rewards. To achieve this we used our FA to *directly* map states into actions, but instead of having as output a scalar estimation of $V(S)$ we have a vector parametrizing the mean of multiple Gaussian distributions. We require such distributions because, unlike the scenario of discrete control were the policy is computed by applying a softmax over the networks output layer and picking the action with the highest probability, our environments with continuous actions space require that the network parametrizes continues distributions. For the Bipedal Walker scenario in which there are 4 possible actions the output of the network are 4 scalars parametrizing the means of 4 different Gaussian distributions. With this in place the agent has a *stochastic policy* that is used to act in the environment by sampling from each probability density.

Following the strategy used in (Schulman et al., 2017), instead of outputting 8 values from the network (4 means and 4 variances), a independent 4 dimensional trainable variable was used to parametrize the 4 variances. Given that variances are by definition positive, to avoid the use of constraint optimization algorithms the trainable variable predicted the logarithms of the variance - hereafter referred as log-variance trainable variance. Finally, to prevent the initial log-variance of the distributions to be zero - which would create the undesired scenario of having point probability densities and reduce action exploration - we always added a intercept value of 1.0 as suggested in (Schulman et al., 2017) to the trainable log-variance variable.

Equations (**3.5**) illustrate the aforementioned description of the Policy Network. For clearer notation instead of writing separate probability densities for each individual action a, the densities are represented as a Multivariate Normal Distribution from which all actions $\boldsymbol{a}$ are sampled. The log-variance variables solely represents the diagonal entries of the covariance matrix $\boldsymbol{\Sigma}_\theta(\boldsymbol{S})$.

$$\boldsymbol{a} \sim \pi_\theta(a \mid \boldsymbol{S}) = \mathcal{N}\left(a \,\middle|\, \boldsymbol{\mu}_\theta(\boldsymbol{S}), \mathrm{e}^{\boldsymbol{\Sigma}_\theta(\boldsymbol{S})}\right) \tag{3.5}$$

$$\boldsymbol{\mu}_\theta(\boldsymbol{S}) \leftarrow \text{ANN} \qquad \boldsymbol{\Sigma}_\theta(\boldsymbol{S}) \leftarrow \text{log-variance variable} \tag{3.6}$$

With the above described Policy Network our agent could (in theory) learn to walk by solely acting according to $\pi_\theta(a|s)$, collecting trajectories of (s, a, r, s'), and optimizing for θ. However as it is described in **section 3.2.3** , optimizing only based on the rewards r is in most scenarios unfeasible. The collection of r despite providing a unbiased estimate of the agents performance (it is unbiased since it is purely based on the agents real experience while interacting with the environment) are extremely noisy estimates with high variance.

To solve this problem there are several estimators with different bias / variance trade offs, such as Temporal Difference (TD) residuals, Q-functions, and Advantage functions. Following the competitive results exposed in (Schulman et al., 2016) we decided to use the Generalized Advantage Estimator (GAE) who's bias / variance trade off is favorable and can be easily controlled by adjusting a single hyperparameter. To compute GAE we required a model estimating the value function $V(S)$. We could had used any FA to do so, but given the aforementioned advantages of using ANN as FA's we decided to estimate $V(S)$ using an additional network parametized by ϕ (this is called the Value Network $V_\phi(S)$).

It is important to notice that unlike some "actor-critic" methods we have two totally disjoint Neural Networks, their parameters are independently optimized and the *only* influence that V_ϕ has over π_θ is in shaping the gradients for updating θ. We decided to use the same architecture for V_ϕ and π_θ with the difference that the output layer of the Value network is a single scalar while the Policy network is a multidimensional vector $\boldsymbol{\mu_\theta(S)}$ (4-dimensional for the Bipedal Walker environment).

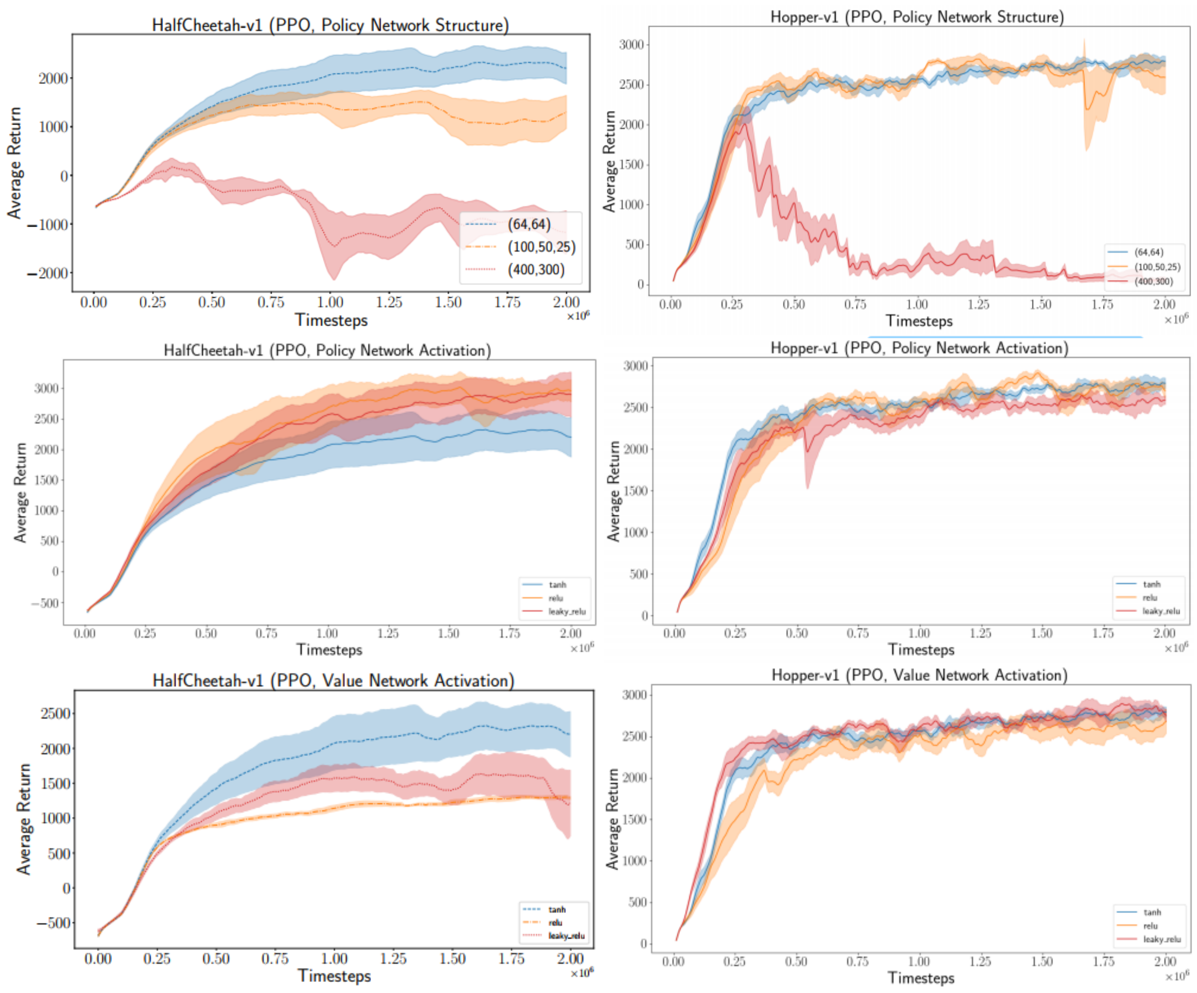

Figure 3.3: *Average return across 5 trials with 2M training samples for different FA architectures. Top: Policy Network* structure *permutations with best score obtained by the (64,64) structure. Middle and Bottom: Policy and Value Network* activation's functions *permutations with best scores respectively achieved by ReLU and tanh activations (figure taken from Henderson et al. (2018))*

Having the Policy and Value Networks FA the final step to fully describe these models is to specify their architectures - which for Deep Reinforcement Learning (DRL) and in general neural network based machine learning is more of an art than a rigorous science. Do to time and computing constraints instead of doing a extensive hyperparameter search we decided to follow the guidelines described in the literature. We chose the networks structure (units per hidden layer, not total depth since different depths were tested for our experiments) and activation functions based on Table.1 from Henderson et al. (2018) in which Policy and Value Networks hyperparemeter permutations where tested in challenging environments (Half Cheetah and Hopper). As seen in the table and the supplementary figures (which we present here in **fig.3.3**), for the policy search algorithm of our choice (Proximal Policy Optimization introduced in **section 3.2.3**) the best average score was obtained by: a π_θ with a structure with two hidden layers with 64 units and Rectified Linear Units (ReLU) as activation functions, and a V_ϕ with 64 units in two hidden layers and a hyperbolic tangent (tanh) activation functions. After all the hidden layers there is a output layer consisting of a dense linear layer.

As the weights initialization plays an important role in the stability of the network while training [7], all parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ are randomly initialized drawn from Gaussian $\mathcal{N}(\mu = 0, \sigma^2 = \frac{1}{n_{in}})$. This initialization attempts to reduce the variance of the weight updates by considering the number of incoming connections n_{in} to each layer (this initialization has produced good results in previous worked I have carried out and was introduced in Klambauer et al. (2017))

3.2.2 MDL/MTL Function Approximator Architecture

With the Artificial Neural Network architecture described in the previous section an agent could learn a policy by sampling episodes from a environment with a single domain/task. However, when such Vanilla architecture is trained with episodes sampled from multiple domains, the function approximator usually encounters the problem of of catastrophic interference, in which previously learnt knowledge is overwritten and forgotten when new tasks are learnt. As studied in (McCloskey and Cohen, 1989) *"New learning may interfere catastrophically with old learning when networks are trained sequentially"*. Although our networks will be trained with episodes from different

[7] For short interesting explanation on the effects of initialization in controlling the gradients variance see: `http://andyljones.tumblr.com/post/110998971763/an-explanation-of-xavier-initialization`

domains fed in parallel (and not sequentially), having to learn based on experience sampled from different domains negatively interferes in the parameters optimization and in some cases makes learning unfeasible.

In order to learn in the multi domain and multi tasks learning problems that we want to tackle, we decided to use the Hard Parameter Sharing approach: were a block of layers is shared across all domains and certain layers specialize for domain specific feature extraction. We denote the block of common layers as the *Core Layers* and the block of specialized layers as *Head Layers*. Because of this, we call this architecture a *Multi Headed Network*, which is shown if **fig.** 3.4.

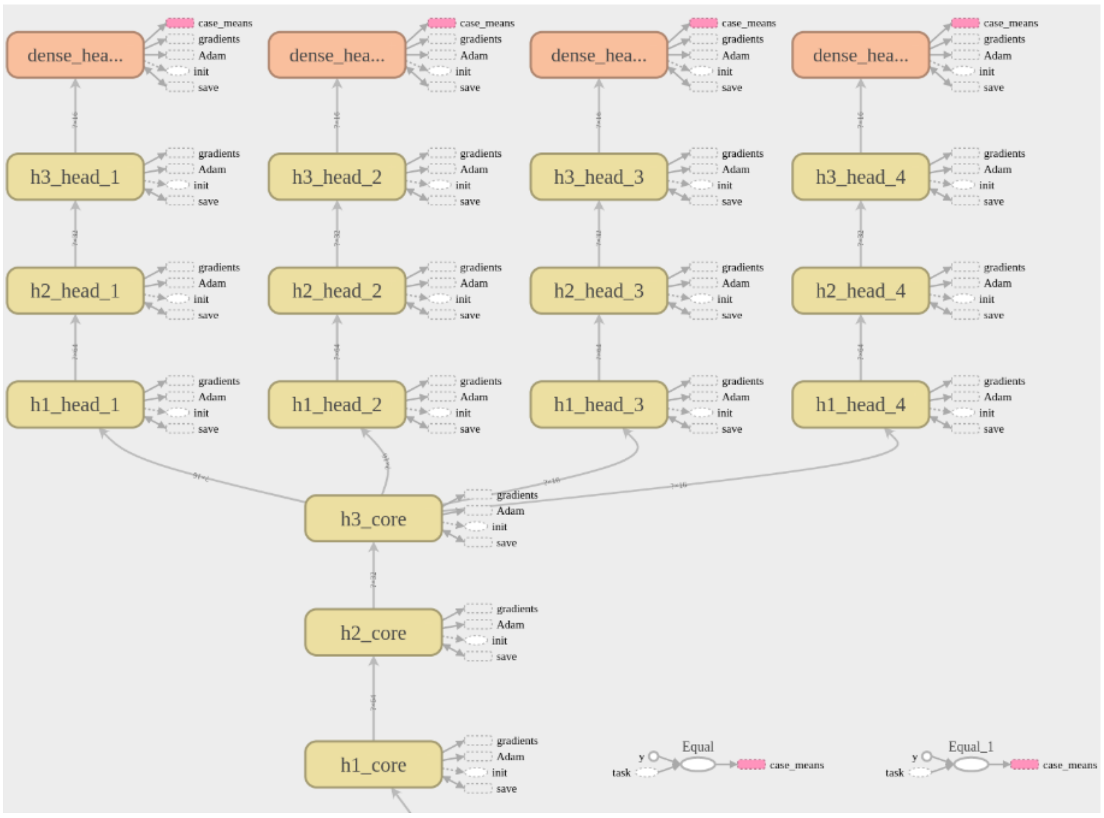

Figure 3.4: *Multi Headed Architecture proposed for learning multiple domains and tasks. The Network is learning from 4 domains/tasks, it has 3 Core hidden layers shared across all domains/task, and each head has 3 hidden layers specializing in each of the domain/tasks. Each head has a final output layer made by a fully connected linear layer (dense). Graph obtained using TensorBoard*

In this architecture, the *ith* head parametrizes the mean $\mu_{\theta_c,\theta_{h_i}}$ of a Gaussian distribution for each action dimension. Notice that the mean depends on two sets of parameters, parameters θ_c of the common core layers and parameters θ_{h_i} of the specialized head . This highlights a detail of the way this architectures needs to be trained; when back-propagating gradients which are computed from episodes sampled from task i, we only want this gradients to me applied to θ_c and θ_{h_i}, and not to any $\theta_{h_{j \neq i}}$. To achieve this we included within the computational graph of the architecture a switch-case operation, which receives as input a scalar indicating from which domain/task the training episodes are being sampled from, and in this way it controls to which parts of the graphs the gradients are applied to. The flow of gradients to each of the heads can be seen in **fig. 3.5**

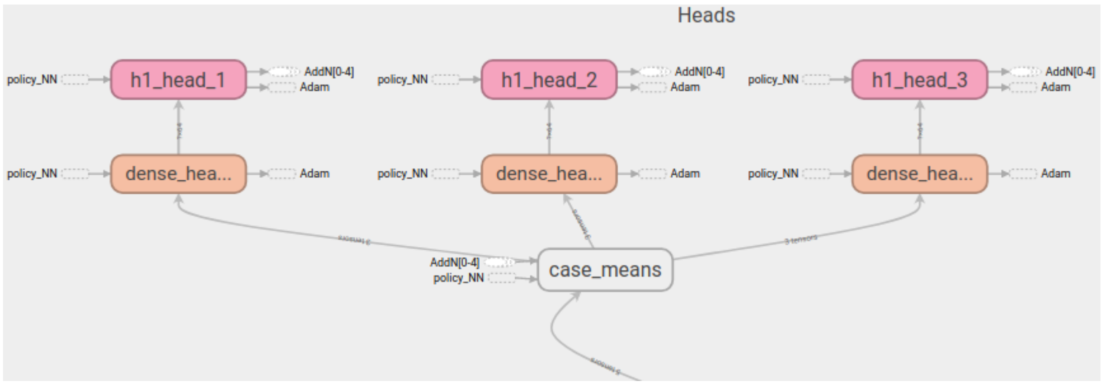

Figure 3.5: *Switch-case operation controlling the direction in which tensors flow when back-propagating the gradients through the Multi Headed Architecture (**note:** unlike the previous graph, this network is solving 3 tasks with a single hidden layer per head). Having this operation inside the computation graph ensures that the gradients only flow through the appropriate head depending on which domain/task is being used to sample the training episodes from. Graph obtained using TensorBoard*

This architecture also has the log-variance trainable variable, one for each action in each domain/task. For example, to solve 4 different domains in the Bipedal Walker scenario which has 4 control actions, the Multi Headed Architecture would in total parametrize 16 means and 16 log variances. Recall that we need a Value Function approximator in order to compute the Generalize Advantage Estimator. Although it is a common approach to use a Vanilla Value Network for each domain/task being solve (i.e. if there are 4 domains 4 independent networks are used), we decided to use a single Value Network with the same principle of the Multi Headed architecture. Each head outputs a single scalar predicting the value of the state sampled from the associated domain/task.

The Multi Headed Architecture was implemented using the same activation's functions as those used for the SDL/STL lagents (ReLU for the Policy Networks and Tanh for the value network) and each layer had by default 64 units. All parameters are initialized based on the Gaussian distribution with a variance set to be the inverse of the number of incoming connections to each layer. ***Note:*** When describing the depth of a Multi Headed Network, a architecture with N - hidden layers has int($\frac{N}{2}$)+1 core hidden layers and each head has int($\frac{N}{2}$) hidden layers i.e. a 7 hidden layer MTL agent has 4 core hidden layers and each head has 3 layers plus a dense output layer; always more layers on the Core block.

3.2.3 SDL/STL Policy Search Algorithm

In the previous sections we described the function approximator (FA) with parameters $\boldsymbol{\theta}$ that will be used to parametrize the single and multi task learning agents' policy π_θ. With this in place a important remaining question is how to find good values of $\boldsymbol{\theta}$ that will maximize the agents utility. As the chosen function approximator (and each of its layers) is fully [8] differentiable we can used gradient based optimization methods (see **section 2.3**) to search for a good policy.

Methods that use a function approximator (FA) directly mapping states to actions and that update the parameters of the FA using its gradients with respect to some objective functions are called *Policy Gradient* (PG) methods. Gradient updates can be applied to our model, a Neural Network, using the backpropagation algorithm (BP) [9]. In BP we have a forward pass in which the signals going from the input to the output layer are propagated by computing the activation of each unit , and in a backward pass the gradient for each weight is efficiently computed by propagating the partial derivatives using the chain rule of differentiation.

As mentioned above we *need* a objective function $J(\theta)$ in order to apply gradient based methods. Unlike supervised learning methods were there exist predicted and target labels, in RL there is is no clear target and thus no unique definition of $J(\theta)$. A common and sensible choice for the objective function is the (discounted) sum of rewards obtained in a trajectory τ of length T (denoted by $R(\tau)$), and maximizing such objective (**3.8**). Its sensible since this is the overall objective of the RL problem: "maximize the expected sum of discounted rewards [the return]". Notice that a trajectory

[8]ReLU's are differentiable every where except at x=0, a issue that can be easily managed using sub-gradient methods already included in most automatic differentiation libraries

[9]Many books attempt to explain BP using δ and other complicated approaches. I find it clearer to view BP as inverting a computational graph and applying the chain rule of differentiation: www.inf.ed.ac.uk/teaching/courses/mlpr/2017/notes/w5a_backprop.html

τ is fully dependant on the environments dynamics $p(s_{t+1} \mid s_t, a_t)$ and what actions a_t we choose by sampling from π_θ. As we are interested in maximizing the *expected* return such expectation is computed under the distribution of trajectories $p_\theta(\tau)$ given by (3.7) [10].

$$p_\theta(\tau) = p(s_0) \prod_{t=0}^{T-1} p(s_{t+1} \mid s_t, a_t)\pi_\theta(a_t \mid s_t) \tag{3.7}$$

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}\left[\sum_{t=0}^{T-1} \gamma^t r_t\right] = \mathbb{E}_{\tau \sim p_\theta(\tau)}\left[R(\tau)\right] \tag{3.8}$$

$$\theta_{k+1} \leftarrow \theta_k + \alpha \nabla_\theta J(\theta) \tag{3.9}$$

To compute the expression for the gradient, we can expand the expectation and use the log-derivative trick ($\nabla_x \log f(x) = \frac{1}{f(x)} \nabla_x f(x)$)

$$\nabla_\theta \log p_\theta(\tau) = \frac{1}{p_\theta(\tau)}.\nabla_\theta p_\theta(\tau) \longrightarrow \nabla_\theta p_\theta(\tau) = p_\theta(\tau)\nabla_\theta \log p_\theta(\tau) \tag{3.10}$$

$$\nabla_\theta J(\theta) = \nabla_\theta \int p_\theta(\tau) R(\tau)\, d\tau \tag{3.11}$$

$$= \int \nabla_\theta \left[p_\theta(\tau) R(\tau)\right] d\tau \tag{3.12}$$

$$= \int R(\tau) \nabla_\theta p_\theta(\tau)\, d\tau \tag{3.13}$$

$$= \int R(\tau)\, p_\theta(\tau) \nabla_\theta \log p_\theta(\tau)\, d\tau \tag{3.14}$$

$$= \mathbb{E}_{\tau \sim p_\theta(\tau)}\left[R(\tau) \nabla_\theta \log p_\theta(\tau)\right] \tag{3.15}$$

From (3.7) we can see that the environments dynamics (both initial state and transition dynamics) are independent of the parameters θ. Thus, the gradient of the log distribution of trajectories solely depends on our policy:

[10]The probability of a trajectory is given by how likely it is to start at initial state s_0, and from there on acting according to π_θ and transitioning in the environment according to its dynamics. Simple understanding of the chain rule of probability theory as seen in Michael's Guttman PMR course: https://www.inf.ed.ac.uk/teaching/courses/pmr/17-18/assets/slides/slides03.pdf

$$\nabla_\theta \log p_\theta(\tau) = \nabla_\theta \left[\log p(s_0) + \sum_{t=0}^{T-1} \log p(s_{t+1} \mid s_t, a_t) + \log \pi_\theta(a_t \mid s_t) \right] \tag{3.16}$$

$$= \nabla_\theta \sum_{t=0}^{T-1} \log \pi_\theta(a_t \mid s_t) \tag{3.17}$$

By inserting (**3.17**) into the expression derived in (**3.15**) we obtain the final equation for the gradient of the objective function (**3.18**). This same equation but for a *single time step* within the complete trajectory (**3.19**) gives us a better view of what PG is doing. Intuitively, what is happening is the following: with $\nabla_\theta \log \pi_\theta(a_t \mid s_t)$ we are telling the network to shift θ in the direction that increases the probability [11] of action a_t being selected - ”regardless” if that action was or not good. Then, using R_t we include the real utility that a_t had in the environment and by multiplying it with the original gradient we had we enforce moving in that direction (if R_t is positive, favoring good actions) or in the opposite direction (if R_t is negative, discouraging bad actions). The gradient is averaged across all the actions in τ and a final pdate is performed. See **fig.3.6** for illustration showing the update process.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[R(\tau) \nabla_\theta \sum_{t=0}^{T-1} \log \pi_\theta(a_t \mid s_t) \right] \tag{3.18}$$

$$\nabla_\theta J_t(\theta) = \mathbb{E}_t \left[R_t \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right] \tag{3.19}$$

With the above we have all the elements to perform gradient ascent on $J(\theta)$ and iteratively obtain better θ to maximize the expected return. However, because of the complexity of the environments we are using in this project, small changes in θ might lead to drastically different trajectories. Given that R_t is based on such real trajectories (which makes it a unbiased estimator), R_t is prone to having high variance, producing unstable gradients, and making learning unfeasable. As described in Schulman et al. (2016) the variance [12] of R_t *”scales unfavorably with the time horizon, since the effect of an action is confounded with the effects of past and future actions”*. This is true since given the stochasticity of the environment, each r_t can be viewed as a random variable -

[11] As π_θ is a distribution, moving θ in this direction will shift the mean of the distribution towards favoring that action

[12] Excellent explanations in Schulman et al. (2016) and `https://danieltakeshi.github.io/2017/03/28/going-deeper-into-reinforcement-learning-fundamentals-of-policy-gradients/`

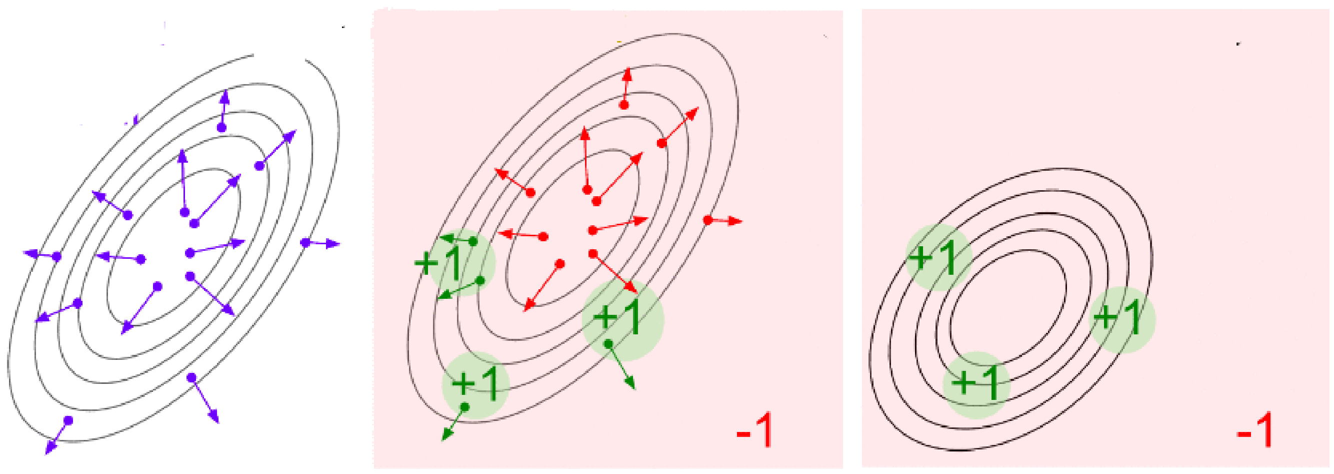

Figure 3.6: *Visualization of a Policy Gradient update. Left: Counter plot of π_θ for initial parameters θ_t along with the selected actions a_t (in purple) and a arrow indicating the direction of the gradient $\nabla_\theta \log \pi_\theta(a_t|s_t)$. The arrow indicates the direction in which the mean of the distribution should be shifted to increase the probability of each action. Middle: Overlayed values of R_t showing highly (+1) and poorly (-1) rewarding actions. The arrows are now color coded because of the multiplication between the cumulative reward and the gradient. Right: By performing the sum of gradients across the complete trajectory (adding green and red arrows), the policy is updated with new parameters θ_{t+1}. Samples from then new distribution will now have a higher expected reward just as was desired (Image and caption partially taken from `http://karpathy.github.io/2016/05/31/rl/`)*

which when summed to obtain R_t produces a high variance estimate of the return (taken from Deep Mimic explanation Peng et al. (2018))

Because of this we decided to replace R_t by a estimator with a much more favorable bias / variance trade-off called Generalized Advantage Estimator (GAE, Schulman et al. (2016)). Given that the agent is at state s_t, a estimate of the Advantage Function $\hat{A}(s_t, a_t)$ measures how much value can be gained by taking action a_t (value given by $Q(s_t, a_t)$[13]) when compared to the average value obtained from such state (value given by $V(s_t)$) i.e. is taking a_t better or worst than average.

$$\hat{A}(s_t, a_t) = Q(s_t, a_t) - V(s_t) = [r_t + \gamma.V(s_{t+1})] - V(s_t) \tag{3.20}$$

As in reality we do not only have access to a one step look ahead of real reward but

[13]Based on equation (2.3) introduced in the background section we can relate Q-values to $V(s)$ by $Q(s_t, a_t) = r_t + \gamma.V(st+1)$

rather to a complete trajectory of length T , the term encompassed in brackets can be estimated by looking a different time steps along our trajectory. This is called a *n-step return* described by (**3.21**); in which with the value of n we can select the bias-variance trade off that we want. For $n = T$ we obtained the unbiased estimate $R_t^{(T)} = R_t$ which uses all the rewards of real experience and thus has high variance, and for $n = 1$ we obtained a biased but low variance estimate $R_t^{(1)} = r_t + \gamma V(s_{t+1})$ (recovering the term term encompassed in brackets of the above equation). The trick that GAE uses to estimate the advantages and to avoid choosing a specific n-step return, is to actually use the return for all possible values of n. This can be achieved by using λ*-returns* obtained by exponentially weighting all n-step returns. As before, $\lambda = 1$ produces the unbiased / high variance R_t and $\lambda = 0$ the low variance single step return $R_t^{(1)}$.

$$R_t^{(n)} = \sum_{i=0}^{n-1} \gamma^i r_{t+i} + \gamma^n V(s_{t+n}) \tag{3.21}$$

$$R_t(\lambda) = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)} \tag{3.22}$$

By replacing the λ*-returns* in the original equation we obtained a family advantage estimators parametrized by γ and λ; were by varying lambda we can easily control the bias-variance trade off that we desire [14](which is precisely the motivation for not using the vanilla R_t). Remember that we do not have access to the actual Value function, but that we have a model parametrized by ϕ which estimates such values.

$$\hat{A}_t^{GAE(\gamma,\lambda)} = R_t(\lambda) - V_\phi(s_t) \tag{3.23}$$

By introducing GAE as the gradient multiplier, we have made the gradients of the objective function much more stable. However, what we have described so far is a purely *On-policy* optimization algorithm. We use policy π_{θ_t} to gather a set of trajectories τ_t, and then using this trajectories we update the policy to obtain $\pi_{\theta_{t+1}}$. Since after the update the policy has changed, it would be incorrect to continue using old trajectories τ_t to perform a second optimization step on $\pi_{\theta_{t+1}}$; we rather need to obtain a new set of trajectories τ_{t+1} to perform any subsequent update.

[14]We will have that Var($GAE(\gamma, 1)$) > Var($GAE(\gamma, 0)$. For an excelent explanation of GAE the reader may refer to the appendix of Peng et al. (2018) or `https://danieltakeshi.github.io/2017/04/02/notes-on-the-generalized-advantage-estimation-paper/`

This is a inefficient process, since ideally we would like to use one batch of data to perform multiple policy optimization steps. To do so we decided to include an importance sampling methods which modifies the gradient $\nabla_\theta J(\theta)$ obtained in (**3.19**) by introducing a ratio $w(\theta)$ between the policy used to gather the data $\pi_{\theta_{old}}$ and the policy that is being optimized π_θ . In this way we can perform several optimization steps over π_θ by recycling off-policy samples obtained with $\pi_{\theta_{old}}$:

$$w(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} \tag{3.24}$$

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} \hat{A}_t^{GAE(\gamma,\lambda)}(\tau) \nabla_\theta \sum_{t=0}^{T-1} \log \pi_\theta \right] \tag{3.25}$$

$$\nabla_\theta J(\theta) = \mathbb{E}_t \left[w(\theta) \hat{A}_t^{GAE(\gamma,\lambda)}(\tau) \nabla_\theta \log w(\theta) \right] \tag{3.26}$$

Although in theory the policy ratio (importance sampling) introduced to compute the gradient should allow several policy updates, there is a limit on how many optimization steps can be done with a given batch of old data. If π_θ diverges too much from $\pi_{\theta_{old}}$ the learning algorithms becomes heavily unstable. To stabilize the policy gradient algorithms and constraint how much both policies differ, we used the methods introduced in Trust Region Policy Optimization (TRPO, Schulman et al. (2015) and the Proximal Policy Optimization (PPO, Schulman et al. (2017).

TRPO limits the difference between both policies by maximizing the same objective function associated with [15] (**3.26**) but subject to ($s.t$) the constraint that the KL Divergence between the old and new distributions is less than a hyperparameters value δ_{KL} (which we denote as KL target value). In practice, instead of enforcing the constraint one can usually enforce a penalty over the objective function, either based on the KL divergence (penalty gain controlled by hyperparameter β) or a Hinge Loss which is enforced when the KL divergence surpasses δ_{KL} (gain controlled by η) . Note that to simplify notation here we drop the GAE term from the advantage estimator and explicitly denote that this advantages are from old experience batches.

$$J^{\text{TRPO}}(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[w(\theta) \hat{A}_{\theta_{old}}(\tau) \right] \ \ \textbf{s.t} \ \ \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[D_{KL}(\pi_{\theta_{old}}(.|s) || \pi_\theta(.|s) \leq \delta_{KL} \right] \tag{3.27}$$

$$J^{\text{TRPO}}(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[w(\theta) \hat{A}_{\theta_{old}}(\tau) - \beta D_{KL}(\pi_{\theta_{old}}(.|s) || \pi_\theta(.|s)) \right] \tag{3.28}$$

[15]Note that the equations we have derived so far are **gradients** of the objective function. After all the derivations, to recover the final objective function that needs to be maximized we simply integrate

Given that the KL Divergence associated with TRPO might be difficult to enforce, PPO simplifies this by limiting the value of the ratio $w(\theta)$ to the range $[1+\epsilon, 1-\epsilon]$.

$$w_{clip}(\theta) = \text{clip}\left(w(\theta), 1-\epsilon, 1+\epsilon\right) \tag{3.29}$$

$$J^{\text{CLIP}}(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}\left[\min\left(w(\theta)\hat{A}_{\theta_{old}}(\tau), w_{clip}(\theta)\hat{A}_{\theta_{old}}(\tau)\right)\right] \tag{3.30}$$

The reasoning behind this is the following: one can see the ratio as a measurement of similarity of the old and new policy. If the ratio is ”1” they are the same, but if the ratio is less or greater than ”1” the policy has decreased or increases the probability of the actions. Under this objective function: for good actions (positive advantage) changing the policy to have a ratio beyond $1+\epsilon$ produces no improvement of the objective function (as seen in the flat are of **fig 3.7**) and thus there is no incentive to continue changing the policy. For bad actions (negative advantage), the policy is only encourage to reduce the probability of such actions upto a ratio of $1-\epsilon$. Limiting the diveragnace betwen old and new policies with this simple clipping methods produces stable learning.

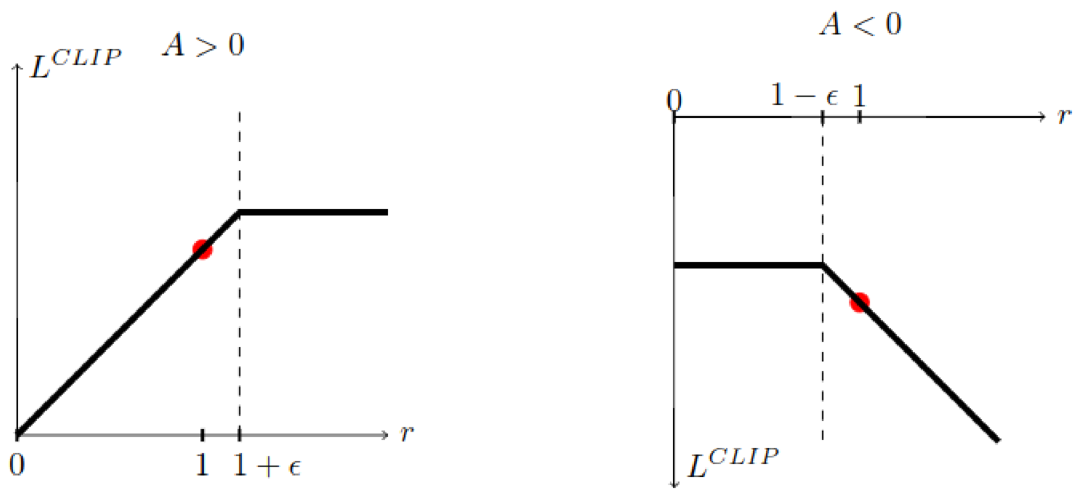

Figure 3.7: *Curves of objective Function when the old and new policy ratios are limited using PPO's method. Note that L^{clip} is analogous to what in our notation is J^{clip} (Figure taken from Schulman et al. (2017))*

All in all, the Policy Gradient algorithm that we implemented in this project uses all the improvements motivated in this section: GAE, importance sampling, TRPO and PPO

Summary of Policy Gradient Algorithm

The entire Policy and Value networks optimization algorithm functions in the following way. The updates we perform are not based on online learning but rather on batch based optimization. We first initialize the networks and sample a set of trajectories usually consisting of 20 complete episodes. Each trajectory is made out of: observations (states), actions, rewards, and advantages (computed based on the predicted values $V_\phi(s)$ and the sum of discounted reward). Once all trajectories have been obtained we optimize π_θ and V_ϕ.

For optimizing the Policy Networks the Proximal Policy Algorithms is applied by using the observations, actions, and advantages. Before starting the first optimization iteration, we compute what the initial means and log-variances predicted by the network: $\boldsymbol{\mu}_{\theta_{old}}$ and $\boldsymbol{\Sigma}_{\theta_{old}}$. Having this as reference values for $\pi_{\theta_{old}}$ we then run several iterations of backprogation. After each iteration, we check that the KL divergence between the distribution parametrized by $\boldsymbol{\mu}_{\theta_{old}}$, $\boldsymbol{\Sigma}_{\theta_{old}}$ and the distribution parametrized $\boldsymbol{\mu}_\theta$, $\boldsymbol{\Sigma}_\theta$ (the policy being updated) does not exceed 4 times the δ_{KL} target value. After all the iterations have gone by (or if $4\delta_{KL}$ is exceeded), the optimization process is stopped .Afterwards the learning rate and β (term controlling the importance of the KL Divergence in the loss function) are updated based on the final value of D_{KL}

For optimizing the Value Network we solve a regression problem by using the observations and the discounted sum of rewards. As the loss function to optimize we use the Mean Square Error between the target value $y(s)$ which is computed using TD(1) target (i.e the λ-return with $R_t(\lambda = 1)$) and the value predicted by the network $V_\phi(s)$. For training the Value Networks we use a replay buffer which stores the trajectories from the previous optimization call. In this way, the data used to train the networks at each optimization call the old trajectories plus the newly sampled trajectories.

A good summary of this training algorithm is shown in **fig. 3.8** which is taken from the chart published in Peng et al. (2018). Our training methods is really similar except that we: *1)* use *TD(1)* (instead of TD(λ)) as the target value y_i to be predicted by the Value Network *2)* In the chart the Value Networks is parametrized by ψ, which is analogous to what in our notation is ϕ parametrizing V_ϕ *3)* For training V_ϕ we use replay buffers storing old trajectories.

```
1:  $\theta \leftarrow$ random weights
2:  $\psi \leftarrow$ random weights
3:  while not done do
4:      $s_0 \leftarrow$ sample initial state from reference motion
5:      Initialize character to state $s_0$
6:      for step = $1, \ldots, m$ do
7:          $s \leftarrow$ start state
8:          $a \sim \pi_\theta(a|s)$
9:          Apply $a$ and simulate forward one step
10:         $s' \leftarrow$ end state
11:         $r \leftarrow$ reward
12:         record $(s, a, r, s')$ into memory $D$
13:     end for

14:     $\theta_{old} \leftarrow \theta$
15:     for each update step do
16:         Sample minibatch of $n$ samples $\{(s_i, a_i, r_i, s'_i)\}$ from $D$

17:         Update value function:
18:         for each $(s_i, a_i, r_i, s'_i)$ do
19:             $y_i \leftarrow$ compute target values using TD($\lambda$)
20:         end for
21:         $\psi \leftarrow \psi + \alpha_v \left( \frac{1}{n} \sum_i \nabla_\psi V_\psi(s_i)(y_i - V(s_i)) \right)$

22:         Update policy:
23:         for each $(s_i, a_i, r_i, s'_i)$ do
24:             $\mathcal{A}_i \leftarrow$ compute advantage using $V_\psi$ and GAE
25:             $w_i(\theta) \leftarrow \frac{\pi_\theta(a_i|s_i)}{\pi_{\theta_{old}}(a_i|s_i)}$
26:         end for
27:         $\theta \leftarrow$
            $\theta + \alpha_\pi \frac{1}{n} \sum_i \nabla_\theta \min(w_i(\theta)\mathcal{A}_i, \text{clip}(w_i(\theta), 1-\epsilon, 1+\epsilon)\mathcal{A}_i)$
28:     end for
29: end while
```

Figure 3.8: *Training Algorithm - Proximal Policy Optimization - used to optimize the Policy and Value Network. We use TD(1) (not TD(λ)) as the target that the Value Networks need to predict (taken from Peng et al. (2018))*

3.2.4 MDL/MTL Policy Search Algorithm

The policy search algorithms used for the Multi Headed architecture is really similar to that used for the SDL agents. A thorough description of this methods is covered in the previous **section 3.2.3** . The main difference is that for each domain or task, we created separate learning rates and β constants which influenced each head of the

Policy Networks updated. Regarding the Value Network, we had a separate replay buffer associated with each individual domain or task.

3.3 Experiments and Results

In the last two sections we have describe the reasons why we choose a specific architecture of Artificial Neural Networks (ANN) as a function approximator (FA) parametrizing a policy, and how the Policy Network could be trained by using Policy Gradients (PG). Because of the different issues that Vanilla PG might encounter in environments such as Bipedal Walker, we introduced the reasoning behind several improvements to Vanilla PG that we used, such as: using the Generalized Advantage Estimator (GAE) as the multiplier of the parameters gradients, and how we limited the size of each gradient step either by imposing a constraint that makes each step lie within a trust region (Schulman et al., 2015; Nachum et al., 2018) or by forcing it to stay within a small interval around 1 (Schulman et al., 2017; Weng, 2018). Recall that we also have a Value Networks which is trained in a much simpler manner to solve a nonlinear regression problem as to minimize the distance between the estimate V_ϕ and the empirical discounted reward.

In this section we will describe the results we obtained by undertaking different experiment with the single (SDL) and multi (MDL) domain learning agents. Our experiments not only tested the performance of the agents in the Bipedal Walker environment but also analyze the GAE, PPO and TRPO methods that were motivated as improvements in **section 3.2.3** and the effects of changes in the Policy/Value networks architectures.

The videos related to the results of our experiments can be found in:: `https://www.youtube.com/playlist?list=PLe5MBbUvqqF_Bb7eiwLHhzE6qVeY5P3-X`.

3.3.1 Experiments Setup

Unless stated otherwise, the experiments for both single and multi domain agents were done with the setup presented in **table.** 3.2. We appended a additional feature to the state representation vector consisting of a time feature which is increased by 1e-3 each simulation time step. We also run a running mean filter which normalizes the observations throughout each experiment. Each SDL agent took an average of 4.62

hours[16] to be trainned for 20000 Episodes.

Recall the different domains consist of a constant negative horizontal force that is analogous to ”wind” pushing the agent in the opposite direction to the direction of motion. Each domain has a different wind condition taking values of 0, 1 or 2 .

Table 3.2: *Experiments General Setup*

Hyperparameter	Value
Discount Factor (γ)	0.995
GAE Hyperparam (λ)	0.980
Batch Size (b)[1]	20
KL Target (δ_{KL})	0.01
Clip Objective (ϵ_{clip})	0.2
Offset log-var[2]	1
Init. KL Loss Gain (β)[3]	1
Hinge KL Target Gain (η)	50
Policy Act. Func	ReLU
Policy Training Epochs	20
Policy Init. Learning Rate (α_π) [3]	≈1e-4
Value Act. Func	Tanh
Value Training Epochs	10
Value Learning Rate (α_V)	≈1e-3
Value Replay Buffer Size	Batch size (B) Episodes

[1] In number of *full* episodes

[2] Offset added to log-var trainable variable

[3] β gain & learning rate dynamically adapted based on $D_{KL}(\pi_{\theta_{old}} \| \pi_\theta)$

3.3.2 Preliminary Results

From preliminary single domain experiments that we carried out (plots not included here) it was evident that by replacing the gradient multiplier R_t with the advantage $\hat{A}_t$ we were able to significantly reduce the variance of the gradients. With R_t we obtained agents that did not learn at all (noisy learning curve constantly staggering at

[16]This short time training period was possible when the whole GPU cluster was available for me. When sharing resources with other students it took *much longer*

negative rewards) or in the best of cases showed minor improvements with extremely unstable updates that eventually ”collapsed” (policies performance suddenly dropping and not improving again). Because of this we decided to use Advantage based gradient multipliers for all of the experiments done in this project (please refer to **section 3.2.3** for the initial reasons we introduced the concept of Advantage Estimators).

By estimating the Advantage using the GAE approach $\hat{A}_t^{GAE(\gamma,\lambda)}$ we could further reduce the variance by adjusting the hyperparameter λ. Instead of performing a massive hyper parameter search on the values that affect GAE’s performance we used the results of the 3D Bipedal Locomotion experiments carried out by Schulman et al. (2016). From their experiments done across 9 different trials they concluded the best performance was achieve by $\gamma \in [0.99, 0.995]$ and $\lambda \in [0.96, 0.99]$. Because of this we selected values of $\gamma = 0.995$ and $\lambda = 0.98$ as in our SDL experiments they presented the biggest performance improvement of using $A_t^{GAE(\gamma,\lambda)}$ when compared to the vanilla R_t. We acknowledge that the 3D Bipedal scenario is different from the one being used in this project, however all of their similarities provide a good basis for using the intervals provided by Schulman et al. (2016). Regarding the MDL optimizers, given the positive results that using $\hat{A}_t^{GAE(\gamma,\lambda)}$ had on the performance of the SDL, we decided to use this type of gradient multiplier from the start of our experiments and did not explore the vanilla R_t. Moreover, we also kept GAE’s hyperparameters the same as those set for the SDL experiments.

From the above described results the improvements that GAE imposed in learning was clear; what was not totally clear were the improvements that constraining the gradient had (i.e. the use of Proximal and Trust Region methods). This motivated use to carry on the set of experiments described in the following section.

3.3.3 Effects of Proximal and Trust Region Methods

Despite the improvements achieved by $GAE(\gamma, \lambda)$ we were still experiencing certain instability while learning and for some MDL experiments we actually saw that limiting the gradients updates had a corrosive effect on the agents performance. This motivated us to perform experiments optimizing three different types of loss: *1)* A loss that only imposes the KL penalty on the policy updates (following TRPO’s principle), *2)* A loss that only clips the policy ratios (following PPO’s principle), *3)* A loss which imposes both the KL penalty and policy ratio clipping. We refer to this losses as ”KL”, ”clip”, and ”both”.

Important Note: On the final stages of this project while writing this report I noticed a bug in the training code. I accidentally clipped the policy ratio by $[0, \epsilon_{clip}]$ rather than $[1 - \epsilon_{clip}, 1 + \epsilon_{clip}]$. I had enough time to redo the compromised experiment and the results reported in this section are the corrected results. However I had already made some experimental decisions (choosing the best loss) based on the results with the bug and I did not have time to repeat all the other experiments from the following sections. Despite this, my mistake does *not* invalidate all of the experiments of the project, it only affect in the fact that I did not select the ”clip” loss function as the best of the experiments. For a better understanding of the effects of my mistake I analyze this in the ”Training issues” section (**section 3.4**) in **fig.3.13**.

Given that in non-linear functions approximation there exist a complex relationship between the depth of the network and its performance [17] (Bengio et al., 2009), we decided to experiment with networks of different depth: 3, 5 and 7 hidden layers [18] with the structure of 64 units per layer as outlined in **section 3.2.1** . We are interested in finding the loss function that performs the best across the 3-7 depth spectrum. To reduce the number of experiments required to test each type of loss (kl, clip and both) and each class of experiment (single and multi domain), for this sections we tested the smallest (3 hidden) and biggest (7 hidden) networks and analyzed which one had the best performance trade-off. Further depth experiments are described in the next section.

For the SDL experiments we evaluated a total of 6 separate agents for each of the 2 architectures depths and the 3 loss types. These agents were trained on a environment with *wind = 0* for 6000 episodes (recall that our training is based of N- *complete* episodes, and not randomly samples time steps. Each episodes has on average between 10000 and 32000 time steps). For MDL there were also a total of 6 separate agents; however, they were trained to solve all the wind conditions *wind = 0, 1, 2* for approximately 6000 episodes (had to stop training at 5720 due to GPU cluster time limits). The results of these experiments are presented in **fig. 3.9**.

Regarding SDL performance, ”KL” and ”clip” losses achieved the highest asymp-

[17]As stated in Sutton et al. (1998) *”Training a network with $k+1$ hidden layers can actually result in poorer performance than training a network with k hidden layers, even though the deeper network can represent all the functions that the shallower network can”*

[18]recall that in MTL, a N-hidden layer architecture has $int(N/2)+1$ core hidden layers and each head has $int(N/2)$ hidden layers i.e. a 7 hidden layer MTL agent has 4 core layers and each head has 3 layers plus and dense output layer

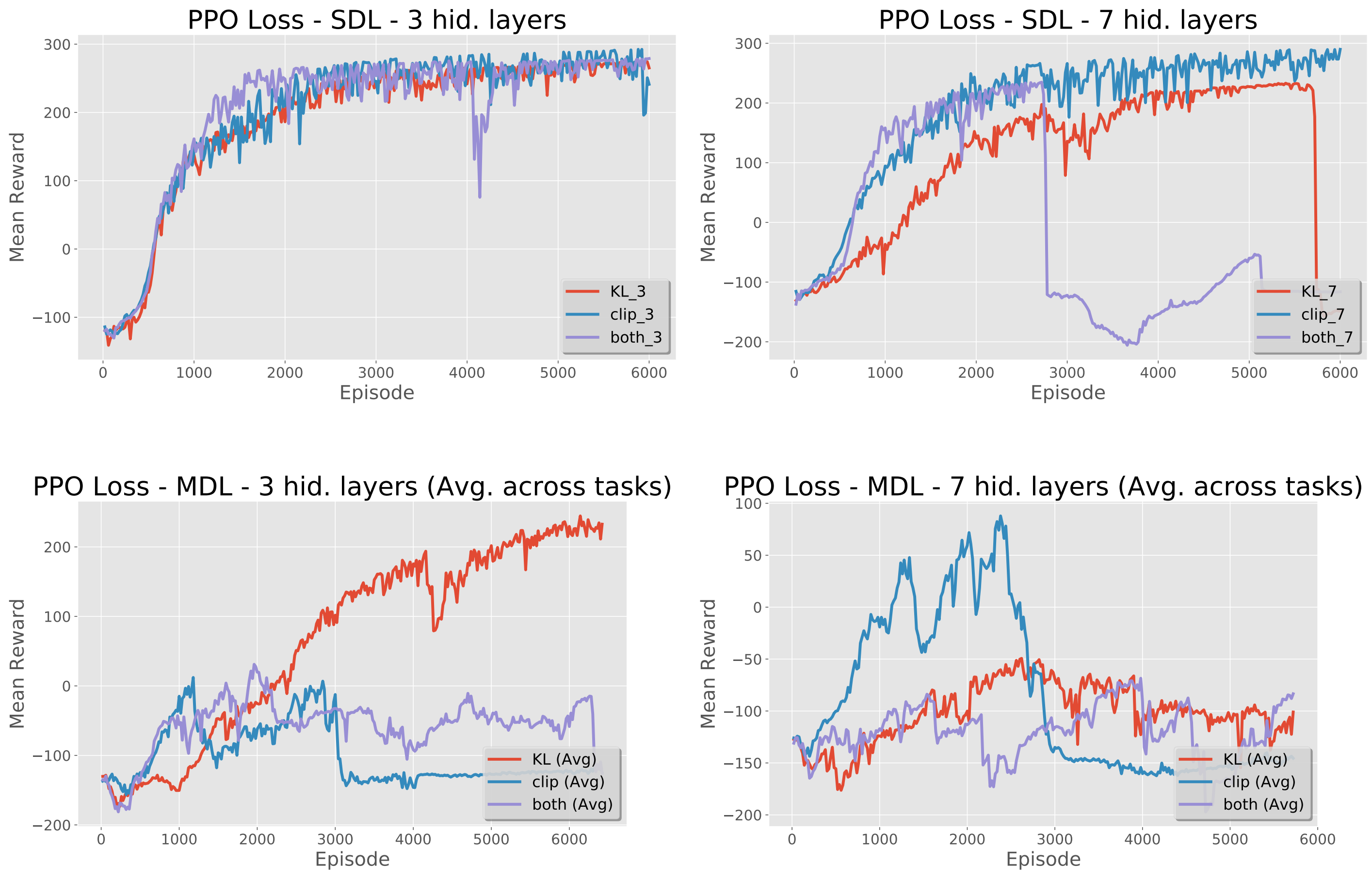

Figure 3.9: *Learning curves testing the performance of several loss functions (KL penalty, policy ratio clipping, and both) and architectures with depth 3 and 7 hidden layers. SDL agents evaluated in domain wind=0 and MDL agents evaluated in wind= 0,1,2. Curves for MDL agents are the average performance across all domains.*

totic reward (end performance). Moreover, ”clip” loss had a better sample efficiency by requiring less episodes to overcome the 200 cumulative reward threshold (threshold at which we could observe in the simulation that the robot had a stable walking gate and successfully managed to get to the end of the environment). Unfortunately do to the bug I did not choose ”clip” as the best SDL loss function, but basd on the results obtained with the bug I chose the ”Both” loss function.

To summarize the results of each experiment into a single scalar we drew inspiration from the approach taken in the original PPO paper experiments(Schulman et al., 2017). They score each agent based on the average total reward of the last 10 policy/value function updates. Given that each of our updates involves a much larger number of time steps (we use full episodes as the training set), we decided to use the final 10 network updates to compute the average. These scores can be seen in **table. 3.3** and **table. 3.4**.

Table 3.3: *Results for SDL loss type experiments. Performance averaged over last 10 Policy/Value updates for each loss type - network depth*

Loss - Depth	Avg. Reward	Avg. Loss Score
KL - 3 hid	272.657	253.567
KL - 7 hid	234.478	
both - 3 hid	272.637	78.070
both - 7 hid	-116.496	
clip - 3 hid	253.001	**265.505**
clip - 7 hid	278.009	

Table 3.4: *Results for MDL loss type experiments. Performance obtained by first averaging cumulative reward across all domains and then computing the average of this over the last last 10 Policy/Value updates for each loss type - network depth*

Loss - Depth	Avg. Reward[1]	Avg. Loss Score
KL - 3 hid	227.807	**57.035**
KL - 7 hid	-113.736	
both - 3 hid	-78.613	-73.534
both - 7 hid	-68.456	
clip - 3 hid	-121.153	-134.271
clip - 7 hid	-147.389	

[1] Average of mean performance across all domains

Based on the results shown on our summary tables we decided that for all subsequent experiments SDL agents would be trained with a ”both” type loss (again, this is due to the erroneous results obtained by the bug) and MDL agents with the ”KL” loss (this results this persist across the bug and bug-free results)

3.3.4 Effects of Networks Depth

Once we had established which type of agent most benefited from using clipped policy ratios and KL divergence constraints, it was important to evaluate which network depth was the most suitable for each of the agents. All though we had already done a partial evaluation of the smallest and biggest depth; the highly unstable learning of the 7 layered architecture motivated us to test all of the proposed depths using the best loss functions which we selected in the previous section. Regarding Single Domain agents, for each domain we tested agents with 3 different depths (for a total of 9 experiments) with the purpose of selecting the most appropriate depth for each single domain. For Multi Domain learning we did 3 experiments consisting of MDL agents with depth 3, 5 and 7.

As the agents with the deepest networks and the MDL agents were requiring an excessive amount of time to learn we decided to reduce the batch size from $B = 20$ to $B = 5$. Although this enabled us to train all agents up to convergence (10000 and 16000 episodes) in a manageable time, it increased the noise present in the learning curve. To be able to compare the performance among learning curves we used a Savitzky–Golay digital filter to smooth the data. This filter consist of a sliding window moving across the signal; for each time window movement a low degree polynomial is fit to the sub set of points. Afterwards, the signal is reconstructed by combining each fitted polynomial through filter coefficients. This type of filter is capable of ”increasing the signal-to-noise ratio without greatly distorting the signal” (Press et al., 1992); we also took care of setting the window size and polynomial order to be small enough to not affect the true nature of each curve. The results of the *complete experiments done for 10000 and 16000 episodes* are shown in **appendix. A** in **fig. A.1**.

From analyzing the complete experiments of **fig. A.1** we noticed that MDL agents became increasingly unstable when more episodes were fed into the network. Our initial motivation to reduce the batch size to $B = 5$ was to fully train our MDL agent; however, this extended training period ended up having an adverse effect on the multi headed network stability. Because of this we decided to only use the results up to 7000 episodes, were MDL training was still stable. Results for all of the SDL and (truncated) MDL experiments are shown in **fig. 3.10**.

From the SDL learning curves one can conclude that for domain wind=0 all depths have a similar sample efficiency learning at a very similar rate, and all of them reach closely similar asymptotic rewards. However, for the more challenging domains the simplest architecture consisting of 3 hidden layers has a more stable learning (as seen in the smoother curve of depth=3 for the SDL wind=1.0 experiment) and also achieves a higher asymptotic reward when compared to 5 and 7 hidden layers. Moreover, the 3 hidden layer architecture is the only setting capable of surpassing the 200 reward threshold in all of the domains.

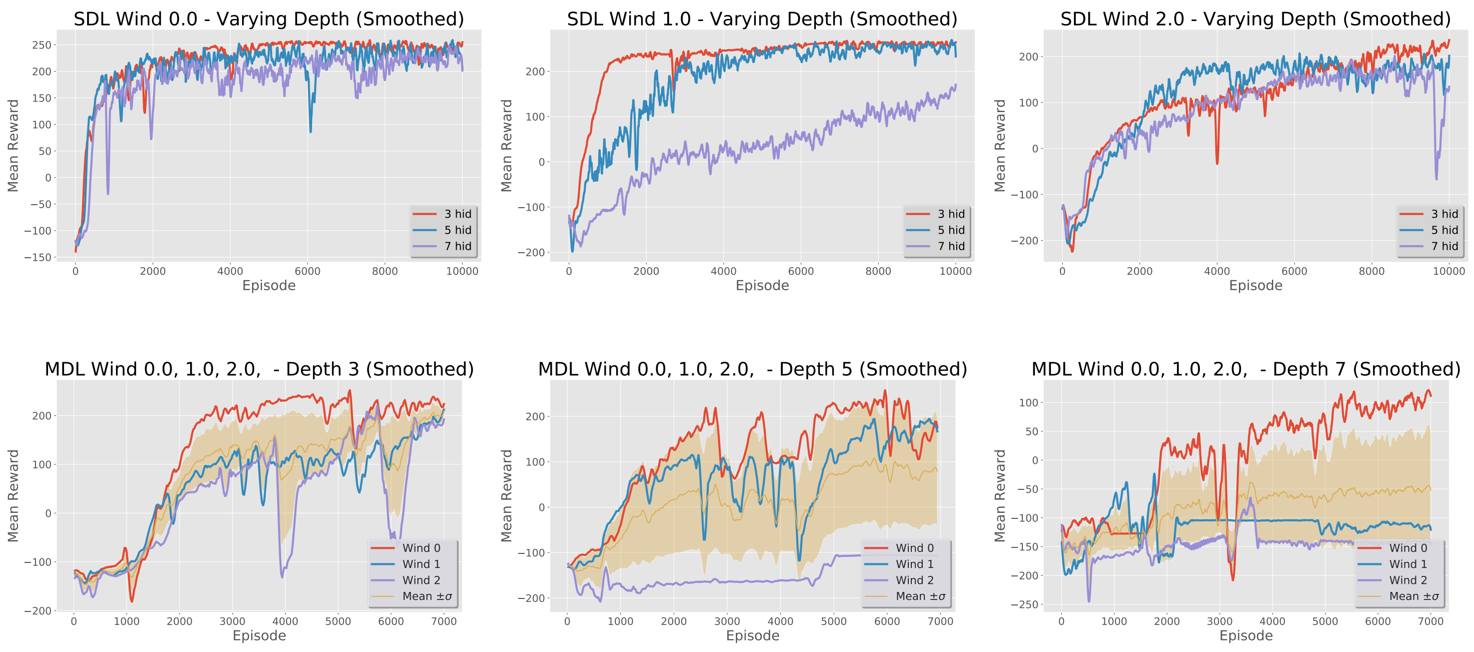

Figure 3.10: *Learning curves for all domains testing the performance for networks depth 3, 5 and 7 with the chosen loss functions ("both" for SDL and "KL" for MDL). On MDL, results have been truncated at 7000 episode do to learning instability. Yellow lined curves indicate the average performance across all domains and the shaded area indicates its standard deviation.*

For MDL experiments we had similar results. The 3 hidden layer multi headed networks was the one achieving the highest average reward across all domains (if the reader verifies the individual performance in each domain, 3 hidden layers has the best cumulative reward in each single domain when compared to the other two architectures). Another advantage of this architecture is that it has the smallest performance variance across domains. Looking at the shaded yellow area of the learning curves, this architecture was able to achieve a high performance across all the domains; being the only architecture who's learning did not collapse in some domain i.e. both 5 and 7 depth experiments had no learning (nearly flat learning curves) in domains wind=2 and wind=1,2.

As a more exact measure of performance, we again used the approach of scoring each agent based on the average performance in the last 10 policy/value updates, a single scalar summarizing the performance of each experiment can be seen in **table.** 3.5 and **table.** 3.6. Based on these scores we had the same conclusions - the 3 hidden layer network is the best across all domains for single domain learning and it is also the best for MDL. Because of all of the above the concluding outcome of this set of experiments is that a **3 hidden layered networks is the best configuration for all single learning domains and for the multi learning agent**.

Table 3.5: *Results for SDL Network Depth experiments. Performance averaged over last 10 Policy/Value updates for each Domain - Network depth*

DOMAIN	DEPTH	AVG. REWARD	BEST DEPTH
WIND=0	3 HID	**248.795**	3 HID
	5 HID	221.065	
	7 HID	222.898	
WIND=1	3 HID	**263.074**	3 HID
	5 HID	255.390	
	7 HID	160.758	
WIND=2	3 HID	**227.461**	3 HID
	5 HID	181.641	
	7 HID	126.568	

Table 3.6: *Results for MDL Network Depth experiments truncated at 7000 epochs do to prolonged training time instability. Performance computed as the average over the last 10 Policy/Value updates of the mean reward across all domains for each Network depth*

DOMAIN	DEPTH	AVG. REWARD[1]	BEST DEPTH
WIND=0,1,2	3 HID	**202.306**	3 HID
	5 HID	81.071	
	7 HID	-46.161	

[1] Average of mean performance across all domains

A possible reason explaining the 3 hidden layer networks dominance is its simplicity and ease of training. Deeper networks effectively provide additional representation power and are capable of learning more complex features and mappings. However, policy gradient is already a unstable process and the exponential number of additional parameters introduced in deeper architectures makes the training of such networks much more difficult and erratic. In addition, deeper networks have a larger variance regarding which actions is selected at each time step [19], producing greater variance in the advantage estimator and causing further instability in the learning process. Finally,

[19]Given the large state-to-actions parametrization, for similar states the networks can produce radically different actions, thus creating highly variable r_t at each time step

as smaller networks require less data to be fitted, 3 hidden layers are able to reach a steady gate much faster and from there on continue a stable improvement. All in all, its kind of a Ockham's razor principle, the 3 hidden layer networks is the simplest and makes the least assumptions about the state-action mapping - possibly avoiding extreme overfitting and greater robustness across all states of the environment.

3.3.5 Effects of Variable Networks' Structure

In addition to finding the optimal network depth , we further explored if changing our defined structure configuration of 64 units per layer had any positive effect on the agents over all performance. The motivation to perform this experiments was based on the wide success of ANN structures who first map high dimensional input features into a sparse representation (a ”code”), and the map such code into the final output. This kind of encoder-decoder structure, roughly based on autoencoders, has proven successful in image classification problems using Convolutional Neural Networks and image generation using Generative Adversarial Networks.

For both SDL and MDL agents we evaluated the effects of: *1)* A network who's width decreased at each subsequent layer (such as Encoder structure, like a funnel), *2)* Networks who's width initially decreased and towards the output layer it increased again (such as a Encoder-Decoder structure, like a horizontal hyperbola). As in the previous section the best results were obtained for 3 hidden layered networks, for the first test (encoder) we tested a structure of 125-50-25 units per layer, and for the second test (Encoder-Decoder) we tested a structure of 125-50-125. The particular values chosen for the encoder and encoder-decoder structure are based the structure tested on the GAE original paper (Schulman et al., 2016) and on a extensive review paper of Henderson et al. (2018); in both paper the 100-50-25 structure is tested. As before, we used the loss functions which gave the best results on ”Effects of Proximal and Trust Region Methods” section.

The results of this experiments are summarized with the metric we have used in the other sections. Results are presented in **table. 3.7** and **table. 3.8**. For reference, we include the score of the default structure 64-64-64 on each table.

From the SDL agents results it is difficult to extract a precise conclusion, given that for each domain a different structure was the best. Regarding the MDL experiments, I hypothesize that the boost in performance achieved by the 125-50-125 structure could

Table 3.7: *Results for SDL Network Structure experiments. Performance averaged over last 10 Policy/Value updates for each Domain - Structure*

Domain	Structure	Avg. Reward	Best Structure
	64-64-64	248.795	
wind=0	125-50-25	**287.460**	125-50-25
	125-50-125	275.961	
	64-64-64	**263.074**	
wind=1	125-50-25	251.403	64-64-64
	125-50-125	249.499	
	64-64-64	227.461	
wind=2	125-50-25	253.888	125-50-125
	125-50-125	**262.653**	

Table 3.8: *Results for MDL Network Structure experiments. Performance computed as the average over the last 10 Policy/Value updates of the mean reward across all domains for each Network Structure*

Domain	Structure	Avg. Reward[1]	Best Structure
	64-64-64	202.306	
wind=0,1,2	125-50-25	238.426	125-50-125
	125-50-125	**260.002**	

[1] Average of mean performance across all domains

had been due to the additional units in each tasks head. Given that this structure has almost twice the units than the default one, this structure enables each head to better specialize on the given domain, achieving a higher mean reward across domains and even a much more stable learning.

To support this last statement the learning curves for the MDL 125-50-125 structure is shown in **fig.** 3.11. In contrast to all of the other MDL experiments from previous sections which as seen in **fig.** 3.9 and **fig.** 3.10 are quite unstable; this MDL structure exhibited a extremely steady training: the performance did not collapse for any of the domains, we did not have to stop the experiment before convergence, and the agent achieves a really similar (and high) performance for all domains (as seen in the small standard deviation across domains performance). We did not see this same stability effects for the 125-50-25 structure (plot not shown). This further highlights the benefit of having wider head structures which enhances performance and multi domain learning stability.

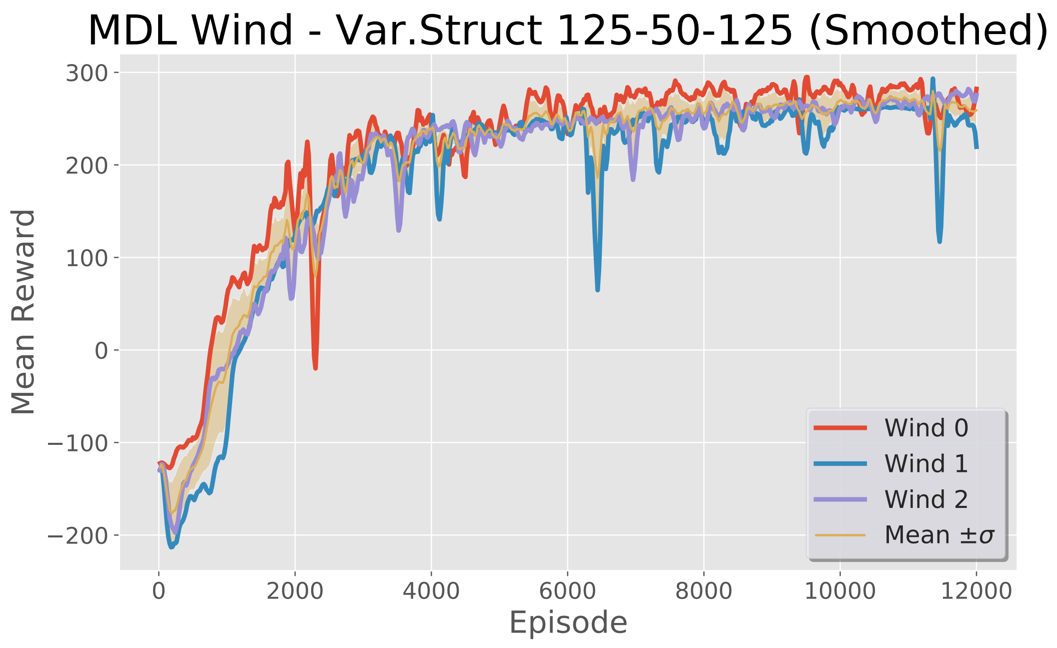

Figure 3.11: *Learning Curve for MDL structure 125-50-125. This structure boosted the MDL agents asymptotic performance and overall learning stability*

Concluding from our results, the **best SDL structure is 125-50-25 for wind=0, 64-64-64 for wind=1 and 125-50-125 for wind=2. The best MDL structure is 125-50-125.**

3.3.6 Walking Under Various Wind Conditions

The results obtained in our previous experiments allowed us to establish the best loss function, appropriate network depth and network structure (width) for our single and multi domain learning agents. The following step was to perform a final comparison analyzing if our proposed multi headed MDL architecture did have any performance improvements with respect to the vanilla SDL agents. To perform this comparison we use the best SDL and MDL configurations as indicated from the results of: Loss (**table. 3.3**, **3.4**), Depth (**table. 3.5**), **3.6** and Structure (**table. 3.7**, **3.8**).

Th final agents comparison is shown in **fig. 3.12** and **table. 3.9**. Note that in this table we introduce a new metric $\bar{R}_{all}$ which is computed by averaging the reward across the entire training period. This metric is a heuristic to measure the sample efficiency of each agent as it favours fast learning. The metric that we have been using to report all past results is here denoted by $\bar{R}_{10}$; since it averages the reward over the last 10 updates this metrics favours high final performance. We used this two metrics as they are interesting heuristics to measuring agents performance per introduced in section 6.4 of Schulman et al. (2017).

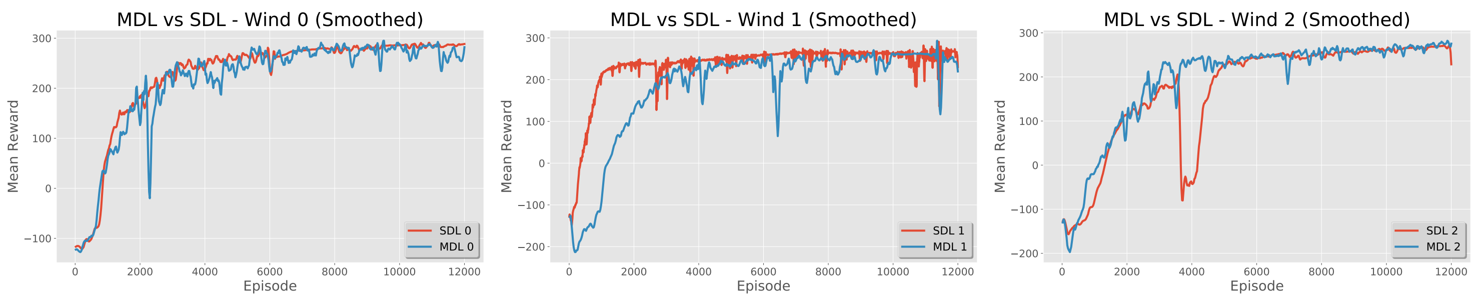

Figure 3.12: *Learning curves comparing the performance of the MDL agent relative to each SDL agent for all domains.*

Table 3.9: *Comparison of MDL and SDL agents for multiple domains in the Bipedal Walker environment. $\bar{R}_{all}$ indicates average reward across the whole training period. $\bar{R}_{10}$ measures the average reward over the last 10 Policy/Value updates. $\frac{\mathbf{MDL}}{\mathbf{SDL}}$ quantifies the relative final performance using $\bar{R}_{10}$ of the MDL agent compared with the SDL agents.*

Domain	$\bar{R}_{all}$ **(agent)**	$\bar{R}_{10}$ **(agent)**	$\frac{\mathbf{MDL}}{\mathbf{SDL}}$	**Best Agent**
wind=0	**226.469** (SDL) 214.793 (MDL)	**287.735** (SDL) 261.712 (MDL)	90.955%	SDL
wind=1	**232.027** (SDL) 182.677 (MDL)	**263.074** (SDL) 242.836 (MDL)	92.307%	SDL
wind=2	170.746 (SDL) **194.548** (MDL)	262.385(SDL) **275.457** (MDL)	104.981%	MDL

Our results indicate that there is a nice overlap between the learning curves exhibited by SDL and MDL, with SDL having a slightly better asymptotic reward for the two first domains and MDL actually having a better performance for wind=2.0. This can further be corroborated by our metrics, which show that for all domains MDL achieved a comparable performance (within less than 9%) when compared to SDL; and for wind=2.0 it even gained a 4.981% improvement on the asymptotic reward and a 13.940 % improvement on the sample efficiency. This results supports our hypothesis as it is clear that the Multi Headed architecture was able to achieve high performance throughout all tasks and in one of them parallel training actually enhanced the agents sample efficiency and final performance. Though MTL agent is solving a much harder problem than STL, over average the MTL agent did 96.081 % as good as the STL.

MDL is better than SDL in the wind=2 domain were high winds make any initial progress quite challenging since the bipedal walker usually gets knock to the ground from the beginning (we saw this on the training videos). This could be an indicator that learning to walk in more simple scenarios were initial progress is easier to convey, could pose a benefit when trying to achieve good performance in a more complex domain. The progress made in the simple domains can be stored in the Core layers and then used on the complex domain to boost initial learning speed.

MDL training is notably more noisy and unstable compared to SDL. A possible cause of this instability are the way in which parameters are updated in the proposed MDL architecture. Recall that in the Multi Headed network each head is sequentially updated , starting from head number 0 to head number 1. If two given domains have (nearly) opposite optimal policies, the gradients computed for one of the domains might move the parameters in the opposite directions compared to the gradients computed in a second domain. This constant parameter jittering might be causing the large fluctuations observed in the learning curves. However, given the complex training dynamics of this large networks it is difficult to truly analyze if this is whats happening; since even if trained sequentially the overall effect of gradients pointing in different directions might cancel off once all of the heads are trained (however, it is difficult to prove or disprove this statement).

3.4 Training issues

The following plot (**fig. 3.13**) shows how the bug that I made in the code affects the clipping of the policy ratio $w(\theta$. The wrong clipping range produce by the bug, for positive advantages does not encourage significant improvements (only with ϵ as upper bound, so almost no improvements are made at each iteration) and for negative advantages it does encourage high changes which might make the the old and new policies diverge not much, and as a result make the whole training unstable

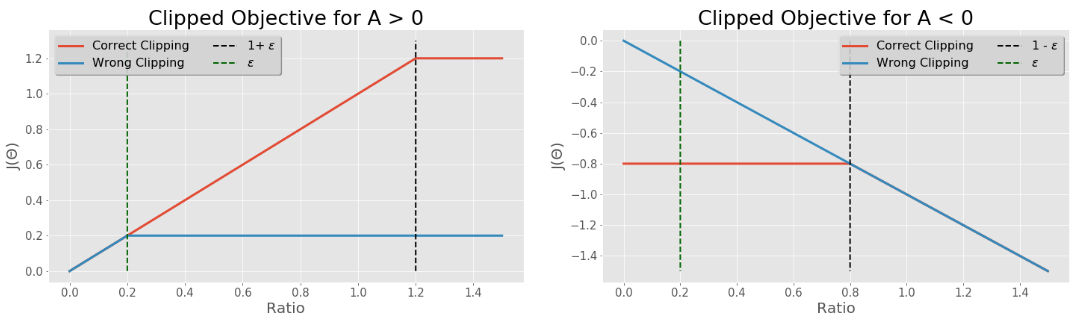

***Figure 3.13:** Effects of the wrong clipping range on the PPO objective function $J(\theta)$*

Chapter 4

Spacecraft : Learning to Fly and Land

Once we had explored the capabilities of our proposed Multi Headed architecture to learn several domains ins parallel, we decided to test if such architecture could also tackle Multi Task Learning (MTL) problems. In this type of problem instead of having a scenario were despite changes on the MDP's structure (i.e. changing wind or gravity) the overall objective is the same (same reward function), in MTL the agents' objective is to maximize different rewards functions.

To perform the MTL experiment we decided to use the lunar lander environment - an environment in which the agent is a spacecraft situated in the moon which is capable of controlling two engines. We choose this environment since it easily enabled us to learn multiple related but certainly different task, such as flying and landing.

In this chapter we describe the details of the Lunar Lander simulation environment that we used, and the different single and multi task experiments we performed which consist of learning the tasks of landing in different positions of the environment and learning how to fly.

4.1 Lunar Lander Environment

The Lunar Lander environment [1] consist of a spacecraft with a main body and two legs **fig. 4.1**. The spacecraft can navigate the environment by ejecting fuel (particles colored as orange circles) with two side engines (left and right) and a main engine (bottom of the spacecraft main body). There is not limit on the amount of fuel available to the agent. The scenario is situated in the moon , were there is a free moving space (colored in black) and a irregular terrain consisting of 11 segments with variable slope (colored in white). Each time a episode is instantiated the length and slope of each segment is randomly changed. Because of the nature of the tasks we intend to solve with the MTL agent, we modified the scenario to have totally flat far left, far right and center segments - guaranteeing that the agent could always find a flat surface to land in.

The environment as originally coded by Open AI is set to have the main goal of landing in the coordinate $(0,0)$, where a pair of sky poles (colored in yellow) indicate the center of the scenario. An episode ends if the lander crashes (main body comes in contact with terrain) or if it comes to rest.

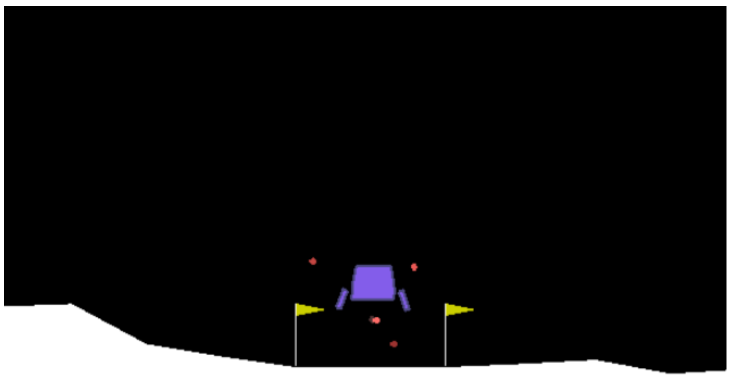

Figure 4.1: *Lunar Lander Environment - Agent in purple and jet propulsion particles in orange*

Observation Space

Each state in the Lunar Lander environment consist of a 8 dimensional vector describing the position of the spacecraft, its velocity, inclination (angle), angular velocity, and a binary variable indicating if the spacecraft legs are touching the ground. The complete state space is summarized in **table. 4.1**.

[1] https://gym.openai.com/envs/LunarLander-v2/

Table 4.1: *Lunar Lander Observation and Action Space*

Observation	Min. Value	Max. Value
X - position (X_{pos})	-1	+1
Y - position (Y_{pos})	0	+1
X - velocity (X_{vel})	-∞	+∞
Y - velocity (Y_{vel})	-∞	+∞
Angle (θ_l) [1]	-∞	+∞
Angular velocity (ω_l)	-∞	+∞
Left Leg Contact (binary) (L_c)	0	1
Right Leg Contact (binary) (R_c)	0	1
Action	**Min. Value**	**Max. Value**
Main Engine Power	-1	1
Side Engines Power	-1	1

[1] Not actually the raw angle, its the cumulative angle per episode

Action Space

The agent can decided over two continuous variable controlling the main and sides engines. The first variable controls the main engine, if it has a value greater than 0 then the main engines power is set to a value proportional to the value of the action. If it is less than 0 the main engine is not turned on. The second variable controls the side engines; its sign determines which engine will be activate - negative action values activate the left engine and positive values activate the right engine. The side engines will only be turned on if the action value is greater that | 0.5 |. The main engine has a significantly larger propulsion power compared to the two side engines (≈20 times higher). See **table. 4.1** for a summary of the action space.

Reward Function

The default reward function of this environment is to land the spacecraft at the positions $(0,0)$ with the smallest x and y velocity and with a angle close to zero. As we are interested in maximizing multiple reward functions in parallel, we modified this default reward function and constructed new ones. These are described in the methods **section 4.2** .

4.2 Methods

4.2.1 Function Approximator and Policy Search

The multi headed architecture introduced in **section 3.2.2** is also suitable for a multi task problem. Because of this we used the same Function Approximator and policy gradient algorithms as those used to solve the multi domain learning problems (refer to **section 3.2**).

4.2.2 Rewards Design

We defined three different task that the MTL agent could attempt to solve: landing in the center of the terrain (referred to as ”goal”), landing as far as possible from the center of the terrain (”not goal”), and flying (”fly”). In order to translate this task descriptions into numerical quantities we designed three separate reward functions introduced in the following paragraph. All three reward functions give a reward of -300 if the spacecraft leaves the visible area of the environment ($| X_{pos} | \geq 1.025$ and $Y_{pos} \geq 1.025$) or it crashes against the terrain; and a reward of +300 if the spacecraft either comes to rest or manages to get to the end of the episode (after the maximum number of time steps have elapsed) without crashing. For both the ”goal” and ”not goal” reward functions enforced that less fuel spent is better, penalizing the agent for each particle shot through the main and side engines.

All of the reward functions that we designed are *shaped rewards functions*, as they give a reward throughout the whole simulation as the agent gets closer to the end goal (in contrast to sparse reward function which only produce a reward signal when the main goal state is reached and no reward anywhere else).

”Goal” Reward Function

For the task of landing in the center of the terrain we defined a reward function (**4.1**) similar to Open AI’s default. The reward function heavily penalizes lander positions different from zero, large magnitudes of the spacecrafts velocity, and angles different from zero. It positively rewards any contact of the landers’ legs with the ground. This function is thus maximized by moving the lander to the center of environment (zero position) with a zero velocity and a perfectly stable spacecraft (zero angle) .

$$r_t \propto -150 \,| X_{pos} | -150 \,| Y_{pos} | -100. \sqrt{X_{vel}^2 + Y_{vel}^2} - 100. \,| \theta_l | +10L_c + 10R_c \tag{4.1}$$

”Not Goal” Reward Function

For the task of landing away of the center of the terrain we had to design a reward function that was similar to the ”goal” reward function; because of this we used the same equation as before (**4.1**) but changing the term related to X_{pos}. Instead of rewarding a X_{pos} of zero the new reward function had to penalize any position close to the center and also positions approaching the far left (-1) and right (+1) of the scenario (given that surpassing the environments limits results in a negative reward). To achieve this we fitted a 4th degree polynomial implementing the above described penalty. As seen in **fig. 4.2**, the ”Not goal” penalty heavily penalizes positions close to zero and has two maximums at $X_{pos} = \pm 0.875$ (unlike the ”goal” task which has a optimal value ot $X_{pos} = 0$). Adding this term with the rest of the equations we obtained the final reward function for the ”not goal” task.

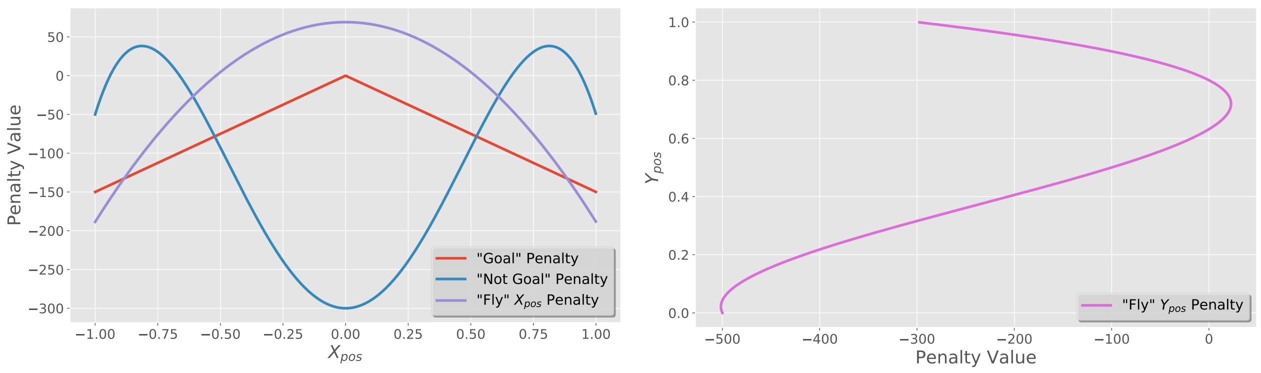

Figure 4.2: *Reward Function penalties for X_{pos} and Y_{pos} for the three tasks*

”Fly” Reward Function

Finally, to encourage the agent to fly we designed the reward function described by equation (**4.2**). In this task we wanted to encourage the agent to maintain a position close to $X_{pos} \in [-0.50, +0.50]$ and penalize any attempt to approach the end of the environment. To implement this penalty we fitted 2nd degree polynomial (taking the place of $f(X_{pos})$) which is shown as the purple curve in the left plot of **fig. 4.2**. Regarding the vertical positions of the spacecraft we needed to penalize any attempt of landing and also penalize flying at high altitudes because of the risk of exiting the environment. To implement this penalty we fitted 3nd degree polynomial (taking the place of $g(Y_{pos})$) which is shown as the purple curve on the right plot of **fig. 4.2**; establishing a optimal flying altitude of $Y_{pos} \in [0.6, 0.8[$. Negative rewards are also produced for large magnitudes of the agents velocity (but not as harsh as the previous functions), angles too far from zero, and any contact of the lander’ legs with the ground are heavily penalized.

$$r_t \propto f(X_{pos}) + g(Y_{pos}) - 5.\sqrt{X_{vel}^2 + Y_{vel}^2} - 10.\,|\,\theta_l\,| - 100L_c - 100R_c \tag{4.2}$$

Note the proportionality sign on the reward function equations. At each time step the rewards are not exactly computed based on this equation, but rather based on the difference between the current state reward value r_t and the value in the previous time step r_{t-1}. Because of this it is not clear how to identify the maximum reward a optimal agent would obtain, and thus, its difficult to normalize all the rewards so that it has the same scale across all tasks.

4.3 Experiments and Results

4.3.1 Experiments Setup

The same experimental setup as that outlined for the bipedal walker experiments (**section 3.3.1**) was set for all evaluations of the Lunar Lander tasks. Please refer to **table.** 3.2 for a description of the Networks hyperparmeters used for the experiments. For all MTL experiments we did not carry the same loss, network depth and network structure experiments we did for the MDL evaluations; instead, we use the best results from the previous chapter to determine the STL and MTL networks architecture. All STL agents policy and value networks have 3 hidden layers, each on made out of 64 units and trained with the ”both” loss function. The MTL agent networks also have 3 hidden layers (which for MTL translates into 2 core hidden layers and three heads each with 1 hidden layer and 1 output layer) with 64 units which are trained using the ”KL” loss function.

Recall that the different tasks consist of Landing on the center of the terrain (”Goal”), Landing away from the center (”Not Goal”) and learning how to fly (”Fly”).

4.3.2 Flying and Landing

Once we had established the reward function for each of the three tasks, we could train the multi headed networks to solve the multi task learning problems. The results for this experiments are shown in **fig.** 4.3. Given that we trained all agents for the same amount of episodes (12000 episodes), this figures directly compare in a single plot the learning curves for the single and multi task agents .

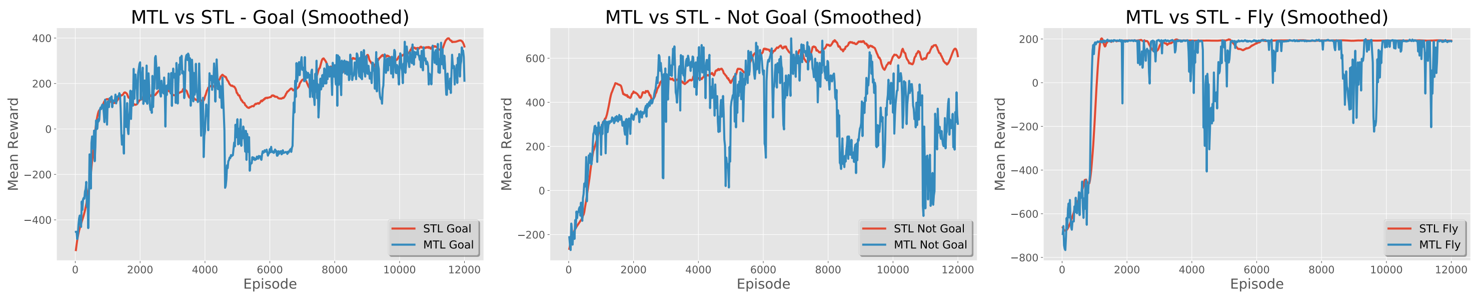

Figure 4.3: *Learning curves comparing the performance of the MTL agent relative to each STL agent for all tasks.*

Unlike the MDL Bipedal Walker experiments in which we noted that the multi headed architecture became unstable as more experience was fed into it; for the MTL tests we did not see this behavior (to see a plot with all the MDL learning curves refer to **section A.2**). As seen in the plots all of the tasks are solved at a really early stage of the training ($\approx$ 1500 episodes) and from there on the performance fluctuates in a relatively stable manner (no extreme drop in performance is seen). The relative stability of MTL compared to MDL does not necessarily imply that learning to maximize different rewards is easier / more stable than learning to act in different domains. This stability might rather be explained by the difference in complexity of the environments involved in each problem, as Bipedal Walker has a observation and action space twice as big (Num. Obs = 24, Num. Act= 4) compared to Lunar Lander. Regarding the STL experiments we observe the same fast learning behavior for each of the three STL agents (recall that each STL curve represents a totally independent network, unlike the MTL curves representing a single multi headed network).

By contrasting the MTL curves to that of each STL, we do not observe any direct benefit of the parallel training to which the MTL agent was exposed. Both type of agents have a similar sample efficiency (as see on the learning speed) and STL agents achieve a slightly higher asymptotic reward.

As mentioned at the end of last section each task has a different reward scale, direct comparison of performance performance across task is not possible. **Table. 4.2** compares the performance of the multi headed MTL agent relative to each of the STL agents using metric $\bar{R}_{all}$ which is the average reward across the entire training period (a measure of fast learning) and $\bar{R}_{10}$ which is the average reward over the last

10 episodes of training (a measure of final performance) . From our results we can observe that for the ”Goal” and ”Fly” task MTL was capable of achieving a really similar asymptotic performance compared to each STL agent. However, for all task the STL agent performed better than the MTL agent. This does not falsify our initial hypothesis, given that the problem that our single MTL agent is solving is much more difficult than those solved by each individual STL. The MTL agent is a single function approximator trained in parallel to solve all tasks. Because of this, even if each tasks performance is slightly less than that of the STL agents, having a comparable performance already demonstrates the capability of the multi headed architecture of achieving parallel learning and demonstrates that this agent is capable of learning to maximize multiple reward functions at the same time. On average, the MTL agent performed 74.983% as good as each STL agent. From the $\bar{R}_{all}$, MTL’s parallel task learning does not produce any improvement regarding learning speed.

Table 4.2: *Comparison of MTL and STL agents for multiple task in the Lunar Lander environment. $\bar{R}_{all}$ indicates average reward across the whole training period. $\bar{R}_{10}$ measures the average reward over the last 10 Policy/Value updates. $\frac{\textbf{MTL}}{\textbf{STL}}$ quantifies the relative final performance using $\bar{R}_{10}$ of the MTL agent compared with the STL agents.*

Task	$\bar{R}_{all}$ **(agent)**	$\bar{R}_{10}$ **(agent)**	$\frac{\textbf{MTL}}{\textbf{STL}}$	**Best Agent**
Goal	**192.891** (STL) 134.495 (MTL)	**382.747** (STL) 298.388 (MTL)	77.959%	STL
Not Goal	**518.796** (STL) 377.908 (MTL)	**631.498** (STL) 305.070 (MTL)	48.308%	STL
Fly	**127.238** (STL) 93.731 (MTL)	**192.446** (STL) 189.911 (MTL)	98.682%	STL

Chapter 5

Discussion and Future Work

5.1 Discussion

In this project we evaluated the capacity of a single agent to learn to maximize its expected reward when being trained in multiple domains or tasks. We proposed a hard parameter sharing architecture - the Multi Headed Networks - which exhibited a competitive performance when compared to multiple vanilla SDL/STL agents. The multi domain learning experiments carried on the Bipedal Walker scenario validated our hypothesis given that the proposed Multi Headed architecture was able to achieve a high performance across all domains. For one of the domains, it produced a 4.981% improvement on the asymptotic reward and a 13.940 % improvement on the sample efficiency when compared to single domain learning agents. This answers our initial research question given that parallel training was indeed able to produce benefits regarding the mean rewards and learning speed.

For multi tasks learning in the Lunar Lander scenario, all thought we did not observe any direct benefit of training across parallel tasks, our MTL agent was able to achieve a comparable performance (**table. 4.2**) relative to each STL agent. On average, it performed 74.983% as good as the other single task agents.

We also showed that despite the additional representational power of deeper networks with 5 or 7 hidden layers, the simplest 3 hidden layer model had a far higher performance. Our results match with those published in the literature such as DDPG (Lillicrap et al., 2015), PPO, TRPO, and ACKTR (Wu et al., 2017), were relatively shallow networks (almost all of them used 2 hidden layered networks with a structure of 64 units per layer) are used to train agents in even more complex scenario of 3D bipedal walkers, 3D labyrinth's and robots with more complex morphology's. Given the success of very deep networks in the field of computer vision, there might be a unexplored potential of using deeper function approximator in RL. However, for such deep explorations more stable policy gradient constraints are needed.

A flaw of the multi headed MDL/MTL architecture that we proposed is that it becomes unstable for long training periods. As more episodes are fed into the multi headed networks the parameters updates become extremely noisy and for some domain/task the networks' perfomrnace totally collapses as seen in **fig. A.1**. However, we demonstrated that this learning instability can be greatly improved by increasing the amount of units on the heads hidden layers (see **fig. 3.11**). By having a structure of 125-50-125 we were able to improve the MDL's agent final reward and stabilize the learning throughout the whole training process.

A further limitation of the proposed architecture is the problem of "What to share". In the complex environments that the RL agent tries to solve it is not trivial to determine at which point the Network should be divided into common layers and specialized layers. Although there exist different approaches such as feature selection (based on regularization and sparsity) or the Low rank approach do determine the relationship among tasks (Zhang and Yang, 2017); deciding the common / shared split is hard. This weakness highlights the benefits of methods using soft parameter sharing which automatically choose how different layers / features can be shared. For instance, future directions could explore architectures such as the Cross-stitch Networks (Misra et al., 2016) or Path Net (Fernando et al., 2017) which create multiple ANN or modules, and through soft-gating methods learn which features extracted by other layers or models could be linearly combined to benefit the overall performance in the MTL problem.

5.2 Improvements and Future Work

The way in which we *sequentially* updated the network for each domain/task might be one of the causes producing the noisy learning of MDL/MTL agents; as for domains/tasks producing gradients in totally different directions the networks might be subject to a constant jittering of parameters. An improvement over this sequential training would be to compute and store the gradients based on the episodes across all domains/tasks *without applying them*, then averaging all gradients for the core layers, and finally applying this average gradient to update the parameters of the core layers.

Although conceptually different from MDL/ MTL learning, recent progress related to *Domain Generalization* [1] could potentially be used for MDL. OpenAI (Marcin Andrychowicz, 2018) was able to effectively transfer a robotic manipulation policy solely learnt in simulation to a real life robot. This was achieved by training the policy on millions of *slightly* different simulations of the robot - a method called *Domain Randomization* (DR). From a domain point of view one can see the training + test set (randomized simulations + real robot) as multiple domains with: different physics (friction, environmental noise), actions characteristics (motor backlash, motor delays), and observations (colors of robot, camera positions, light condition). From this perspective their method has effectively achieved MDL - highlighting that training with DR could potentially enable learning a policy capable of achieving high performance across a distribution of domains (i.e. solving MDL problems). A major limitation of this approach would be to create the highly engineered randomization space, given that as described in Marcin Andrychowicz (2018) they carefully randomized specific aspects of the simulation (delays in the robots actuated tendons, forces applied to the objects being manipulated etc.). This type of hand crafted features make this methods less flexible to adapt to other problems.

The lack of robustness of traditional (non multi task, transfer or domain adaptation) models to small changes in the environment [2] (as demonstrated by Kansky et al. (2017)) is a clear indicator of the lack of generalization that DRL suffers from. One might think these models are simply catastrophically overfitting to the training data they are trained in not allowing them to generalize to test states $\boldsymbol{S}^*$ sampled from *marginally* different distributions to that in which they were trained. One of the core methods used to prevent overfitting of large ANN models is dropout, in which at train time hidden units are randomly removed with probability p in the forward and backwards pass. Such unit's removal prevents overfitting (more specifically complex co-adaptations between unit Srivastava et al. (2014)) and acts as if multiple models were being combined in a single network. A really recent paper (as of 10 Aug 2018) extends the notion of dropout to a larger context, demonstrating that dropout can be seen as using a stochastic delta rule (Hanson, 1990) in which "each weight in the network is a random variable with mean

[1]The problem of constructing models robust to changes of domain and able to perform well (with out any training) in new domains (taken from Li et al. (2017))

[2]See animations demonstrating how traditional DRL models are unable to cope with minuscule changes in the environment: `https://www.vicarious.com/2017/08/07/general-game-playing-with-schema-networks/`

$\mu_{w_{ij}}$ and standard deviation $\sigma_{w_{ij}}$" (taken from Frazier-Logue and Hanson (2018)). This approach creates a distribution over multiple networks with shared weights that can be seen as a efficient model averaging (a Bayesian network).

In this project we created a single model which explicitly defined domain/ task specific sub branches. Considering the parameters as distributions could be a promising direction to naturally mix into a single model what can me learnt in multiple tasks, sharing knowledge across tasks through interactions of the parameter distributions, and preventing overfiting (a possible reason explaining the poor DRL model robustness). Though potentially useful, using parameters as random variables might lead to extreme difficulties when training with non-stationary data such as that used in RL (the sole use of dropout has proven to be difficult in RL Gal et al. (2017)).

A final word on alternatives approaches to RL. The careful reward function design that we had to perform for the MTL experiments highlights one of the main weaknesses of RL - it assumes the existence of a reward function, which needs to be carefully human-engineered. Some times creating such reward functions is a really difficult task and for open ended environments it is not always obvious what you want to maximize or how to design this function . This highlights how important it is to explore alternative approaches to RL for *learning from experience*, such as those agents driven by curiosity (seeking cognitive dissonance, Forestier and Oudeyer (2016)), intrinsically motivated agents (were the reward is based on the amount of uncertainty or surprise, maximizing entropy, Gottlieb et al. (2013)), and other developmental learning methods which are partially inspired by infant development (Oudeyer, 2014).

Bibliography

Alvarez-Charris, D. C. (2018a). Deep multitask reinforcement learning for continuous control tasks.

Alvarez-Charris, D. C. (2018b). Nip coursework: Review - progressive neural networks.

Bengio, Y. et al. (2009). Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127.

Fernando, C., Banarse, D., Blundell, C., Zwols, Y., Ha, D., Rusu, A. A., Pritzel, A., and Wierstra, D. (2017). Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*.

Forestier, S. and Oudeyer, P.-Y. (2016). Curiosity-driven development of tool use precursors: a computational model. In *38th annual conference of the cognitive science society (cogsci 2016)*, pages 1859–1864.

Frazier-Logue, N. and Hanson, S. J. (2018). Dropout is a special case of the stochastic delta rule: faster and more accurate deep learning. *arXiv preprint arXiv:1808.03578*.

Gal, Y., Hron, J., and Kendall, A. (2017). Concrete dropout. In *Advances in Neural Information Processing Systems*, pages 3581–3590.

Geramifard, A., Walsh, T. J., Tellex, S., Chowdhary, G., Roy, N., How, J. P., et al. (2013). A tutorial on linear function approximators for dynamic programming and reinforcement learning. *Foundations and Trends® in Machine Learning*, 6(4):375–451.

Gottlieb, J., Oudeyer, P.-Y., Lopes, M., and Baranes, A. (2013). Information-seeking, curiosity, and attention: computational and neural mechanisms. *Trends in cognitive sciences*, 17(11):585–593.

Hanson, S. J. (1990). A stochastic version of the delta rule. *Physica D: Nonlinear Phenomena*, 42(1-3):265–272.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. (2018). Deep reinforcement learning that matters. *CoRR*, abs/1709.06560.

Ichter, B. and Pavone, M. (2018). Robot motion planning in learned latent spaces. *arXiv preprint arXiv:1807.10366*.

Kansky, K., Silver, T., and A., D. (2017). Schema networks: Zero-shot transfer with a generative causal model of intuitive physics. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1809–1818, International Convention Centre, Sydney, Australia. PMLR.

Klambauer, G., Unterthiner, T., Mayr, A., and Hochreiter, S. (2017). Self-normalizing neural networks. In *Advances in Neural Information Processing Systems*, pages 971–980.

Levine, S., Finn, C., Darrell, T., and Abbeel, P. (2016). End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373.

Li, D., Yang, Y., Song, Y.-Z., and Hospedales, T. M. (2017). Learning to generalize: Meta-learning for domain generalization. *arXiv preprint arXiv:1710.03463*.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Marcin Andrychowicz, Bowen Baker, M. C. (2018). Learning dexterous in-hand manipulation. *arXiv preprint arXiv:1808.00177*.

McCloskey, M. and Cohen, N. J. (1989). Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier.

Misra, I., Shrivastava, A., Gupta, A., and Hebert, M. (2016). Cross-stitch networks for multi-task learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3994–4003.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529.

Nachum, O., Norouzi, M., Tucker, G., and Schuurmans, D. (2018). Smoothed action value functions for learning gaussian policies. In *ICML*.

Ng, A. (2018). Cs229 - machine learning.

Oudeyer, P.-Y. (2014). Developmental learning of sensorimotor models for control in robotics. *SIAM News*, 47(7).

Peng, X. B., Abbeel, P., Levine, S., and van de Panne, M. (2018). Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. *arXiv preprint arXiv:1804.02717*.

Press, W., Teukolsky, S., Vetterling, W., and Flannery, B. (1992). Statistical description of data. *Numerical Recipes in C: The Art of Scientific Computing*, page 636.

Pyeatt, L. D., Howe, A. E., et al. (2001). Decision tree function approximation in reinforcement learning. In *Proceedings of the third international symposium on adaptive systems: evolutionary computation and probabilistic graphical models*, volume 2, pages 70–77.

Rusu, A. A., Rabinowitz, N. C., Desjardins, G., Soyer, H., Kirkpatrick, J., Kavukcuoglu, K., Pascanu, R., and Hadsell, R. (2016). Progressive neural networks. *arXiv preprint arXiv:1606.04671*.

Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. (2015). Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2016). High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

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

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958.

Sutton, R. S., Barto, A. G., et al. (1998). *Reinforcement learning: An introduction*. MIT press.

Weng, L. (2018). Policy gradient algorithms.

Westervelt, E. R., Chevallereau, C., Choi, J. H., Morris, B., and Grizzle, J. W. (2007). *Feedback control of dynamic bipedal robot locomotion*. CRC press.

Wu, Y., Mansimov, E., Grosse, R. B., Liao, S., and Ba, J. (2017). Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. In *Advances in neural information processing systems*, pages 5279–5288.

Yang, Y. and Hospedales, T. (2015). *A Unified Perspective on Multi-Domain and Multi-Task Learning*.

Yang, Y. and Hospedales, T. (2017). *Deep Multi-task Representation Learning: A Tensor Factorisation Approach*.

Yang, Y. and Hospedales, T. M. (2016). Trace norm regularised deep multi-task learning. *arXiv preprint arXiv:1606.04038*.

Yu, W., Turk, G., and Liu, C. K. (2017). Multi-task learning with gradient guided policy specialization. *arXiv preprint arXiv:1709.07979*.

Zhang, Y. and Yang, Q. (2017). A survey on multi-task learning. *arXiv preprint arXiv:1707.08114*.

Appendix A

Complete Learning Curves

A.1 Complete MDL (Bipedal Walker) Depth Experiments

In **fig.** A.1 the complete learning curves for all the 16000 Episodes of MDL experiments done on the Bipedal Walker scenario are shown.

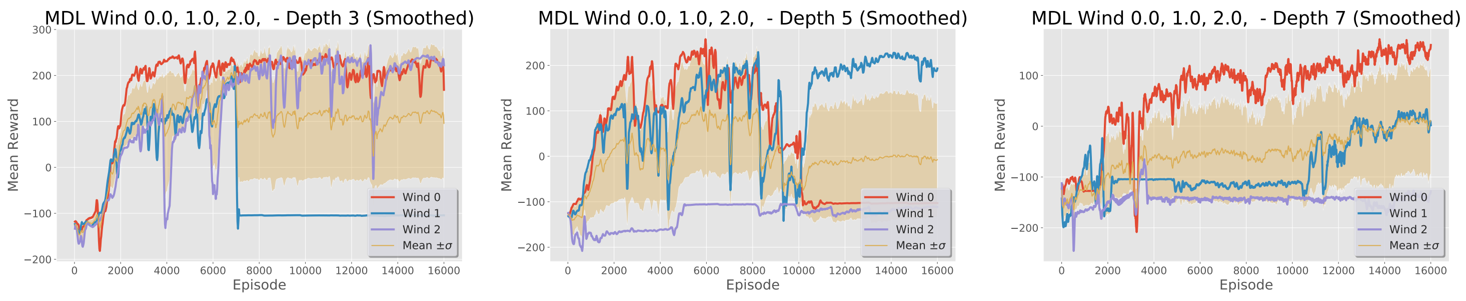

Figure A.1: *Complete Multi Domain Learning experiments for networks depth 3, 5 and 7. Yellow line indicates the average performance across all task and the shaded area indicates its standard deviation.*

MDL agents become increasingly unstable as more and more experience is fed into the networks . The instability point is (except the 7 hidden layers network) usually after 7000 or 8500 episodes.

A.2 Complete MTL (Lunar Lander) Learning Curves

In **fig.** A.2 the complete learning curves for all the task solved by the MTL agent in the Lunar Lander scenario are shown. It can be seen that unlike the MDL learning, MTL does *not* become unstable as more experience are fed into the multi headed network.

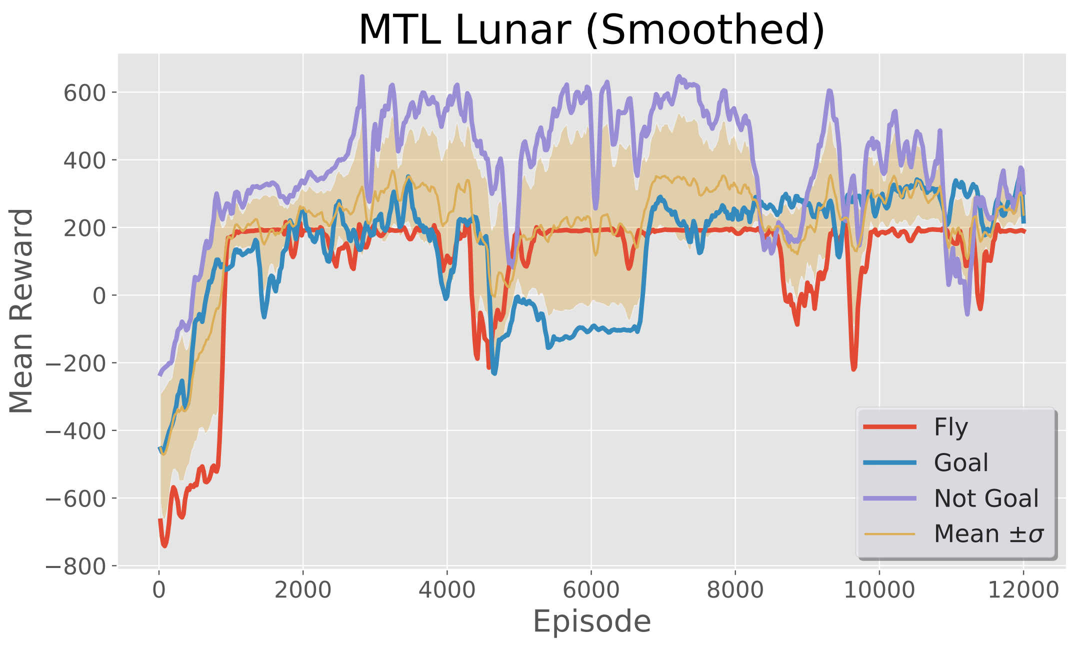

Figure A.2: *Complete Multi Task Learning curves. The plots highlight how MTL in Lunar Lander is much for stable than the experiments we did in MDL for the Bipedal Walker scenari.*