Reinforcement Learning in the Real World: Strategies for Computing Resource Allocation and Simulation to Reality Conversion

RAFAEL CARVALHAES POSSAS

Supervisor: Prof. Fabio Ramos

A thesis submitted in fulfilment of
the requirements for the degree of
Doctor of Philosophy

Faculty of Engineering
The University of Sydney
Australia

August 2022

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the University or other institute of higher learning, except where due acknowledgement has been made in the text.

Rafael Carvalhaes Possas

August 2022

Abstract

Recent advances in machine learning and robotics are automating several processes in the real world. For instance, robots are now able to solve complicated tasks that until recently only humans were capable of doing. A specific branch of machine learning called reinforcement learning (RL), has shown remarkable results on learning tasks by merely allowing a controller to interact with the environment while provided with positive and negative reinforcement signals. Such methods, however, come with a high cost: the amount of data to train such behaviours can be prohibitive. One possible solution is to use simulators to collect the data but this this creates the "reality gap" problem where control policies initially trained on simulation do not transfer well when deployed to its target environment. In this context, this thesis addresses the problem of using RL in the real world by incorporating prior information into the training process that allows such methods to make better decisions when presented with real data.

As the first contribution, this thesis provides a method to learn energy-efficient policies where the learned behaviour is optimised for both accuracy and energy consumption. The method uses the signal collected in the real environment and decides whether to make decisions using a vision based or motion based sensor. The approach highlights the importance of considering the uncertainty of real-world processes when optimising for a specific resource. For instance, the system battery may have different discharge rates based on the temperature of the environment. This chapter serves as a motivation for the remaining of the work.

The second contribution of this thesis addresses the specific problem of minimising the Sim-to-Real gap. The proposed method incorporates prior information about the real world in order to find the most suitable simulation environment to train a RL policy. This is performed by using Bayesian Likelihood-Free Inference methods where our initial prior is refined as it is presented with real-world data. The framework allows for a more structured approach to the aforementioned problem as it incorporates the uncertainty of the real environment into the controller fine tuning process.

Lastly, this thesis connects simulation parameter inference with policy training. We present a method for simultaneously optimising the policy as the simulator continuously improves its accuracy in representing the real environment. The end-to-end approach significantly reduces the time required to learn a policy that has similar performance between simulation and real world. The framework highlights the

importance of treating simulator parameter inference and controller optimisation as a unified problem where both parts are equally important for the overall performance of the system.

Acknowledgements

This thesis is the result of years of work that could not be done without the support of many others that shared their knowledge and their experiences throughout the four years of the Ph. D program. First and foremost, I would like to thank my family that supported and encouraged me to take on this new Chapter. My wife Bruna has given me the support during difficult times and always believed in me. My parents have given me the foundation and all the invaluable lessons throughout my life. Lastly, my son Daniel who has given me the opportunity to share and teach a little bit of what I have learned with all these experiences. Prof. Fabio Ramos has been an excellent supervisor. His support and ideas guided me throughout the Ph. D. He has given me the opportunity to work with the most important people in my field of study. Moreover, his support and understanding during difficult times were of the utmost importance to keep me motivated and moving forward with my goals. I would also like to thank other people that I collaborated with in papers. Their input and combined effort helped me to understand that research is usually not done alone, but as a result of different people with different backgrounds working together.

Lastly, I would like to thank god for opening up this opportunity in my life and enabling me during the course of the program.

CONTENTS

List of Figures

List of Tables

Nomenclature

Basic fonts

a	a scalar
$\mathbf{a}$	a vector
$\mathbf{A}$	a matrix
$\mathcal{A}$	a set

Matrix operations

$[\mathbf{A}]_{ij}$	the element in the i'th row and j'th column of $\mathbf{A}$
$\mathbf{A}^\mathsf{T}$	transpose
$\mathbf{A}^{-1}$	inverse
$\mathrm{tr}(\mathbf{A})$	trace

Basic operations

$\mathcal{U} \backslash \mathcal{V}$	the complement of set $\mathcal{V}$ in $\mathcal{U}$
$\langle f, g \rangle_\mathcal{F}$	the inner product between $f, g \in \mathcal{F}$
$\limsup_{n \to \infty} a_n$	the limit superior of a real-valued sequence, i.e. $\limsup_{n\to\infty} a_n := \lim_{n\to\infty} \left(\sup_{t \geq n} a_t\right)$
$\|f\|_\mathcal{F}$	the norm of $f \in \mathcal{F}$
$\overline{\mathcal{U}}$	the closure of the set $\mathcal{U}$, i.e. the smallest closed set containing $\mathcal{U}$.
$\mathbf{x} \cdot \mathbf{y}$	the dot product, i.e. the standard inner product in Euclidean vector spaces
$f(\mathcal{U})$	The image of set $\mathcal{U}$ under the mapping f
$f^{-1}(\mathcal{V})$	the pre-image of set $\mathcal{V}$ under the mapping f

Special symbols

$\mathcal{D}_n$	a dataset containing n entries
$\varnothing$	the empty set
$\mathbb{E}$	expected value
$\mathbb{1}_\mathcal{A}(\cdot)$	the indicator function of a set $\mathcal{A}$
$\mathbb{N}$	the set of natural numbers, i.e. the positive integers, $\mathbb{N} = \{1, 2, 3, \dots\}$
$N(\hat{\mathbf{x}}, \mathbf{\Sigma})$	a normal distribution, i.e. a Gaussian probability measure, with mean $\hat{\mathbf{x}}$ and covariance $\mathbf{\Sigma}$
P	a probability measure

$\mathbb{P}\{\cdot\}$	probability of an event involving random variables, e.g. for a random variable ξ defined on a probability space $(\mathcal{X}, \mathfrak{A}, P)$ and valued in $(\mathcal{W}, \mathfrak{W})$, $\mathbb{P}\{\xi \in \mathcal{A}\} = P[\xi^{-1}(\mathcal{A})]$, $\forall \mathcal{A} \in \mathfrak{W}$.
$\mathbb{R}$	The field of real numbers, the real line
$\mathcal{O}$	big-O notation for the set of upper bounds valid up to a constant factor
$\mathbb{V}$	variance

CHAPTER 1

Introduction

The problem of automating tasks through the use of robotics is steadily increasing in popularity as the field of machine learning evolves. Before, robots had to deal with only a few sensors, nowadays they can make use of cameras, sound and mostly all kind of multi-modal sensor modalities data that human beings have to make decisions. For instance, the field of deep neural networks has made remarkable strides, outperforming previous state-of-the-art in various disciplines of machine learning and also allowing the use of unstructured data like images, audio, text and etc. This allows robots to reason with data that is largely available in our world. Such a tremendous leap motivated several new developments to the field of robotics; Simulators have been improved to take into account these new capabilities this new world of machine learning has brought.

1.1 Motivation

Several modern issues are demanding increasing amounts of data: Spacecrafts and robots have been sent to explore other planets in our solar system, autonomous trucks are performing dangerous tasks inside mines and increasing demands for energy and manufactured goods have been causing environmental issues such as air pollution, floods and etc [6].

The aforementioned challenges have been successfully tackled with the use of robots [7, 8, 9]. Machines can be programmed to perform tasks that have been deemed impossible to human beings. They are more accurate and more efficient than we have ever been. We can plan complex tasks in simulation, and only after a certain degree of confidence is reached, we can then deploy the robot in the real environment to get the task done. This has been made possible due to the advances in computational power and consequently the development of realistic computer simulations of reality.

Robotics applications have evolved considerably in the last few years due to the recent advances in machine learning. Machines nowadays are able to mimic human behaviour by learning tasks through experimentation. Given enough time and the correct reward function, robots are able to find optimal solutions for non-trivial tasks. For instance, robots are able to slide objects to a specifc target [10] without any interference from humans. All of this happened due to the recent leaps in Reinforcement Learning and Deep Learning. Models for image processing are not only extremely accurate but also lightweight [11] to be deployed in a small Robot have widened the possibilities for deploying such machines in the real world. Challenges in robotics provide inspiration, impact and validation for new developments in Reinforcement Learning and Deep Learning. These frameworks offer a set of tools that enables a robot to autonomously discover an optimal behavior without any prior knowledge of the environment. Through a series of trial-and-error interactions, algorithms are able to find optimal policies that work remarkably well in real life applications such as the ones in robotics. Such techniques, however, often not only require tremendous amounts of data but also expects an approximately uniform distribution to perform well at a task. In practice, data distribution can go through various changes (e.g. seeing a new class of data) when working with long streams of data, causing learning problems/environment to become *non-stationary*.

A known problem that presents some level of difficulty even to the most skilled human beings is, for instance, solving a Rubik's cube. This is a 3D Combination puzzle invented in 1974. The cube is covered with multicoloured squares and the player attempts to twist and turn so that all the squares on each face are of the same colour. Solving such mathematical problem using robotics can lead the way to much more complex tasks. In this way, researchers have been using such problem as a training ground for developments in robotics and simulation. Training a robot in simulation as in figure 1.1 has several challenges. The work was purely focused on manipulation. Firstly, the grasp of the robotic hand needs to be able to deal with different types of friction coefficients of the cube that are dependent on the angle in which the cube is placed. Differences in scene lighting can cause the robot vision to under-perform hence providing noisy inputs to the control system. Lastly, the robot needs to be able to generalise to different cube materials, sizes and shapes, as long as the puzzle core goal remains the same. Such examples are all tasks that humans can deal without so much effort but robots struggle to do it. This happens as most robots are trained on a very specific task, without seeing any possible variation of the same problem. Finding better techniques for domain randomization allows robots to deal with such variances in the data and still achieve the desired goal even when input distribution changes.

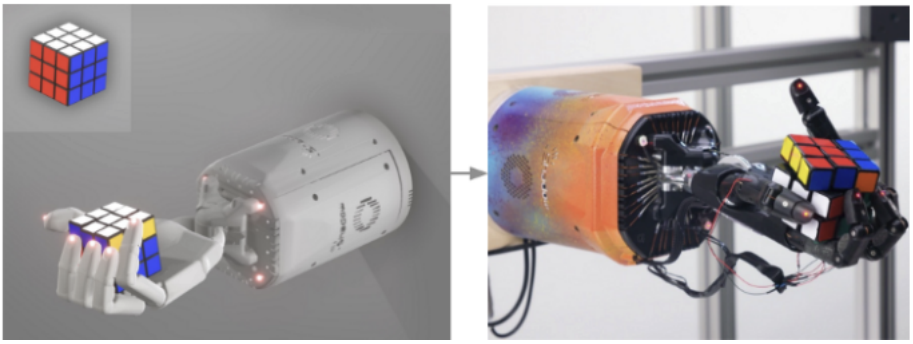


FIGURE 1.1: Training a robot in simulation to solve the real Rubik's cube puzzle.

Another problem is that of manipulating objects. Human beings are able to grasp objects of different sizes, colours and shapes in a matter of seconds. Robots, on the other hand, might fail in manipulating an object that was not available in their training set. As shown in figure 1.2, robots have to experience different types of configurations of objects in order to be able to generalise well on the real environment. This poses a fundamental problem: There are way too many combinations of configurations even for a small set of objects, its a very expensive tasks to allow training to be performed in all of them. Such examples are just a few of all the challenges that researches and practitioners in robotics are facing while trying to develop more intelligent machines.

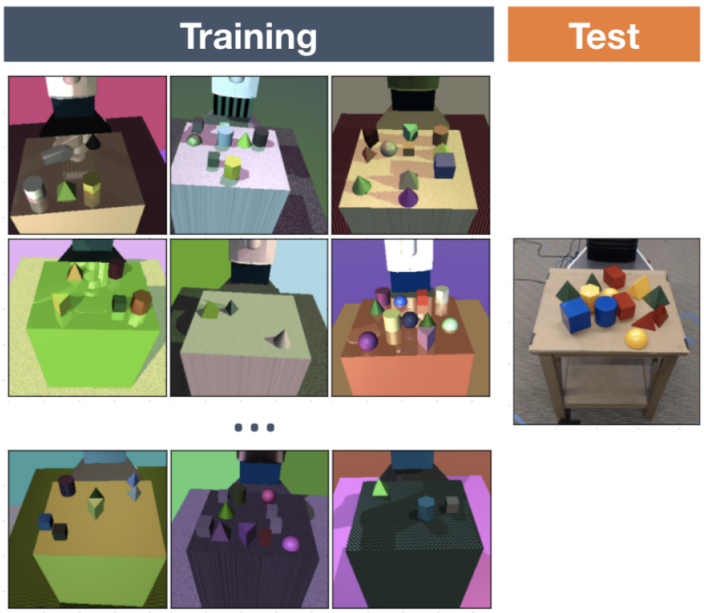


FIGURE 1.2: Images are randomised in the training environment [1].

The biggest challenge in robotics is to match a simulation configuration to the behaviour we observe in a real environment. The advance in this area has allowed such programs to become photo-realistic and also to implement advanced physics engines. However, they are still full of parameters to configure, and finding the set that best mimics reality is still a huge challenge. The difference between what a robot sees in the simulation environment to that of what it sees in the real world is known as Sim-To Real Gap. This issue has held back lots of developments in the field even though we had several advances in control and vision algorithms.

As it has been mentioned above, data generated in the real world can be non-stationary at times. It is well known that problems with physical components are inherently uncertain, and therefore, it is not only important to find the right parameter for simulators but to also measure the uncertainty around that estimate. Sim-To-Real gap has been solved in several cases by randomly sampling several different random simulations and training policies on all of them. The main problem of this approach is that several of the parameters combination are not useful, as they are impossible to happen in our real-world due to constraints like the ones in physics laws. This approach generally works mostly for tasks where we are simulating the visual components of the simulation, but it fails when trying to use it on parameters of physics engines.

This thesis is motivated by bridging the simulation to reality gap while also measuring the uncertainty of our predictions. We propose methods that mix simulation training with real-world data collection to find distributions that correctly measures the uncertainty around parameters of a simulated environment. The main benefit is that it removes the 'guess' work around tuning control policies in robotics and allows practitioners to systematically improve the performance of their robotics systems.

Moreover, several sensors used in such robots are powered by limited energy supply such as batteries. Maximising the use of such batteries is of the utmost importance to make such systems more pervasive. We also propose methods to improve energy usage while providing a reasonable model performance trade-off.

1.2 Uncertainty Estimation

Its common that problems in robotics will present a very high degree of uncertainty in measurements. For instance, mobile robots often have to navigate with information only about their kinematic models rather than fully-estimated dynamics. In order to achieve a more comprehensive and useful model, one

would require to also obtain estimates of properties of the environment in which the robot was inserted. Properties of the terrain ahead, inclination and shape are some of such variables. In this way, a control algorithm will have to deal with different sources of uncertainty when guiding the robot through an unknown environment.

In some problems, effects of action on the current state of the environment can be fully determined. For instance, when applying specific torques to a set of robotic arm joints, it is possible to determine quite accurately what would its final pose be. This all is represented well in simulation and might not benefit entirely from estimating the uncertainty. However, a simulated robotic task is almost never only about the robot. Simulators have to also create a realistic environment in which the robot is inserted, where all possible physical components are modelled through differential equations and their parameters estimated to match the real world.

Simulators have become increasingly more accurate in modelling the real world. However, it is not easy to translate for example a terrain friction to a single number. As it has been discussed above, this value can vary in a specific range, with a non-stationary standard deviation. Learning algorithms can use such distributions in order to learn control policies that are robust to changes in the real environment. This has become the cornerstone in robotics, and most of the state-of-the-art techniques nowadays do rely in uncertainty estimation to perform well.

1.3 Problem Statement

Despite all the advancements in simulation technology, many problems in robotics are still open. A few of these problems are considered in this thesis. Firstly, most robots operates in energy constrained environments, learning intelligent policies is just the first step of making robotics more pervasive, it is also necessary to consider energy consumption of such algorithms. Secondly, the sim-to-real gap have mostly been solved using random search or brute-force approaches, where no prior knowledge is taken into account when trying to create a robust control policy [12]. Such Domain Randomisation (DR) can be fast to compute if it exploits parallel computation but, on the other hand, its efficacy has been proven to work only on a handful of closed-loop robotics problems. Thirdly, in some scenarios, simulating all possible parameters might not make sense at all. For instance, the likelihood of some physical properties to assume specific values is very low. Current methods give equal probability to all values. Lastly, training a robot in simulation, is only the first step in achieving the desired performance

in a task. Several iterations are required to tune a control policy to be able to match the simulation behaviour in the real world. This is a long process that sometimes does not reach the desired result.

In summary, this thesis addresses the problem of: *Bridging the gap between simulation and real world using uncertainty estimation*. The main novelty comes from incorporating Bayesian techniques in a process that until now has been mostly deterministic.

1.4 Contributions

This thesis is composed of three contributing chapters, whose main contributions are summarised below.

1.4.1 Learning Energy Efficient Policies

Chapter 3 addresses the problem of learning policies that take into consideration real world constraints such as energy consumption. The chapter presents a Reinforcement Learning (RL) algorithm to select energy-efficient ways of predicting egocentric tasks using smartglasses. The policy actions are to choose between different sensor inputs such as vision and motion in order to provide the best trade-off between accuracy and energy consumption. This can be seen as the motivational work of this thesis, as estimating energy consumption of a battery, for example, depends on physical properties of the environment to determine the optimal discharge rate.

1.4.2 Likelihood-Free Inference for Simulation parameter estimation

Chapter 4 uses the motivation of the previous chapter to develop an algorithm that uses data collected from a real environment to better estimate distribution around simulation parameters. This chapter focuses, more specifically, in addressing the problem with point estimates of simulation parameters. It provides more informed decision making when choosing the range of parameters that better describes a physical environment. Several experiments are conducted to show the algorithm efficiency. The main caveat of this method, is that both policy and simulation parameter optimisation is performed separately.

1.4.3 End-to-End policy Optimisation and Simulation parameter estimation

The last chapter of this thesis puts together two important aspects of robotics learning: policy optimisation and simulator parameter discovery. The previous approach is combined in a single framework

where policy optimisation steps are alternated with simulation parameter discovery. Instead of training a policy until it reaches convergence, we perform only the necessary steps to be able to collect more data from the real world. In this way, the algorithm is able to save time by avoiding unnecessary policy training iterations which can be costly depending on the type of controller used.

1.5 Outline

This thesis' main components are a background chapter, followed by three contributing chapters and a conclusion. The three contributing chapters follow the same basic structure. After an initial motivating introduction, each chapter describes related work, methodology and experiments. These are outlined below.

Background presents the theoretical background needed to understand the main contribution of this thesis. Neural Networks and Deep learning and Reinforcement Learning are formally introduced and their interactions with the core methods of this thesis: Likelihood-free Inference.

Chapter 3 presents a method to find energy-efficient policies to be used in egocentric vision systems. This can be seen as the main motivation of pursuing the above direction of this work. We have found the aforementioned problems while conducting this work and then realised that this was a gap that needed further work.

Chapter 4 describes BayesSim where the goal is to find simulation parameters configuration that matches a real world environment along with their uncertainties. This is an offline algorithm, where policy training and simulation inference are performed over different stages.

Chapter 5 connects both simulation parameter inference and policy optimisation together in a end-to-end framework. Moreover, we extend the method to scale to higher dimensional data problems using deep learning techniques.

CHAPTER 2

Background

This chapter presents a review of concepts that form the basis for the methods this thesis proposes. Most of the problems faced by this work deal with the gap between the data collected in a simulation environment and the data collected in real life. They have significant differences that are hard to estimate and/or provide a closed form solution of the system. In this case, a logical choice of approach is to apply likelihood-free inference methods as we will see in Chapter 3. To form a basis for this approach, the chapter begins by reviewing some of the basic concepts of neural networks in machine learning. In particular, we show how neural networks can be used to estimate parameters of a mixture of Gaussians using the MDN Networks [13]. Next we review the principles of Reinforcement Learning and how they have been used in several robotics tasks to learn useful controllers. Lastly we review recent literature in likelihood-free inference and how to combine such methods with the ones from previous sessions to provide a framework to bridge the sim-to-real gap in robotics

2.1 Supervised Learning

We begin this chapter by considering the case of supervised learning [14]. Given a dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ comprised of n input-output pairs $(\mathbf{x}_i, y_i)$ where outputs are real-valued scalars $y \in \mathbb{R}$. The main task in this scenario is to learn a function f mapping each input $\mathbf{x}_i$ to the corresponding output y_i, so that we can predict the function value at unobserved inputs. Generalizing over inputs, we can assume they are d-dimensional vectors from a set $\mathcal{X} \subset \mathbb{R}^d$. Such values are often noisy, so that:

$$y_i = f(\mathbf{x}_i) + \phi_i \, , i = 1, \ldots, n \, ,$$

where ϕ_i is independent and identically distributed (i.i.d.) random noise drawn from a given probability distribution. Typically, it is considered that the noise follows a zero-mean Gaussian distribution with variance σ_ϕ, i.e. $\phi_i \sim N(0, \sigma_\phi^2)$.

In order to find a solution to the above problem we try to approximate the function f until it best explains the data of the dataset $\mathcal{D}$. For instance, in a linear regression case we can assume that:

$$f(\mathbf{x}) = \mathbf{w}^\mathsf{T}\mathbf{x}$$

where the vector $\mathbf{w}$ composes the set of parameters of our model. The main task in supervised learning is to find parameters $\mathbf{w}$ that best explains the observed data. Generally, this is formulated as minimising a data-dependent loss function, choosing the $\mathbf{w}$ that provides the most similarity with the given dataset. The above example is limited as it only fits data with linear relationships.

Another usage for supervised learning is in classification, in this case $y \in \{-1, +1\}$. This method is known as logistic regression [14] and instead of fitting the best fit line, it consdenses the output of the linear function beteween 0 and 1. The function we are optimising is non-linear, given by:

$$p = \frac{1}{1 + e^{(b+w_1 x)}}$$

where p is the probability of of the input belonging to a class. The logistic regression is a binary classification model, that concentrates its outputs between between 0 and 1. This type of non-linearity was highly used to develop neural networks, a much more powerful class of methods.

2.2 Optimisation

In this section we discuss the seminal methods for performing model optimisation. Such techniques are the cornerstone of any machine learning model, as they allow models to learn through data. More formally, we describe gradient descent, its variation for larger models - Stochastic gradient descent and finally we discuss better ways of controlling the update steps through Momentum and ADAM.

In Machine Learning, most algorithms rely on specific techniques to "learn" the correct parameters of a desired function. In this context, when we optimise a function w.r.t parameters θ, we are learning the optimal values that minimise a pre-defined loss-function. Several optimisation techniques will be discussed in this chapter, and they are the cornerstone of ML algorithms, as they allow such methods to search for parameters values in a multi-dimensional space.

2.2.1 Gradient Descent

Most machine learning algorithms rely on optimisations of weights so as to find the function that best fits a given dataset. In order to calculate these values one should choose a cost function that measures the variance between the actual result and the desired output on each iteration of training. The foundational method for all types of optimisation is known as Gradient Descent [2]. This is a method for finding global/local minimum/maximum of functions. The main intuition behind this technique is that it iteratively calculates a gradient for the weights/biases of a model and updates it from the current value. This updates can either push to the maximum or minimum of a function. As for the latter, it can be seen as pushing a ball through the valley as depicted in figure 2.1. Gradient Descent is an iterative algorithm

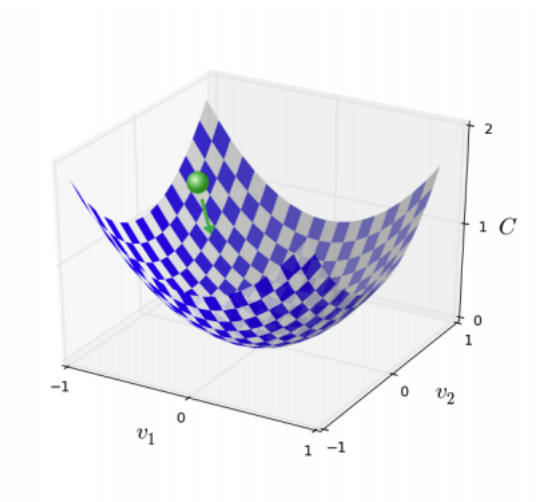


FIGURE 2.1: Gradient Descent can be seen as pushing the ball down the valley [2]

where the step size determines how much we go in the direction the function is being optimised. More formally its calculation can be performed as:

$$\theta = \theta - \alpha \frac{1}{N} \sum_{i=1}^{N} \frac{\partial}{\partial \theta} \mathcal{L}(\mathbf{x_i}, y_i)$$

where α is the step size of the learning rate and θ is usually the weights of the model we are optimising and $\mathcal{L}$ is the loss function. Its worth to note that such calculations are done for all data points in the dataset, as we will see, this become unfeasible for models with million of parameters such as deep neural networks. Instead calculations should be performed on batchs of the data to speed up calculations.

2.2.2 Stochastic Gradient Descent

Gradient Descent is a simple first order optimisation method and is heavily used in Deep Learning. Its success comes mainly from being computationally inexpensive and empirically it obtains good performance. In models like the ones found in Deep Learning the number of samples of datasets is in the hundreds thousands, it not millions. Evaluating the gradient $\nabla_\theta \mathcal{L}(\mathbf{x_i}, y_i)$ becomes very expensive and it is often impossible to apply the calculation above as the dataset would not fit in memory to compute the true gradients. Stochastic Gradient Descent (SGD) it the preferred method to train any model where the amount of data and parameters is relatively high. It uses an approximation of the true gradient computed using only a randomly sampled mini-batch of data at a time,

$$\theta = \theta - \alpha \frac{1}{p} \sum_{i=1}^{N} \frac{\partial}{\partial \theta} \mathcal{L}(\mathbf{x_i}, y_i)$$

where p is the mini-batch size of the sample and N the total size of dataset. SGD is proven to converge under mild conditions [15], when the following condition is satisfied,

$$\sum_t \alpha^2 < \infty \; and \; \sum_t \alpha = \infty.$$

Following a learning rate that satisfies the above condition will generally lead to slow convergence. Hence methods to approximate the step size have emerged such as RMSProp and ADAM [16], as mentioned before. These methods automatically tune the step size during the optimisation process.

2.2.3 Momentum

Finding the optimal learning rate for gradient descent is always a challenge, as at each point of the optimisation process this number could be different. Momentum [17] is a method that is used along with Stochastic Gradient Descent and provides resiliency to the optimisation process since it helps to avoid convergence to local minimum which can be seen as a minor "bump" in the loss function surface and the chosen step size was too small. In figure 2.2 we can see that the learning rate reduces as we approximate the minimum of a function. The updates accumulate "momentum" from previous step while at the same time the learning rate is reduced. SGD with momentum can be formulated as a two step process. First we compute the running exponentional average of the gradient

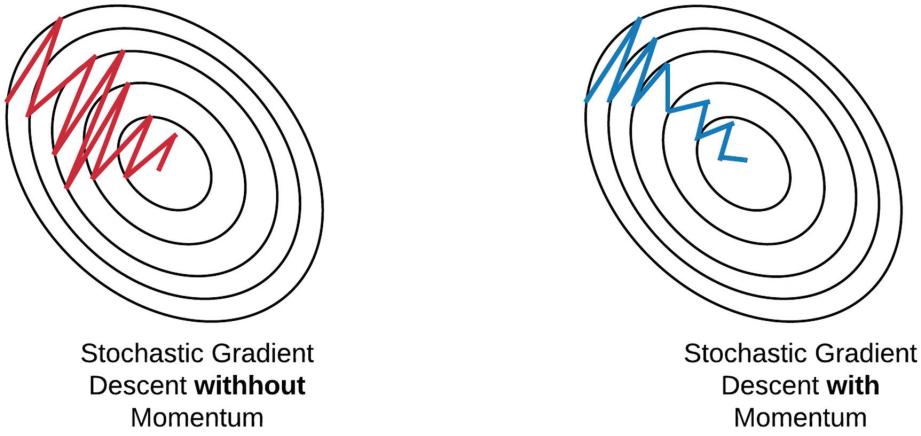


FIGURE 2.2: Comparison of optimisation process with and without momentum.

$$m_t = \beta m_{t-1} + (1 - \beta)\nabla_\theta \mathcal{L}(\theta_{t-1})$$

where m_t is the momentum at time-step t and by iteratively applying this equation until the change $|\theta_t - \theta_{t-1}| \leq \epsilon$, we find the set of parameters which optimise our objective function. The biggest advantage of this method is that its an adaptive learning rate where choosing the initial learning rate have less effect in the final result, as it will calibrate accordingly once updates are performed.

2.2.4 Adam

A method that explores further the integration between SGD and Momentum is Adaptive Moment Estimation (Adam) [18]. The addition of momentum to SGD allows previous gradients to create a momentum effect on the current optimisation step by accumulating the first order moments (the gradient values) of past steps and averaging those out with the current gradient. In this way, a current push of the ball down the slope does not depend solely on the slope it is currently on, but on the entire section the ball came rolling down from. This length is represented by the β parameter in the momentum calculation. Adam takes this idea one step further and incorporates also second order moments (the variance) into the optimisation step. This allows optimisation steps to be taken with more certainty as it helps steps to have consistent direction, which means they have lower variance. On the other hand, having different directions at every step, indicates low certainty and high variance; thus we would want to take smaller, more conservative steps.

The aforementioned ideas are incorporated by Adam. The variance information is added to the optimisation process, and by doing this, every parameter in θ is adaptively updated according to the notion just described. Adding second order moments, makes Adam a more sophisticated approach that improves significantly over the momentum idea. The iterative process of Adam can be broken down to mainly three steps. Firstly, the first and second order exponential average momentum terms are calculated:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla_\theta \mathcal{L}(\theta_{t-1})$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_\theta \mathcal{L}(\theta_{t-1}))^2$$

where this can be understood as an exponential running average, where bias is accumulated towards the earlier steps momentum terms. Secondly, this bias is correct for both momentum terms as follows:

$$m'_t = \frac{m_t}{1 - \beta_1^t}$$
$$v'_t = \frac{v_t}{1 - \beta_2^t}$$

finally, the parameter can be updated as

$$\theta_t = \theta_{t-1} - \alpha \frac{m'_t}{v'_t + \epsilon}$$

where α is the step size, β_1 is the first order momentum accumulation parameter and β_2 the second order momentum accumulation. The effect of this operation is constant normalisation of the gradient step size w.r.t to the current running variance. In other words, one can imagine the running variance value v'_t as constructing a trust region in which the size of the step is allowed to be. Moreover, with all the this new terms, the role of the original step-size α has now slightly changed, it now indicates how large is the step within the trust region.

2.3 Neural Networks

There are many types of parametric and non-parametric models in machine learning; in this section we provide a technical understanding of one main model used in this thesis. Neural networks are a class of differentiable algorithms and were initially introduced by [19] in 1943. Their motivation was

to imitate the functionality of the human brain. However, since this conceptualisation progress slowed down, only a few years later, the first computational neural component inspired by how neurons in our brain spike was proposed by [20] and termed perceptron. The brain inspiration did not last long and slowly faded away, researchers stopped to try to imitated the human brain and progress was finally made again with backpropagation [21]. This allowed efficient optimisation of Neural Networks and, years later, Convolutional Neural Networks [22] allowed for efficient spatial processing of images.

One can think of Neural Networks as a set of perceptrons, a linear binary classifier, aligned into several layers having outputs from layer L mapped to inputs of the layers $L + 1$. The output of each perceptron is named activation and the set of activations in the final layer gives the desired classification output. For each connection, a neuron has one weight connecting with all neurons of the next/previous layer, and one bias. The goal is to find the values that minimise a given cost function.

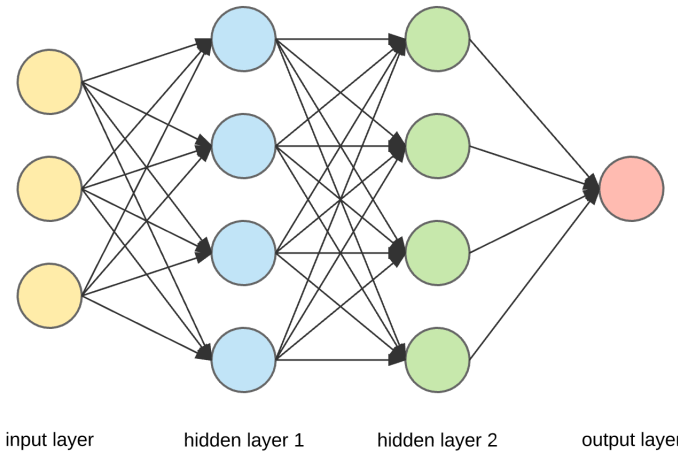


FIGURE 2.3: A simple neural network architecure. Multiple layers can be stacked together.

2.3.1 Network Layers

Given an input to a Neural Network, computation flows from left to right through different blocks, until it reaches the final output. Backpropagation computes the derivative of every parameter in the network so that Gradient Descent can be applied. Until now, we have abstracted the components that are used as computational blocks, and in this section the basic building blocks that a Neural Network is comprised will be discussed. Such blocks, can operate on the input as a whole, on certain parts of the input or perform point-wise operations (e.g. activation layers). We will show common ways these components are combined together.

Linear Layers are the major consumers of computational resources in Neural Networks and are also the ones with the higher amount of parameters exists. The name comes from the fact that these can at most model linear dependencies withing the data.

The fully connected layer is analytically defined as a matrix $W \in \mathbb{R}^{N \times M}$ summed with a bias $b \in \mathbb{R}^{N \times 1}$, and as such will have N(M+1) parameters. The operation of this layer for a given input vector $\mathbf{x} \in \mathbb{R}^{M \times 1}$ will produce an output vector $\mathbf{y} \in \mathbb{R}^{N \times 1}$ as such:

$$\mathbf{y}_{N \times 1} = W_{N \times M}\mathbf{x}_{M \times 1} + \mathbf{b}_{N \times 1} = \sum_{i=1}^{N} \mathbf{W}_{i,M}\mathbf{x}_{M \times 1} + \mathbf{b}_{i,1}$$

.

Activation Layers are important on Neural Networks to add non-linearities capabilities to our models and are placed after linear layers. Without such layers, Neural network models would be limited to modelling only linear dependencies for the given data. These are point-wise operations which apply a non-linear function on every value of the input which give Neural Networks the ability to bend the linear decision boundaries so that complex correlations within the data can be modelled with non-linear functions. Different from linear layers, they do not have parameters which get updated upon optimisation of a Neural Network. Usually, an activation function will not change the dimension of the data it operates on as it applies the function on every value independently. There are many types of activation functions, and research into different types is a broad topic in machine learning. In this section we will cover the activation layers that are used in this thesis, and thus limit the scope to a subset of the existing activation layers.

Logistic Function: is one of the first activation functions to be used in Neural Networks. They can be seen as a smooth replacement for the perceptron. The emulation of a firing neuron in the perceptron (fire and no-fire) has been inspired by the biological behaviour of our brain. Therefore, it is not a continuous function, and as such the gradients can often cause numerical difficulties as computation devices use a finite amount of bits when representing values. The Sigmoid was created with the main goal to circumvent this problem, it allows the on-off state behaviour of the Perceptron to be "softer" and, in this way, become differentiable while maintaining the monotonic property. The Sigmoid function $f(x)$ outputs values between (0,1) and is analatically written as:

$$f(x) = \frac{1}{1 + e^{-x}}$$

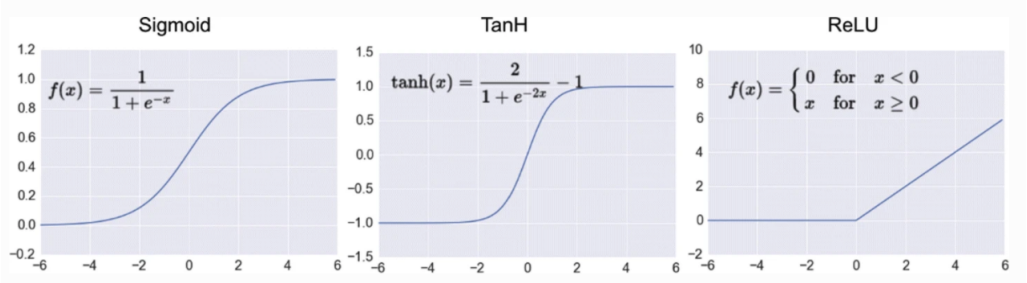


FIGURE 2.4: From left to right: Sigmoid, Tanh and ReLU functions

Tanh: The tanh is an alternative to the sigmoid function, and has similar properties such as differentiable anywhere and monotonic. It supports a wider range of outputs since his range is (-1, 1):

$$f(x) = \frac{2}{1 + e^{-2x}} - 1$$

ReLU: The Rectified linear unit (ReLU) [23] solves numerical problems in both sigmoid and tanh activation functions. The problem of "vanishing gradients" is that where gradient updates become smaller and smaller, until there is no further "information" passing through the activation function. It mainly happens due to the saturated regions in both of these functions, causing gradients to zero out in case the input value to the function is in those regions. This problem halts the operation of Gradient Descent as the value of the gradients is zero, and the parameters do not get updated. ReLU resolves this issue by formulating the function as seen in figure 2.4 where it adds hard threshold for calculating such values.

2.3.2 Backpropagation

Backpropagation is an application of the chain-rule from calculus which allows efficient optimisation of Neural Networks with many layers. As any differentiable machine learning algorithm, Neural Network models rely on optimisations of weights ω and biases β. In order to calculate these values one should choose a cost function that measures the variance between the actual result and the desired output on each optimisation step. The two methods for running such process on Neural Networks are known as Backpropagation and Gradient Descent. The main goal is to propagate small changes δ applied to any of the neurons of the network to all the neurons until the final layer.

Usually, the most basic types of NN, learn from data using convex optimisation techniques, that involves calculating derivatives of linear and polynomial cost functions. The learning algorithm finds weights and biases that best approximates the output $y(x)$. This is usually done by finding the global or local minimum of a chosen cost function, in other words, the algorithm should stop when the difference between the predicted and real values for all datapoints is the lowest. For each different task, a different cost function could be chosen and, ultimately, the use of partial derivatives with respect to the weights ω and biases β will provide the direction of the minimum of the cost function ∇C, namely gradients.

$$\nabla C \equiv \frac{\partial C}{\partial \omega}, \frac{\partial C}{\partial \beta}$$

Assuming a classification problem where the dataset is given by

$$\mathcal{D} = \{(\mathbf{x_1}, y_1), ..., (\mathbf{x_i}, y_i), ..., (\mathbf{x_n}, y_n)\}$$

where $\mathbf{x_i}$ has some dimensionality x_{dim} and y_i is the scalar label output. The Neural Network will predict $\hat{y_i}$ for each input $\mathbf{x_i}$ performing the following process; Firstly, the hidden layer activations are computed as,

$$h_i = \sigma(\omega_h \mathbf{x_i} + \beta)$$

where $\omega_h x \in \mathbb{R}^{h_{dim} \times x_{dim}}$, x_{dim} is the dimensionality of an input, h_{dim} is the number of hidden neurons and $\beta_x \in \mathbb{R}^{h_{dim}}$ is the bias of the input layer. Non-linearity in NN is provided through a sigmoid function, defined as $\sigma(x) = \frac{1}{1+exp^{-x}}$. Given a Neural Network with one single layer h, the output $\hat{y_i}$ can be calculated as,

$$\hat{y_i} = \sigma(\omega_{yh} h_i + \beta_h)$$

where $\omega_{yh} \in \mathbb{R}^{h_{dim}}$, and $\beta_h \in \mathbb{R}$ is the hidden layer bias. For each of the calculated neural network output, we use a log loss function calculated as,

$$\mathcal{L}(\mathbf{x_i}, y_i) = -(y_i log(\hat{y_i}) + (1 - y_i) log(1 - \hat{y_i}))$$

Lastly, in order to update the parameters of the network, one can compute the gradient of the parameters w.r.t loss,

$$\theta = \theta - \alpha \frac{1}{N} \sum_{i=1}^{N} \frac{\partial}{\partial \theta} \mathcal{L}(\mathbf{x_i}, y_i)$$

where $\theta = \{\omega_{hx}, \omega_{yh}, \beta_x, \beta_h\}$ and α is the learning rate or the step size. Before deep learning, neural networks were relatively small and used for datasets of relative low dimensionality. This allowed for optimisation using full batch gradient descent, meaning the entire data was used to compute a single step of the gradient descent for updating the weights of the network.

2.4 Convolutional Neural Networks

Deep Learning is a field within Neural Networks where the architectures are made of many hidden layers. When following this approach, the main barrier was to scale gradient calculations throughout all multiplication operations. For instance, the problem of vanishing/exploding gradients would cause numerical instability to the calculations, as update values would be either too big or too small, causing the network to respectively become unstable or stop learning at all. This is caused by the chained computation where the gradient of the lowest layer is computed using the weight values of all the layers above it. For instance, if the weight values of the layers above are very small, the gradients of the lowest layer will be exponentially smaller. The popularity of deep neural networks has risen with the developments of methods that were able to alleviate such problems. The introduction of an unsupervised pre-training phase helps to find a good initialisation of weights and biases. Later, better initialization techniques such as Xavier Initialization [24], and optimisers such as RMSProp [16] and Adam [18] enabled deep networks to eliminate the need of pre-training phase and directly start training.

It has been shown [2], that, generally, deep networks can have better results than shallow neural networks. This is due to the fact that layers farther from the input layer are able to learn complex features by incrementally building on the features found in the previous layers. In machine learning, the performance of a model is highly dependent on the features used to solve the task, the ability to engineer a broad spectrum of features allow deep networks to complete the task successfully. Another advantage is that such methods are able to engineer task-specific features using only the raw data provided, this can not only save time but provide better accuracy.

Nowadays, deep networks are heavily used in machine learning to solve a wide range of tasks in fields such as object recognition [11], activity recognition [25], and pose estimation [26]. Also their use can be seen within robotics and natural language process, for instance, in manipulation tasks [12] or speech recognition [3].

Convolutional Neural Networks (CNN) are a class of Deep Neural Networks that have shown to perform remarkably well in computer vision tasks such as object classification [11], object detection [27] and scene segmentation [28] and many others. The main reason for success of such methods is that they are efficient in exploiting information in the data while preserving their spatial structure; this is important in unstructured data scenarios such as images. On the other hand, fully-connected neural networks require the two-dimensional input to be unwrapped to a one dimensional input, losing its spatial structure completely. The name of this architecture comes from the fact that it uses convolutional layers for extracting features. The general architecture is made of convolutional layers, pooling layers and a set of fully-connected layers with softmax/classification on top. A convolutional layer takes a three-

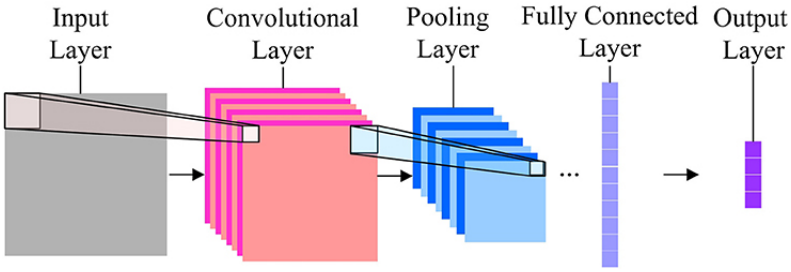


FIGURE 2.5: Convolution Neural Networks stacks convolutional layers to detect features and pooling layers to reduce image sizes at each step. A classification layer is added on top. [3]

dimensional input with fixed height, width and depth (usually referred as channels), it uses a convolution filter that is often smaller than the image itself. This filter is applied as a patch to the image, where it multiplies the filters values by the image values on the current patch. This is done until the filter covers the entire image, and the resulting scalar values are known as the convolutional features. More specifically, l^{th} convolution layer, takes an input h^{l-1} with a fixed width and height and d^{l-1} channel depth where h^0 is the input image $\mathbf{x}_i$ from the dataset $\mathcal{D}$, where h^0 is loosely used to refer to $h^0(\mathbf{x}_i)$. Then the l^{th} convolution layer convolves the input with d^l kernels $K_1^l, ..., K_n^l$ with a fixed stride to produce $h^l, ..., h_n^l$ where,

$$h_n^l = Relu(h_n^l - 1 * K_n^l + b_n^l)$$

where * denotes convolution operation and $Relu(x) = max(0, x)$ is the rectified linear unit activation and b_i^l is the bias.

Another important layer type is the Pooling Layers. They are designed to reduce the dimensionality of the data as it flows through the network. Applying the pooling operation to inputs is what makes the CNN translation invariant. For instance, the Max Pooling operation moves a kernel over the image and at each location the pooling operation takes the maximum value of the overlapping patch of the input. The l^{th} max-pooling layer takes an input of fixed width and height and a depth of d^{l-1}, and subsamples the input by sliding d^{l-1} kernels with kernel size z^l and stride s^l to produce $h_n^{l,max}$.

Finally, a set of fully-connected layers follows the convolution and pooling layers. At first, the fully-connected layer unwraps the ouput of the last convolution/pooling block to a one dimensional vector. This vector is then propagated through the remaining layers as it would do in a traditional architecture. The output of the final fully-connected layer is fed into a classifier (e.g. softmax) or a regressor (e.g. linear regression) to make a prediction. Kernel parameters and fully connected weights are optimised using Backpropagation [17] and the preferred stochastic gradient method. This is done through minimising the discriminative error of inputs given by the following loss function for all convolutional layers up to C,

$$\mathcal{L}^C(\mathbf{x}_i, y_i) = -\sum_{j=1}^{C}(y^j \; log \; \hat{y}_i^j + (1 - y_i^j) \; log(1 - \hat{y}_i^j)).$$

2.5 Recurrent Neural Networks

Recurrent Neural networks are another special type of neural networks, these are frequently used in problems involving sequential data. While Traditional Neural Networks and Convolutional Neural Networks takes in a fixed size vector as input, RNNs are designed to take a series of input with no predetermined limit on size.

Such methods are designed to remember past data-points and the final decisions are influenced directly by what is has learnt from it. In other words, RNNs are designed to work on sequential data problems. The main difference in input/output is that RNNs can take one or more input vectors and produce one or more output vectors. The input is not only influenced by weights applied to inputs like a regular Neural Network but it also by a "hidden" state vector representing the context based on prior input/output. In

other words, the same input could produce a different output depending on previous inputs in the series. As per illustrated on figure 2.6, a vector $\mathbf{x_1}$ is combined with a hidden state $\mathbf{h_0}$ through weights $\mathbf{w_h}$ to generate an output y_1.

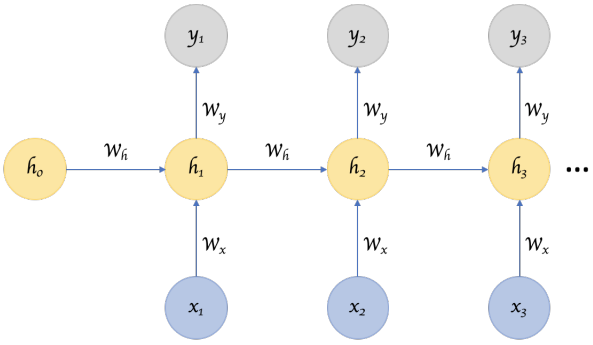


FIGURE 2.6: Inputs are influenced by both weights and hidden states.

The main difference in RNNs is how they calculate their hidden state $\mathbf{h_t}$. The hidden state is a function σ of the previous hidden state h_{t-1} and the current input. More precisely we can define this function with parameters θ as,

$$\mathbf{h_t} = \sigma(\mathbf{h_{(t-1)}}, \mathbf{x_t}; \boldsymbol{\theta})$$

Forward propagation can be then performed using the input to hidden weights $\mathbf{w^{hx}}$, hidden to hidden weights $\mathbf{w^{hh}}$ and finally hidden to output weights $\mathbf{w^{yh}}$ as,

$$\mathbf{h_t} = \sigma(\mathbf{w^{hh}h_{t-1}} + \mathbf{w^{hx}} + \mathbf{b^h})$$

where σ is an activation function in the same family of the ones used in a traditional neural network (e.g. Sigmoid, RELU, Tanh and etc). Each time step t can have their output calculated as,

$$\hat{y_t} = \mathbf{w^{yh}h_t}$$

where the predicted output $\hat{y_t}$ is not subject to any non-linearity. For specific use cases of RNNs, this can be changed, and a non-linearity specific to the problem can be applied.

The main caveat in such architecture is what is called 'Long-Term Dependencies'. For simple cases where the past information is 'not so distant', traditional RNNs can perform well. However, there are

cases where more context is needed and, therefore, the input depends further on results from a more distant past. The entire design of traditional RNNs is made in a way, that information of this distant past is easily erased in favour of more recent data. This gap is, fortunately, solved by Long Short Term Memory (LSTM) networks [29].

These are designed to directly solve the Long-Term Dependency problem. The key idea is introduced through a cell state, which run straight down the entire chain of neurons, with only some minor linear interactions. The goal is that information can just flow along unchanged. The Network, however, have the ability to remove or add information to the cell state, regulated by structures called gates. These can optionally let information through allowing them to keep information for longer periods of time.

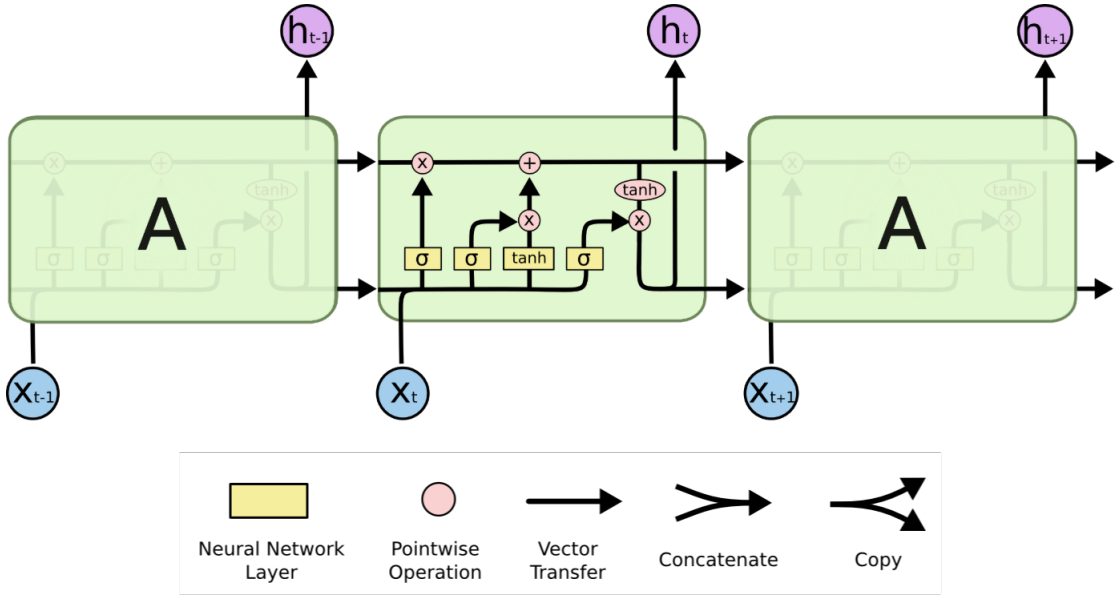


FIGURE 2.7: In a LSTM Cell, each line carries an entire vector from the output of one node to the input of the others.

The input of the LSTM needs to take into account information seen on previous data-points and is formally defined as,

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

where W is the weights, h_{t-1} the hidden state of previous timestep, x_t the current input and b_f the bias. The next step is to decide what new information the cell state is going to store,

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C$$

where i_t scales how much the cell state will be changed given the new value and $\tilde{C}_t$ is the current cell state. Its now time to update the previous cell state C_{t-1} value using the new calculated value

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Lastly, the output is decided using non-linearities in the output layer o like other Neural Network methods, and it is defined as:

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

2.6 Mixture-Density Networks

Mixture of Density networks are a class of generative neural networks developed in [13]. They have recently found a series of different applications in speech generation [30] and generation of artificial handwriting [31]. Many modern neural network architectures can be extended to become MDNs. They can be seen as an extension module, applicable to a broad variety of relevant tasks.

At its very core, the MDN concept is straightforward and appealing: Combine a deep neural network and a mixture of distributions. The network provides the parameters for multiple distributions, which are then mixed by some weights. These weights are also provided by the neural network. The resulting multimodal conditional probability distribution helps us to model complex patterns found in real-world data. We are, therefore, better able to evaluate, how likely certain values of our predictions are.

Generally, a Gaussian mixture is capable of modelling arbitrary probability densities if it is adequately parametrised. Formally, the conditional probability for a mixture is defined as,

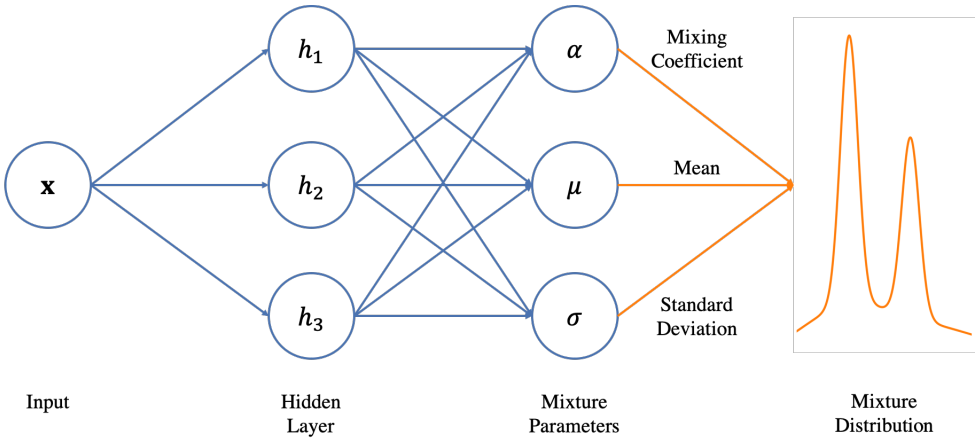


FIGURE 2.8: MDN Architecture in a single layer neural network. Distribution parameters are predicted at the final layer.

$$p(y|x) = \sum_{c=1}^{C} \alpha_c(\mathbf{x}) p(y|\lambda_{1,c}(\mathbf{x}), \lambda_{2,c}(\mathbf{x}), ...)$$

where c denotes the index of the corresponding mixture component. There are up to C mixture components (e.g distributions) per output, α denotes the mixing parameter, $\mathcal{D}$ is the corresponding distribution to be mixed and λ the parameters of the distribution $\mathcal{D}$ with λ_1 corresponding to the conditional mean $\mu(\mathbf{x})$ and λ_2 to the conditional standard deviation $\sigma(\mathbf{x})$ for the Gaussian mixture.

The formulation of the conditional probability as a mixture of distributions can already help to solve multiple problems. Firstly, the distribution can be arbitrary, as we are theoretically able to model every distribution as a mixture of Gaussians. Sencodly, using multiple distributions helps us to model multimodal signals. Thirdly, the standard deviation is now conditioned on the input, accounting for variability in the deviation. Lastly, the problem of the linearity of the function can be circumvented by choosing a non-linear model, which conditions the distribution parameter on the input.

2.7 Reinforcement Learning

Several applications of machine learning in real life requires labelled data for a specific problem. In RL, instead, the designer provides feedback through a reward function that measures the current performance of the agent. For instance, in the work of Muelling et al. [32], a robot is trained to return a table tennis ball over the net. In this case, observations are comprised of specific ball positions along with the internal joint positions and velocities of the robot. This captures the state of the system, and allows

the RL algorithm to learn optimal actions when visiting the same or similar states again. The actions available to the robot might be the torque sent to motors or the desired accelerations sent to an inverse dynamics control system. A RL learning problem is to find a policy that optimises long term rewards $R(s, a)$.

Reinforcement Learning is better understood by contrasting its problems with the ones in machine learning. For example, in supervised learning, an agent is presented with a sequence of examples of correct predictions while in Imitation Learning, an agent is provided demonstrations of actions of a good strategy to follow in given situations [33]. The difference between both frameworks lies on the alternate dimension of sequential interaction. Supervised learning can be seen as a simplified variant of imitation learning since in the latter we assume that an expert that we wish to mimic provides demonstration of a task. While "correct answers" are provided to the learner, any mistake made by the learner modifies future observations from what would have been seen had the expert chosen the controls.

Although RL could be categorized as a machine learning problem, it is quite different from traditional methods. While more widely studied problems of supervised learning is a presentation of the input/output pairs, the learning system is presented with a reward as an output of an action instead of being told which action it should have taken at that step. Another important difference is that the framework is optimised in an on-line fashion where the evaluation is concurrent with learning [4].

There are two main strategies for solving RL problems. One could search in the space of behaviours so as to find one that performs well in the current environment. On the other hand, one could use statistical techniques and dynamic programming methods to estimate the utility of taking actions in states of the world. Even though it is not yet clear which set of approaches is best in which circumstances, most of recent work has been devoted to the latter approach since it takes advantage of the special structure of RL problems that is not available in optimisation problems in general.

Reinforcement Learning problems can be framed as a Markov Decision Process (MDP). They are formally described by a tuple $< S, A, T, R >$ where S is the state space, A is the action space, $T : S \times A \rightarrow S$ it the transition function and $R : S \times A \rightarrow \mathbb{R}$ is the reward function [4]. The agent is connected to its environment via perception and action. For each interaction with the environment, the agent receives an input i and current state s; the agent then chooses an action a to generate as output. This action is responsible for changing the current state of the environment and this state transition is translated into a reward signal r. The key concept underlying RL is the Markov property - where

only the current state affects the next state or, in other words, the future is conditionally independent of the past given the present state.

In order to have an optimal behaviour, one needs to specify how the agent takes the future into account while taking decisions in the present. There are mainly three different ways for taking the future into consideration: finite-horizon, receding-horizon control and infinite-horizon. The first is summarized by the equation $\mathbb{E}[R] = (\sum_{t=0}^{h} r_t)$ and is the simplest from all three since the agent is expected to optimize the expected reward for the next h steps. The second method follows the same equation but instead of calculating the expectation until the final step h we limit how far ahead the agent is going too look. The infinite-horizon is a discounted model described by the equation $\mathbb{E}[R] = (\sum_{t=0}^{h} \gamma^t r_t)$ which uses a geometrically discounted value for the rewards received in the future. The γ in the equation is the discount factor (where $0 \leqslant \gamma < 1$). This can be mainly interpreted as a mathematical trick to bound the infinite sum [34]. The more general understanding behind it is that rewards received far in the future are worth less than rewards received in the present.

The usefulness of all three models has currently different perspectives. The finite-horizon model is a good fit for when the agent's lifetime is well known. A problem with a hard deadline would be appropriately modeled with this approach. On the other hand, the relative usefulness of infinite-horizon discounted and receding-horizon models is still under debate. While bias-optimality has the advantage of not requiring a discounted reward, its algorithms are not yet as well-understood as those for finding optimal infinite-horizon discounted policies.

All of the above criteria is used to evaluate the policies learned by a given algorithm. However, it is also desirable to be able to evaluate the quality of learning itself. Many algorithms have an eventual convergence to optimal. They come with a provable guarantee of asymptotic convergence to optimal behavior [35]. In many applications an agent that quickly reaches a plateau at 99% of optimality can be preferable to an agent that has only a stochastic guarantee of reaching an optimal. The speed of convergence to optimality is a quite complicated measure to define. Since most algorithms do not actually reach the true optimum, one should be talking about speed of convergence to near-optimality. However, this measure begs for the definition of how near to the optimal is sufficient. A practical approach, therefore, is to measure the performance after a given time.

2.7.1 Markov Decision Processes - MDP

In general a RL policy maps from states to a probability distribution over actions: $\pi : S \rightarrow p(A = a|S)$. The sequence of states, actions and rewards in an episodic MDP constitutes a trajectory or rollout of the policy. Every one of these accumulates rewards from the environment resulting in the discounted return $R = \sum_{t=0}^{T-1} \gamma^t r_{t+1}$. The goal of all algorithms is to find an optimal policy π^* which achieves the maximum expected return from all states:

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, \mathbb{E}[R|\pi]$$

The Markov property in RL helps understanding how current algorithms works. Any decisions made in a state s_t can be based solely on s_{t-1}, rather than $s_0, s_1, ...s_n$. This assumption is held by the majority of algorithms, however, it is somewhat unrealistic, as it requires fully observable states. In this way, a generalisation of MDPs known as Partially observable MDPs (POMDPs) is then used. An agent in a POMDP receives an observation o_t where the distribution of the observation $p(o_{t+1}|s_t + 1, a_t)$ is dependent on the current state and the previous action [36]. These algorithms typically maintain a belief over the current state given the previous belief state, the action taken and the current observation. There are two main approaches to solving RL problems: methods based on value functions and methods based on policy search [5].

2.7.2 Value Functions

Almost all RL algorithms involve estimating value functions. They determine how good it is for an agent to be in a given state. For this we use a state-value function $V^\pi(s)$. This function measures the expected reward when starting in state s and following π and is formally defined through $V^\pi(s) = \mathbb{R}[R|s, \pi]$. In the same way, an optimal policy π^* has a corresponding state-value function $V^*(s) = \max_\pi V^\pi(s)\, \forall s \in S$. With the optimal value function in hand, we could retrieve the optimal policy by choosing among all actions available at s_t the one that maximises the reward on that state.

As transition dynamics in the RL setting are unknown we need to construct another function that represents such transitions mathematically. The state-action-value function $Q^\pi(s, a)$ is similar to the value function previously defined except that the initial action is provided and the policy π is only followed from the next state onwards. In the same way as done with the value function, we can find an optimal policy by acting greedily at every state: $argmax_a Q^\pi(s, a)$. The value function can then be defined as

$V^{\pi}(s) = max_a\ Q^{\pi}(s,a)$. In order to learn Q^{π} one needs to use a Bellman Equation [37]:

$$Q\pi(s_t, a_t) = \mathbb{E}_{s_{t+1}}[r_{t+1} + \gamma Q_{\pi}(s_{t+1}, \pi(s_t + 1))].$$

2.7.3 Dynamic Programming

The Bellman optimality equation is a foundational equation in Reinforecement Learning. It can be solved using dynamic programming algorithms to find the optimal value function and the optimal policy. The equation can be proved using the Banach fixed point theorem where the Bellman optimality operator is a contraction over a complete metric space of real numbers [4]. The Bellman equation is defined as

$$V_*(s) = \max_a R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_*(s')$$

As mentioned in the previous section, one can find an optimal policy once the optimal value functions $v*$ or $q*$:

$$V_*(s) = max_a[R_{t+1} + \gamma V_*(s_{t+1})|S_t = s, A_t = a]$$
$$Q_*(s,a) = \sum_{s',r} p(s', r|s, a)[r + \gamma max_{a'} Q_*(s', a')]$$

The general idea of dynamic programming is to take these 2 equations above, and turn them into update rules for improving the approximations of our value functions.The first component of the framework is policy evaluation. It means that one needs to compute the state-value function V_{π} for an arbitrary policy π, this is referred by [4] as the "prediction problem":

$$v_{\pi}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_{\pi}(s')]$$

where $\pi(a|s)$ is the probability of taking action a in state s under policy π. Considering a sequence of approximate value functions $v_0, v_1, v2, ...$ where the initial approximation v_0 is chosen randomly, policy iteration algorithm obtains each following approximation using the Bellman equation as an update rule:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')]$$

In each iteration, the values are backed up from every state to produce the new approximate value function, this is referred as full backup policy evaluation [4].

Given that policies can be evaluated through the approximation of their value functions, one should now consider how to improve such policies. A simple way to answer this is by taking some action a in some state s and then to continue to follow the policy π. The idea is that taking the best action every time, could then lead to a policy that in fact is better overall. This allows the definition of a greedy policy π'

$$\pi'(s) = argmax_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]$$

The process of taking an old policy and making a new and improved one by selecting greedy actions with respect to the value function of the original policy is called policy improvement. Finally, once we use a policy π to yield a better policy π', we can then follow the same steps and improve it again. The technique of obtaining an optimal policy is known as policy iteration [4],

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*,$$

FIGURE 2.9: Policy Iteration, "E" denotes policy evaluation and "I" denotes policy improvement [4].

Generalised Policy Iteration refers to the idea of letting the two above processes - policy evaluation and policy improvement to interact. The main workflow is to randomly initialise the value function estimates of every state, and choose a random policy. We then evaluate the values of every state using this policy, and we update the policy by making greedy action choices, with respect to the value functions (e.g. taking the action that moves you to the state with the highest value). This framework is summarised in figure 2.10

2.7.4 Monte Carlo

Monte Carlo methods is an alternative to Dynamic Programming (DP) that can be applied in non-Markovian environments. However, they can only be used in episodic MDPs. This is due to the way these methods estimate the expected return. While in DP we use bootstrapping for improving value functions, in Monte Carlo the value is calculated through averaging the return from multiple rollouts. The λ value is used to interpolate between Monte Carlo evaluation and Bootstrapping. Figure 2.12 depicts a spectrum of RL methods based around the amount of sampling utilised.

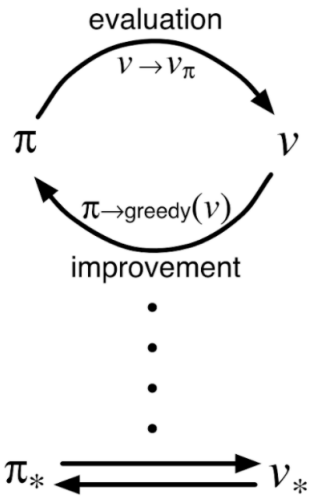


FIGURE 2.10: GPI Loops until it converges to the optimal policy [4].

The value-function methods presented until now are known for having very high variance. One way to stabilize the learning process is to learn an advantage function $A^\pi(s, a)$ [38, 39]. Unlike producing absolute state-action values, the advantage instead represents relative state-action values. This is akin to removing a baseline or average level of a signal; more intuitively, it is easier to learn that one action is better than another than it is to learn the actual return from taking the action. The advantage function can be formalized as $A^\pi = Q^\pi - V^\pi$. This idea has been applied in many recent Deep Reinforcement Learning algorithms [40, 41, 42].

Another spectrum of RL focuses on optimizing policies directly, these are known as policy search. Instead, these methods do not need to maintain a value function model, they directly search for an optimal policy π^*. The main goal is still to find the maximum expected rewards but now functions are usually approximated by a parameter θ. The parameters can be optimised using either gradient-based or gradient free methods [43]. Event though successes were achieved in both gradient based [44, 45] and gradient free methods [46, 47], gradient-based are still the technique of choice for training Deep RL algorithms.

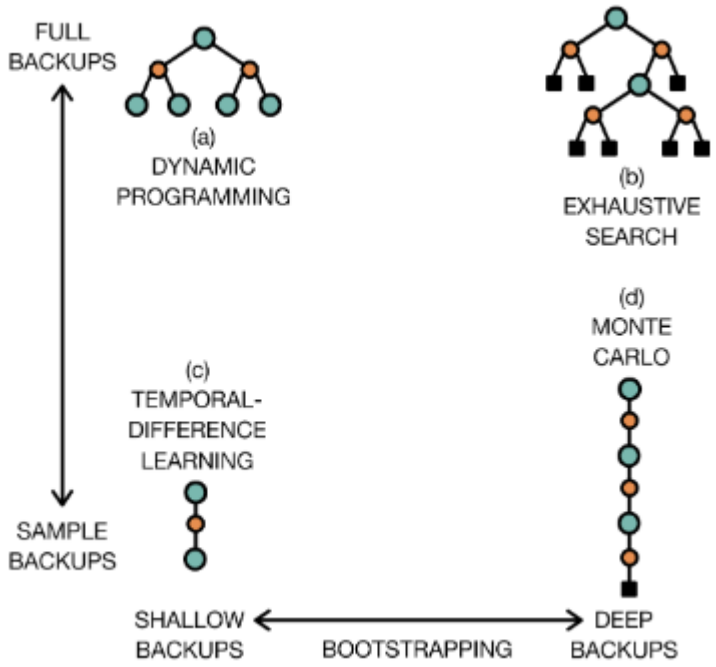


FIGURE 2.11: The whole spectrum of RL Methods [5].

2.8 Model-Free RL

2.8.1 Temporal difference Learning

Temporal different learning (TD Learning) refers to a class of model-free RL algorithms which learn by bootstraping from the current estimate of the value function. These methods sample from the environment like Monte Carlo methods and perform updates based on current estimates, they are very similar to dynamic programming methods. The difference between TD and Monte Carlo methods is that the first only need to wait until the next time-step to create and update instead of waiting until the end of the episode, when the reward is actually returned. The basic equation for performing value iteration is defined as follows:

$$V(s_t) = V(s_t) + \alpha[R_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

where the main difference is that instead of the target being set as the total reward G_t as in Monte Carlo, we rely on the TD error, as it measure the difference between $V(S_t)$ - the estimated value, and

$R_t + 1 + V(S_t + 1)$ - the better estimate. The above defintion is a one step TD, or TD(0) [4], that is, because it updates the value following the next state only, this can be seen as a special case of $TD(\lambda)$

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

where γ is the discount factor. The clear advantage of TD methods is that they can work in a model-free learning and they can also be used online, that is, we do not need to wait until the end of an episode to perform the calculations. Similarly, an evolution of TD learning is the SARSA algorithim [4].

2.8.2 Model Free vs Model Based RL

One of the most important branching points in an RL algorithm is the question of whether the agent has access (or learns) a model of the environment. By a model of the environment, we mean a function which predicts state transitions and rewards. The main upside to having a model is that it allows the agent to plan by thinking ahead, seeing what would happen for a range of possible choices, and explicitly deciding between its options. Agents can then distill the results from planning ahead into a learned policy. The main downside is that a ground-truth model of the environment is usually not available to the agent. If an agent wants to use a model in this case, it has to learn the model purely from experience, which creates several challenges. The biggest challenge is that bias in the model can be exploited by the agent, resulting in an agent which performs well with respect to the learned model but behaves sub-optimally in the real environment.

Algorithms which use a model are called model-based methods, and those that do not are called model-free. While model-free methods forego the potential gains in sample efficiency from using a model, they tend to be easier to implement and tune. This thesis only makes use of model-free methods, therefore, model-based are not covered.

2.8.3 Q-Learning

The key idea behind Q-Learning is to improve the way model-free is done. To do this, it combines the ideas from Monte Carlo and Dynamic Programming. Similarly to Monte Carlo methods, Q-Learning can work in a model-free learning. Similarly to DP, such methods update the value estimates based partially on other estimates, without having to go through the entire episode (this is called bootstraping).

In order to derive the Q-Learning algorithm, we need to rely on the original TD Learning definition as explained above. Instead of learning a state-value function, SARSA introduces the action-value function $Q_\pi(s, a)$ and it can be optmised similarly to the TD method above:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t]$$

The above update is done after moving from a state S to a state S' and, similarly to all on-policy methods, we continuously estimate Q_π while taking the greedy action in policy π. Finally Q-Learning presents a few differences to the SARSA algorithm. It is an off-policy control method and the update stage of Q(s,a) we take a maximum of next-state action pairs as follows:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

Doing such, the learned action-value function directly approximates Q^* - the optimal action-value function, independently of the policy being followed.

2.8.4 Policy Gradient

In essence, policy gradient methods update the probability distribution of actions so that the ones with higher expected reward have higher likelihood on a specific state. The objective function for policy gradients $J(\theta)$ is defined as:

$$J(\theta) = \mathbb{E}[\sum_{t=0}^{T-1} r_{t+1}]$$

In other words, the goal is to learn a policy that maximises the cumulative future reward to be received starting from any given time t until the terminal time T.

REINFORCE [4] is a Monte Carlo variant of policy gradients. The agent collects a trajectory τ of one episode using its current policy and updates the policy parameter. Since one full trajectory must be completed to construct a sample space, REINFORCE is updated in an off-policy way.

The objective of an agent is to maximize the expected reward when following a specific policy π. The main idea behind policy gradients is to push up probabilities of actions that lead to higher return, and push down the probabilities of actions that lead to lower return, until converging to an optimal policy,

$$\nabla J(\pi_{\theta_k}) = \mathbb{E}_\tau[\sum_{t=0}^{T-1} \nabla_\theta log \pi_\theta(a_t|s_t) G_t]$$

where G_t represents the cumulative rewards or the Q-value $Q(s_t, a_t)$. This can be usually parametrised with a function approximator such as a neural network as we will see later in this chapter.

The algorithm works by updating policy parameters via stochastic gradient ascent on policy performance:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_{\theta_k})$$

Training is done in an on-policy fashion which means that exploration happens by sampling actions according to the latest version of its stochastic policy. The amount of randomness in action selection depends on both initial conditions and the training procedure. Over the course of training, the policy typically becomes progressively less random, as the update rule encourages it to exploit rewards that it has already found. This can cause the policy to get trapped in local optima.

2.8.5 Actor-Critic Methods

A combination of value functions and an explicit representation of the policy is possible [4]. The policy (Actor) receives feedback from the "Critic" (Value Function) resulting in a tradeoff between variance reduction of the policy gradient with bias from value function methods [48]. The principle of the actor-critic method is them as the one in introducing unbiased estimates such as the baseline in traditional policy search algorithms. The main different is that in the actor-critic, the baseline is actually learned.

Actor-critic is similar to REINFORCE with baseline. As REINFORCE is the Monte Carlo learning that indicates that total return is sampled from the full trajectory. In the actor-critic, instead, we use a bootstrap technique, it mainly changes the advantage function

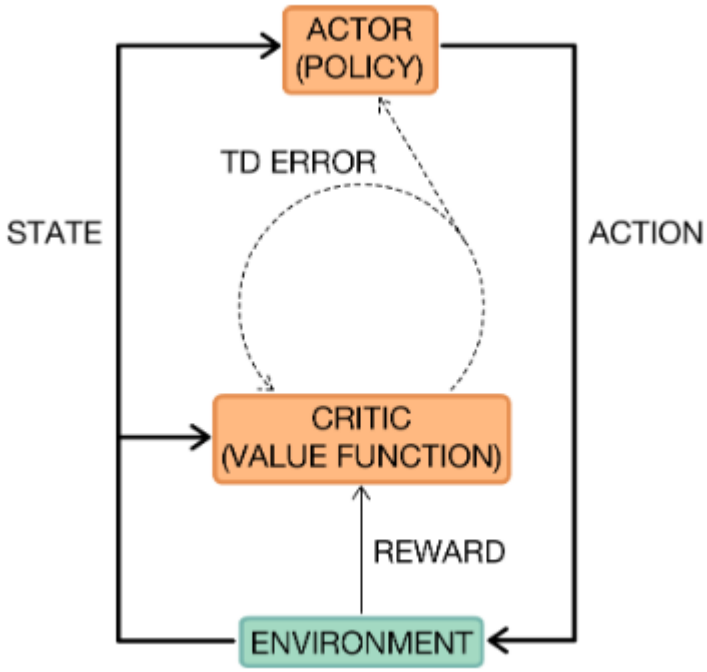


FIGURE 2.12: Actor-Critic methods combine value functions and policy search [4]

2.8.6 Deep Deterministic Policy Gradients - DDPG

Deep Deterministic Policy Gradient (DDPG) [47] is an algorithm which concurrently learns a deterministic policy and a Q-Function by using each to improve the other. Moreover, it is an actor-critic, model free algorithm based on the determinstic policy gradient that can operate over continuous action spaces. In the DQN Algorithm [49], the optimal action is taken by calculating the argmax over the Q-Values of all actions:

$$\mu(s) = argmax_a Q(s, a)$$

In DDPG, however, the actor is a policy network that outputs the action directly, bypassing the argmax operation. This creates a deterministic policy, and in order to promote exploration, some Gaussian noise is added to the action determined by the policy. To calculate the Q-Value of a state, the actor output is fed into the Q-Network. This is done only during the calculation of TD-error which will be demonstrated later.

The target networks are delayed networks compare to main/current networks. Their weights are updated periodically based on the main network. While in DQN the target gets a hard update, which means that network weights are copied over peridically, in DDPG a soft-update is performed, where only a fraction of main weights are transferred in the following manner:

$$\theta^{\mu}_{targ} = \tau\theta^{\mu}_{targ} + (1-\tau)\theta^{\mu}$$
$$\theta^{Q}_{targ} = \tau\theta^{Q}_{targ} + (1-\tau)\theta^{Q}$$

where τ is a parameter that is typically chosen to be close to 1 (e.g. 0.99).

As any actor-critic method, DDPG rely on two separate function approximators. The loss function for the Critic Q and Actor μ is defined as follows:

$$J_Q = \frac{1}{N}\sum_{i=1}^{N}(r_i + \gamma(1-d)Q_{targ}(s'_i, \mu_{targ}(s'_i)) - Q(s_i, \mu(s_i))^2$$

$$J_\mu = \frac{1}{N}\sum_{i=1}^{N} Q(s_i, \mu(s_i))$$

The Actor policy network loss is simply the sum of Q-Values for the states. In order to compute such values, we use the Critic Network and pass the action computed by the Actor-Network. The goal is to maximize this result as the best policies is that with maximum Q-Values. The Critic Loss is a simple TD-error where a different target network is used to compute Q-Values for the next state. This loss needs to be minimised and the error propagated backwards. The derivatives for the critic loss are simple since μ is treated as a constant, however, for the actor loss the μ function is contained inside the Q-Value, for this, chain rule is used as follows:

$$J_\mu = \mathbb{E}[Q(s, \mu(s))]$$
$$\nabla_{\theta^\mu} J_\mu = \mathbb{E}[\nabla_\mu Q(s, \mu(s)) \nabla_{\theta^\mu}\mu(s)]$$

In order to increase stability during training a target critic and actor networks are included to calculate Q-Value for next state in TD-error computations.

2.8.7 Hindsight Experience Replay - HER

One of the challenges for RL is the sparse reward setting. That is, when the agent only gets a reward if he reaches the goal state. However, most RL learning algorithms need to get feedback by a reward to learn to solve the task or to learn at all. In this way, by not getting any reward, most algorithms are destined to fail if they never encounter the goal state. In order to reach difficult goal states and to finally be able to learn, special exploration strategies are needed.

The unique idea behind HER [10] is to replay each episode with a different goal than the one the agent was initially trying to achieve. For instance, instead of reaching the exact goal, replace it by a new one that is in the vicinity of the true goal. In this case, one would replay the episode with the new goal that is led by a set of specific states, this usually is within the trajectory of the true goal, which allows episodes to become shorter and easier to learn. The simplicity of HER lies in the fact that breaking down an episode into shorter tasks leads to not only data augmentation for the learning algorithm but also more frequent rewards. As HER is a sampling technique it can be used with algorithms like DDPG and DQN, to improve their convergence in problems with sparse rewards settings.

2.8.8 Trust Region Policy Optimisation

The main problem with traditional policy gradient method is that during optimisation, overconfidence can lead to unpredictable moves that ruin the progress of training. Trust Region Policy Optimisation [50] solves this problem by establishing a trust region in which gradient updates can happen. This reduces the chances of having over confident updates through the steps ascent of gradient descent. For instance, Policy Gradient uses first-order derivatives and it expects surfaces to be flat; If the surface has high curvature, updates can end up overshooting. Other problems in such methods are that they cannot map changes between policy and parameter space easily, wrong use of learning rate causes vanishing or exploding gradient and, lastly, they have very poor sample efficiency.

The main idea in TRPO is that it guarantees that policy updates always improve the expected reward. It borrows ideas from the Minorize-Maximization algorithm [4]. Such method maximises a lower bound function iteratively, which is the same as approximating the expected reward locally. When training on policy, theoretically the policy for collecting data is the same as the policy that we want to optimise. However, when rollout workers and optimisers are running in parallel asynchronously, the behavior

policy can get stale. TRPO considers this subtle difference: It labels the behavior policy $\pi_{\theta_{old}}(a|s)$ and thus the objective function becomes:

$$J(\theta) = \mathbb{E}_{s\sim p^{\pi_{\theta_{old}}}, a\sim\pi_{\theta_{old}}} \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A_{\theta_{old}}(s,a) \right)$$

The method aims to maximise the objective function $J(\theta)$ constrained by the distance between old and new policies measured by KL-Divergence to be small enough, withint a paramater δ

$$\mathbb{E}_{s\sim p^{\pi_{\theta_{old}}}} \left[D_{KL} \left(\pi_{\theta_{old}}(.|s) || \pi_\theta(.|s) \right) \right] \leq \delta$$

2.8.9 Proximal Policy Optimisation

Proximal Policy Optimisation is a type of policy gradient algorithm where the policy is updated explicitly. The main challenge in the vanilla policy gradient is the high variance caused by gradient updates. The standard approach is estimate an advantage function that measures how much better an action is compared to another. The core goal is to reduce gradient variance between the old policy and the new policy.

PPO is a first order optimisation that simplifies the TRPO implementation. Similarly it defines the probability ratio between the new and old policies. Instead of adding the KL divergence, PPO imposes that the policy ratio should stay within a small interval around 1 that is the interval between $1-\epsilon$ and $1+\epsilon$ where ϵ is the hyperparameter chosen by the user. The PPO objective function can then be written as:

$$J(\theta) = \mathbb{E}[min(r(\theta)A_{\theta_{old}}(s,a), clip(r(\theta), 1-\epsilon, 1+\epsilon)A_{\theta_{old}}(s,a))]$$

where the function clip truncates the policy ratio between the range $[1-\epsilon, 1+\epsilon]$. The objective function of PPO takes the minimum value between the original value and the clipped value. Similar to what we discussed in vanilla policy gradient section above, positive advantage function indicates the action taken by the agent is good. On the other hand, a negative advantage indicated bad action. For PPO, in both cases, the clipped operation makes sure it won't deviate largely by clipping the update in the range.

2.8.10 RL and policy search in robotics

We consider the default RL scenario where an agent interacts in discrete timesteps with an environment $\mathbf{E}$. At each step t the agent receives an observation $\mathbf{o}_t$, takes an action $\mathbf{a}_t$ and receives a real number reward r_t. In general, actions in robotics are real valued $\mathbf{a}_t \in \mathbb{R}^D$ and environments are usually partially observed so that the entire history of observation, action pairs $\boldsymbol{\eta} = \{\mathbf{s}_t, \mathbf{a}_t, \mathbf{o}_t\}_{t=0}^{T-1}$. The goal is to maximize the expected sum of discounted future rewards by following a policy $\pi(\mathbf{a}_t|\mathbf{s}_t; \boldsymbol{\beta})$, parametrized by $\boldsymbol{\beta}$,

$$J(\boldsymbol{\beta}) = \mathbb{E}_{\boldsymbol{\eta}} \left[\sum_{t=0}^{T-1} \gamma^{(t)} r(\mathbf{s}_t, \mathbf{a}_t) | \boldsymbol{\beta} \right]. \tag{2.1}$$

In recent years, the advancements in traditional RL methods have allowed their application to control tasks with continuous action spaces. Inheriting ideas from DQN [51], Deep Deterministic Policy Gradients have been relatively successful in a wide repange of control problems. The main caveat of DDPG algorithms is that they rely on efficient experience sampling to perform well. Improving the way how experience is collected is one of most important topics in today's RL community. Experience Replay [52] and Prioritized Experience replay [53] still performs poorly in a repertoire of robotics tasks where the reward signal is sparse. Hindsight Experience replay (HER) [10], on the other hand, performs well in this scenario as it breaks down single trajectories/goals into smaller ones and, thus, provides the policy optimization algorithm with better reward signals. HER has been mostly based in a recent RL concept: Multi-Goal learning with Universal Function Approximators [54].

Another set of successful policy search algorithms is based on optimization through trust regions. They are less sensitive to the experience sampling problem mentioned above. The maximum step size for exploration is determined by its trust region and the optimal point is then evaluated progressively until convergence has been reached. The main idea is that updates are always limited by their own trust region, and, therefore, learning speed is better controlled. Proximal Policy Optimization [55] and Trust Region Policy optimization [50] have applied these ideas providing state of the art performance in a wide range of control problems.

Both techniques differ on the way they sample experiences. While the first is an off-policy algorithm - experiences are generated by a behaviour policy, the second is an on-policy algorithm where the policy used to generated experience is the same used to perform the control task. These algorithms will have comparable performance on different robotics control scenarios therefore should be considered the current state of the art on such problems.

2.9 Likelihood-Free Inference

Many domains of science have developed complex simulations to describe phenomena of interest. Mechanistic models can be used to predict how systems will behave in a variety of circumstances. For instance, those include particle physics, molecular dynamics, protein folding, population genetics and etc. The expressiveness of programming languages facilitates the development of complex, high-fidelity simulations and the power of parallel modern computing provides the ability to generate synthetic data from them. Unfortunately, these simulators, are not suited for statistical inference. The main problem is that the probability density (or likelihood) for a given observation (an essential ingredient for both frequentist and Bayesian Inference Methods) is typically intractable. Such scenarios have models that are often referred as implicit models and contrasted against prescribed models where likelihood for an observation can be explicitly calculated. The described setting has been dubbed likelihood-free-inference (LFI), which the mail goal is to estimate posterior distributions from data on systems where the likelihood is intractable.

In a typical scenario, a Likelihood-free inference method takes a *prior* $p(\boldsymbol{\theta})$ over simulation parameters $\boldsymbol{\theta}$, a black box generative model or simulator $\mathbf{x}^s = g(\boldsymbol{\theta})$ that generates simulated observations $\mathbf{x}^s$ from these parameters, and observations from the physical world $\mathbf{x}^r$ to compute the posterior $p(\boldsymbol{\theta}|\mathbf{x}^s, \mathbf{x}^r)$. The main difficulty in computing this posterior relates to the evaluation of the likelihood function $p(\mathbf{x}|\boldsymbol{\theta})$ which is defined *implicitly* from the simulator [56]. Here we assume that the simulator is a set of dynamical differential equations associated with a numerical or analytical solver which are typically intractable and expensive to evaluate. Furthermore, we do not assume these equations are known and treat the simulator as a black box. This allows LFI methods to be utilized with many robotics simulators (even closed source ones) but requires a method where the likelihood cannot be evaluated directly but instead only sampled from, by performing forward simulations. The most popular family of algorithms to address it are known as approximate Bayesian computation (ABC) [57, 58, 59].

2.9.1 Approximate Bayesian Computation - ABC

In ABC, the simulator is used to generate synthetic observations from samples following the prior. These samples are accepted when features or sufficient statistics computed from the synthetic data are similar to those from real observations obtained from physical experiments. As a sampling-based technique,

ABC can be notoriously slow to converge, particularly when the dimensionality of the parameter space is large.

A number of methodological advances have been developed to tackle the problems described above and, thus, enhance the sampling efficiency of traditional ABC. The most common approach is to use a set of monotonically decreasing ϵ values, allowing the algorithm to sequentially adapt the prior distribution and converge to a computationally feasible final value of ϵ. However, such algorithms still rely on a step function (boxcar kernel) to evaluate the fitness of each sample, therefore are not particularly efficient in high dimensional search spaces.

2.9.2 Rejection ABC

Formally, Rejection ABC approximates the posterior $p(\boldsymbol{\theta}|\mathbf{x} = \mathbf{x}^r) \propto p(\mathbf{x} = \mathbf{x}^r|\boldsymbol{\theta})p(\boldsymbol{\theta})$ using the Bayes' rule. However as the likelihood function $p(\mathbf{x} = \mathbf{x}^r|\boldsymbol{\theta})$ is not available, conventional methods for Bayesian inference cannot be applied. ABC sidesteps this problem by approximating $p(\mathbf{x} = \mathbf{x}^r|\boldsymbol{\theta})$ by $p(\parallel \mathbf{x} - \mathbf{x}^r \parallel < \epsilon|\boldsymbol{\theta})$, where ϵ is a small value defining a sphere around real observations $\mathbf{x}^r$. The quality of the approximation increases as ϵ decreases however, the computational cost can become prohibitive as most simulations will not fall within the acceptable region. This method has the practical advantage of being relatively easy to implement and use, but its efficiency depends critically on the choice of the prior sampling distribution. If the prior distribution is a poor approximation of the actual posterior distribution, then many of the proposed samples will be rejected. This leads to dramatically low acceptance rates, and hence excessive CPU times.

In other words, the ABC rejection (ABC-REJ) algorithm proceeds as follows. First we sample a candidate point θ^* from some prior distribution $p(\theta)$. We then use this proposal to simulate the output of the model, $y \sim Model(\theta^*)$. We then compare the simulated data y, with the observed data $\hat{y}$ using a distance function. If this distance function is smaller than some small positive tolerance value, ϵ, then the simulation is close enough to the observations and θ^* has some nonzero probability of being in the approximate posterior distribution. By repeating this process N times, Rejection ABC provides an estimate of the actual posterior distribution.

Standard rejection sampling method usually requires massive computational resources to generate sufficient number of samples from the posterior distribution. Under sampling probable regions of the parameter space results in an undesirable sample density which can lead to misleading results particularly on high dimensional data.

2.9.3 Monte Carlo ABC

As seen on previous section, rejection sampling is inefficient and can cover only a limited amount of problems. In order to guarantee convergence to the appropriate limiting distribution, the value of ϵ in the Rejection Algorithm needs to be taken very small. Ranging from 0.01 to 0.05, are often deemed appropriate. This will, unfortunately, produce very low acceptance rates, particularly, if the the prior distribution is not the adequate. To increase sampling efficienty, one needs to reduce the value of ϵ and to use the accepted samples to iteratively adapt the prior distribution. This is the principal idea behind population Monte Carlo (PMC) algorithms. In short, the ABC-PMC sampler starts out as a ABC-REJ during the first iteration, but using much larger initial value for ϵ. This results in a much higher acceptance rate. On each successive iteration, the value of ϵ is decreased and the distribution is updated accordingly. In summary, a sequence of proposal distributions is used to iteratively refine the samples and explore the posterior distribution.

The PMC Sampler, assumes that the sequence of ϵ values is specified by the user. Practical experience suggests that a poor selection of $\epsilon = \epsilon_1, ..., \epsilon_n$ can lead to very low acceptance rates, or, sometimes, premature convergence. An improvement developed in [60] introduces an alternative variante of the ABC-PMC. The adaptive selection of ϵ_j with $j > 1$. This method requires the user to specify only the initial kernel bandwidth ϵ_1, and subsequence values of ϵ are determined from the values of the N most recently accepted samples. This approach is not only more practical, but also enhances convergence speed to the posterior distribution.

The adaptive capabilities of the ABC-PMC sampler can offer several computational advantages over the Rejection ABC algorithm. However, there is still further methodological advances that enable inference of complex simulation models, involving high dimensional parameter spaces. The use of boxcar fitness kernel is theoretically convenient but makes it very difficult for any sampling algorithm to determine the preferred search direction.

2.9.4 ϵ-free with Bayesian Conditional Density Estimation

As seen in previous section, ABC algorithms simulate the model for each setting of proposed parameters, and rejects parameters where the generated data is not within a certain distance from the observations. Accepted parameters form a set of independent samples from an approximate posterior. Such algorithms, suffer from some drawbacks, for instance, sample based estimation easily gives estimates and errors bars of individual parameters but the process is considerably noisy and slow. Moreover, the parameter samples do not come from the correct posterior, but rather from an approximation assuming that the data is within an ϵ-ball centred on the data actually observed. Finally, as we reduce values of ϵ, it can become impractical to simulate the model enough times to match the observed data even once. This is more relevant on scenarios where simulations are expensive to perform as the small amount of data collected hinders good quality inference.

In ϵ-free approaches [61], a parametric approach to LFI is used, which unlike conventional ABC does not suffer from the above issues. The goal is to learn the exact posterior over parameters using a parametric approximation. This is not only faster than returning samples from an ϵ-approximation to that posterior, but it is more accurate. This approach relies on conditional density estimation with Bayesian neural networks.

CHAPTER 3

Egocentric Activity Recognition on a Budget [1] [2]

3.1 Introduction

Recent advances in embedded technology have enabled more pervasive machine learning. One of the common applications in this field is Egocentric Activity Recognition (EAR), where users wearing a device such as a smartphone or smartglasses are able to receive feedback from the embedded device. Recent research on activity recognition has mainly focused on improving accuracy by using resource intensive techniques such as multi-stream deep networks. Although this approach has provided state-of-the-art results, in most cases it neglects the natural resource constraints (e.g. battery) of wearable devices. We develop a Reinforcement Learning model-free method to learn energy-aware policies that maximize the use of low-energy cost predictors while keeping competitive accuracy levels. Our results show that a policy trained on an egocentric dataset is able use the synergy between motion and vision sensors to effectively tradeoff energy expenditure and accuracy on smartglasses operating in realistic, real-world conditions. The main caveat of this approach is that it has been trained in simulation only, hence, battery consumption results are only valid for optimal conditions where the battery discharges at its standard rate. This can change drastically if factors such as temperature increases significantly. For instance, in a hot day, the battery can overheat leading to a higher discharge rate. Such changes in temperature were not accounted for in simulation, leading to reality gap. This chapter sets the foundation for this entire thesis, as we have identified that solving the "reality gap" is an important step forward to allow machine learning to be pervasively deployed in the real world.

In this chapter, we propose an energy-aware framework for EAR whose goal is to minimize energy consumption while maintaining reasonable predictive performance. Specifically, we make the following contributions:

[1] The contents of this chapter were presented in CVPR 2018 as an Oral Talk

[2] https://openaccess.thecvf.com/content_cvpr_2018/papers/Possas_Egocentric_Activity_Recognition_CVPR_2018_paper.pdf

- We propose a Reinforcement Learning (RL) Policy Gradient framework that balances energy consumption and accuracy through a customizable hyper-parameter.
- We achieve higher accuracy over other benchmark, Multimodal egocentric dataset as presented in [62] while using less energy compared to previous work.

Although we have shown promising results, the energy consumption optimizer will only work in ideal conditions, one in where the energy threshold for the battery is not violated, allowing it to discharge in a predictable manner.

3.2 Related Work

Activity recognition from visual information has attracted great attention in the last years. It has been traditionally tackled using third-person view cameras and then extended to egocentric cameras. In the external perspective context, the successful application of deep learning approaches for image classification [63] has resulted in their extension to the context of activity recognition over video [64, 65, 66, 67, 68].

At first, a natural step has been to extend Convolutional Neural Networks (CNNs) 2D filters that explore spatial information over images to 3D filters to add temporal information over videos as it was done in [69]. In fact, Karpathy et al. [66], explored spatio-temporal schemes using both 2D and 3D convolutions, finding that 3D convolutions over videos were giving a very short gain in performance in comparison with 2D convolutions over images of single frames of videos. This fact suggests that (CNNs) architectures are not able to properly learn motion features, due to the non-existence of sufficiently large video datasets. Simonyan et al. [68] proposed the use of stacks of dense optical flows to encode temporal information achieving higher success.

In order to preserve longer temporal information, some architectures based on Recurrent Neural Networks (RNNs) have been proposed [64, 65, 67]. Donahue et al. [65] have used Long Short-Term Memory (LSTM) networks over features obtained from a CNN applied to single frames of videos. Ng et al. [67] proposed a similar idea over both single frames and optical flow. Finally, Ballas et al. [64] suggested the use of convolutions inside the recurrent units of GRU to better capture temporal features of the sequence of images.

In the context of EAR, traditional vision methods have encoded prior knowledge by using handcrafted features and mid-level representations involving the detection of objects [27, 70, 71, 72, 73], hands [71, 72, 73, 74, 75, 76], gaze [71, 74], motion [72, 74, 77, 78, 79], among others. Object-based techniques presumes that an activity can be inferred by the group of objects that appear in a video. Thus, object-based techniques rely on the object recognition domain and, therefore, inherit its challenges. Another commonly used strategy has involved the use of optical flows to express motion [62, 79]. Motion-based techniques focus on the fact that different kind of activities create different body motions presenting remarkable robustness to deal with some of the vision challenges. However, they perform poorly when dealing with activities that lacks movement patterns such as sitting, watching TV, and reading.

Zhan et al. [79] combined optical flow from video with acceleration features. Ogaki et al. [77] used ego-motion features combined with eye-motion in order to improve the classification accuracy. Finally, Ryoo et al. [78] describes a novel feature representation based on time series pooling to represent motion. Activities such as walking, lying down, or running, for example, do not require the use of a specific object or the hands.

Most of the aforementioned representations prevents the learning algorithm from generalizing to a more realistic set of activities. Ideally, a framework must learn meaningful features automatically. Recent advances in third-person activity recognition have used deep learning to address this requirement. Simonyan et al. [68] proposed a Two-Streams CNN approach over spatial (image frames) and temporal (optical flows) streams and has been taken as a baseline of other works since it outperformed handcrafted features methods [80]. Song et al. [62] extended this approach to the EAR domain obtaining encouraging results.

One reason by which the novel strategy did not overcome previous methods yet could be the nonexistence of large enough datasets which prevent the networks from learning more complex patterns proper of the egocentric domain. In fact, due to the restricted size of egocentric datasets, both streams used pretrained models over non-egocentric datasets.

Furthermore, the advent of wearable devices brought new challenges. Visual information is usually affected by real life conditions of the wearer. For example, shaken and blurred shoots are common in activities that require movements. Dark and rainy environments lead to unintelligible images. In these conditions, the use of another source of information such as accelerometers and gyroscope becomes

essential. In fact, Egocentric Activity Recognition is one of the fields that has drawn most benefits from the use of multiple sensors [81].

Wearable cameras can produce visual information that is affected by real conditions of the wearer leading to unintelligible images. Thus, the use of other sources of information is needed. Wearable devices have typically been attached to some parts of the body [82, 83, 84], to external objects [85, 86] and to the environment [83, 84] and their use was often limited to controlled experiments, differing in high degree from a real life application.

Reinforcement Learning (RL) is another technique that has achieved successful results lately. Successes are wide, ranging from playing Atari games [49] to teaching a robot to play soccer [87] or detecting objects in vision tasks [88]. In the context of budget-restricted prediction, Karayev et al. [89] applied RL to determine the order of computation of handcrafted features. Although budget awareness is desirable, handcrafted features increases the complexity and could prevent generalization to new domains as described before. To the best of our knowledge, there is no current application of RL on the EAR problem.

Energy awareness has been greatly overlooked in all recent multi-stream activity recognition work. This factor can not be ignored anymore if we desire to use these methods in a realistic setting which has major constraints such as battery consumption. Despite all previous work, there have been no/few attempts to balance computational resources and other constraints with the overall accuracy on activity recognition tasks. Therefore we propose a Reinforcement Learning framework that makes use of policy learning in order to balance two different activity predictors using data from motion and vision sensors.

3.3 Method

The goal of the method introduced in this section is to perform decision making in the EAR context using a RL approach that balances energy consumption and accuracy on resource constrained devices.

3.3.1 Overview

We consider tasks in which a wearable device receives data from motion sensors and uses a policy $\pi_\theta(a_t|s_t)$ to take actions $a \in (\alpha, \beta)$ from a given state s, where action α means motion predictor is used while action β means the vision predictor is used to recognize the activity. At each time-step t we give a

reward r_t that reflects the accuracy and energy consumption of the selected action. The long-term return $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ is the total accumulated reward from time step t with discount factor $\gamma \in (0, 1]$. We also define a training dataset $D = \{x_t, y_t\}_{t=1}^{N}$ with $x_t = \{I_t, acc_t, gyr_t\}$ where I_t is a sequence of images from the device's camera, and acc_t and gyr_t is a sequence of accelerometer/gyroscope x,y,z values. The class labels for the 20 activities are denoted by $y_t \in \{y^1, ..., y^{20}\}$.

Our framework is shown in Figure 3.1. We learn a policy for energy optimization through a LSTM, mapping accelerometer and gyroscope data to actions. The network outputs a probability distribution over actions $a \in (\alpha, \beta)$ that attempts to select actions with higher average rewards. The model's actions define whether the final prediction should come from a motion predictor, defined by $\rho^m(x_t)$, or vision predictor, defined by $\rho^v(x_t)$. These are represented by a separate LSTM and LRCN Network respectively.

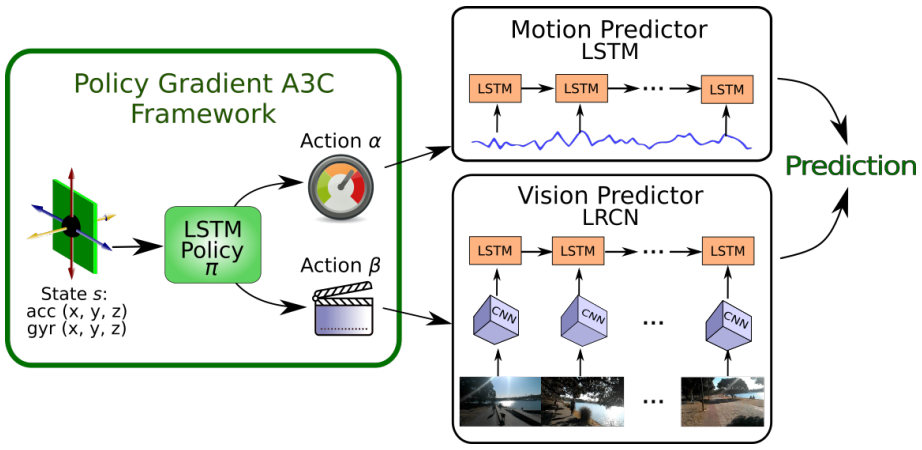


FIGURE 3.1: **(Left)**: Policy Function approximator represented by a LSTM Neural Network. **(Right)**: Activity predictors pre-trained on an activity recognition dataset.

3.3.2 LSTM (Motion) Predictor

For predictions of sensor data we use a LSTM network as explained in chapter 2, section 2.5. This is a specific Recurrent Neural Network (RNN) architecture that has been widely used to solve problems where data has an intrinsic temporal structure. Its main idea is to regulate the flow of information through neurons' specific gates. They control how much information should be remembered or whether the network should forget or keep a memory. LSTMs were designed to solve the problem of long-term dependencies on vanilla RNNs, where data with long temporal dependencies caused the gradient of the network to vanish or explode. Our motion predictor architecture is composed of one layer with 64 neurons unrolled through time and is optimized using RMSProp with learning rate of 0.001. The main

advantage of using motion data for prediction is that it has a very low energy profile as mobile devices require minimum energy to capture their values. Its predictive strength concentrates on activities with high body movement patterns such as running, walking and cycling. On the other hand, the network often performs poorly for activities where there is no such pattern and/or limited movement. For this reason, we also use a vision predictor that helps to increase accuracy in these scenarios.

3.3.3 LRCN (Vision) Predictor

CNNs have dominated recent image recognition tasks. However, these models are not very effective on tasks involving sequences of images. Our vision predictor uses a mix of CNNs and RNNs called Long-term Recurrent Convolutional Networks (LRCN). The model's first layer uses an Inception V3 pre-trained on Imagenet followed by a LSTM with 512 neurons and a softmax on the last layer. The CNN acts as a feature extractor while the LSTM captures the temporal structure of the data. All training happens end-to-end where we first freeze the inception layers and train only the LSTM for 10 epochs. Then, we unfreeze the 3 last blocks of the CNN and train again for 20 more epochs. The model receives a sequence of images and outputs a vector of activity probabilities. Even though the accuracy of vision methods have shown higher overall accuracy on previous works [62, 90, 91], it still is very energy inefficient model. For instance, we evaluated the camera's consumption on a Vuzix M300 smartglasses, and it takes on average three times more energy than only motion sensor measurements. Our aim is to optimize the overall accuracy by balancing our predictors usage appropriately.

3.3.4 Reinforcement Learning

Until recently, RL methods were constrained to discrete state spaces and actions. However, the use of deep networks as function approximators have extended their use to continuous inputs and outputs. While in supervised learning the main goal is to only map inputs to outputs, in RL the choice of what to approximate is what makes current methods to differ from each other. One could optimize policies, value functions, dynamic models or some combination thereof. In fact, there are mainly two perpendicular choices to be made: what kind of objective to optimize (e.g. policy, value function or dynamics) and what kind of function approximators to use. In these lines, our framework can be fully defined by the tuple $< S, O, A, r, \lambda, \gamma >$, with:

- S: Set of states $\{s_1, ..., s_n\}$ where $s_t := (o_t, y_t)$ where,

  o_t: Sensor observation at time-step t,

  y_t: True activity label at time-step t.
- O: Set of observations $\{o_1, ..., o_n\}$ where $o_t := (acc_t, gyr_t)$ and,

  acc_t: 3D accelerometer values at time-step t,

  gyr_t: 3D gyroscope values at time-step t.
- A: Set of actions $\{a_1, ..., a_n\}$ where $a_t \in [\alpha, \beta]$ where,

  α: Motion (LSTM) is used for prediction,

  β: Vision (LRCN) is used for prediction.
- r: $S \times A \rightarrow \{1 + \lambda, -1 - \lambda, 1, -1, \lambda\}$ is a reward function with five possible outputs, corresponding to executing an action a_t in state s_t.
- $\lambda \in [0, 1]$: Parameter that limits the rewards given to high energy consumption actions.
- $\gamma \in (0, 1]$: is the discount factor on future rewards.

Further, we denote the LSTM policy network by $\pi_\theta(a_t|s_t)$ which is parametrized by θ. Through the RL framework, we select actions that give the highest average reward for every hidden representation of the input state s_t. The network architecture is composed of a layer with 256 neurons followed by a softmax with one neuron for each action. The policy $\pi_\theta(a_t|s_t)$ chooses between two different actions: predict y_t (the activity label) from the motion predictor $\rho^m(x_t)$ or predict y_t from the vision predictor $\rho^v(x_t)$. Since our state is partially represented by the accelerometer and gyroscope readings, these are required to be turned on in the device at all times.

Reward Function: The design of the reward function is one of the most important parts of any RL framework. Its outputs should be able to increase or decrease the likelihood of selecting an action in order to provide a better behavior for the entire system. Positive rewards increases the probability of an action through gradient ascent, while negative rewards will decrease it through gradient descent. In our case, we have a lower accuracy but more energy efficient motion predictor and a more energy demanding/accurate vision predictor. The goal is to only use vision methods when low-energy motion predictions are incapable of providing the right outcome. In order to determine the reward, we use a training dataset where we have access to the correct labels for all given inputs. We evaluate the results on both motion and vision predictors calculating the advantage of choosing one predictor over the other and updating our policy accordingly. For instance, we give only λ reward for choosing the LRCN model for prediction using vision when we observe that the LSTM is also able to provide a

correct outcome using motion sensors. The function gives constant rewards $r_t \in [-1, 1]$ for choosing the motion predictor while giving λ controlled rewards to vision predictor actions. The formal definition of the reward $r_t(s, a)$ is as follows,

$$\begin{cases} 1 & \pi_\theta(a_t|s_t) = \alpha,\ \rho^m(x_t) = y_t \\ 1+\lambda & \pi_\theta(a_t|s_t) = \beta,\ \rho^m(x_t) \neq y_t,\ \rho^v(x_t) = y_t \\ \lambda & \pi_\theta(a_t|s_t) = \beta,\ \rho^m(x_t) = y_t,\ \rho^v(x_t) = y_t \\ -1-\lambda & \pi_\theta(a_t|s_t) = \beta,\ \rho^m(x_t) = y_t,\ \rho^v(x_t) \neq y_t \\ -1 & \text{otherwise.} \end{cases}$$

Episodes and Steps: Each episode contains 15 seconds of data split in equal time-steps t of 1 second each. Videos on the Multimodal dataset have 1 single activity per 15 seconds of footage while on DataEgo, a video has 5 minutes of duration conformed by 4-6 activities. Actions are taken every second using the readings from both accelerometer and gyroscope. A separate buffer of image is kept during training in order to evaluate the reward function. On real life settings the camera would need to be turned on in order to provide data for the LRCN model.

3.3.5 Policy Learning

We solve the RL problem through the use of a model free framework. The goal is to optimize a parametrized stochastic policy $\pi_\theta(a_t|s_t)$ and a value function $V_{\theta_v}(s_t)$ using gradient methods. While policy methods learn a policy directly, value iteration methods such as Q-Learning focuses on updating state-value functions using the Bellman equation [92]. Our method uses an Actor-Critic framework that combines the benefits of both approaches. The actor takes actions based on a policy $\pi_\theta(a_t|s_t)$ and an estimate of the value function (critic) $V_{\theta_v}(s_t)$. The value function $V_{\theta_v}(s_t)$ determines how good a certain state is while following a policy π_θ. While actor and critic parameters θ and θ_v are shown as being separate, we share some of the parameters in practice to improve stability. We use our LSTM policy network with a softmax output for the policy $\pi_\theta(a_t|s_t)$ and one linear output for the value function $V_{\theta_v}(s_t)$.

Policy learning is performed through infinitesimal updates in both θ and θ_v. The sign and magnitude of our reward determines if we are making an action more or less probable as it performs gradient ascent and descent respectively. For the policy update, we calculate the gradient over an expectation using a *score function* estimator as shown in the work of [93]. The value function is updated using a squared

loss between the discounted reward and the estimate of the value under parameters θ_v. Optimization is a two-step process where we first train our predictors $\rho^m(x_t)$ and $\rho^v(x_t)$ on the training dataset and then we use their predictions to optimize both the policy θ parameters and value function θ_v parameters.

The main benefit of the actor-critic method is to use an advantage function instead of discounted rewards in the update rule. As rewards have high variance during learning, the use of an estimated value speeds up the process while reducing variance on updates. The advantage function $A(s_t, a_t, \theta_v)$ is an estimate of the advantage and is given by $A(s_t, a_t, \theta_v) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V_{\theta_v}(s_{t+k}) - V_{\theta_v}(s_t)$.

We also added entropy regularization to our policy updates as proposed by [94]. The idea is to have a term in the objective function that discourages premature convergence to suboptimal deterministic policies. The final update rule for the algorithm takes the form

$$\theta \leftarrow \theta + \nabla_\theta(\log\ \pi_\theta(a_t|s_t))\ A(s_t, a_t, \theta_v) +\ \eta\ \nabla_\theta H(\pi_\theta(a_t|s_t)),$$

where $H(\pi_\theta(a_t|s_t))$ is the entropy for our policy and η is the parameter that controls its relative importance. In this way the algorithm is more efficient, leading to better convergence of policies

3.3.6 Asynchronous optimization

The availability of multi-core processors justifies the development of RL techniques with asynchronous updates. Recent work of [50, 51] have shown that updates in on-line methods are strongly correlated mainly because the data observed from RL agents is non-stationary. Techniques like experience replay as presented in [51] focuses on storing batches of experience and then performing gradient updates with random samples from each batch.

Here, we follow a different approach to solve the same problem. We use asynchronous gradient optimization of our controllers that executes multiple workers in parallel on multiple instances of the environment. This process decorrelates the data into a more stationary process. In fact, this simple idea enables a much larger spectrum of RL algorithms to be executed effectively. The asynchronous variant of actor-critic methods is the state-of-the-art method on several RL complex domains as it was shown by Mnih et al (2016).

3.4 Experiments

3.4.1 Datasets

The large majority of previous work on egocentric Activity Recognition have used either raw data acquired from sensors or video data from cameras (not both). One of the few datasets available was proposed by [91]. We refer to this dataset as Multimodal. Its main limitation is that videos are split by activity instead of a more natural setting where there is a flow between different activities.

FIGURE 3.2: The sequence of activities was developed by different subjects.

This work uses a novel egocentric activity dataset DataEgo that contains a very natural set of activities developed in a wide range of scenarios. There are 20 activities performed in different conditions and by different subjects. Each recording has 5 minutes of footage and contains a sequence of 4-6 different activities. Images from the camera are synchronized with readings from the accelerometer and gyroscope captured at 15 fps and 15 Hz respectively. In total, our dataset contains approximately 4 hours of continuous activity while the multimodal dataset has only 50 minutes of separate activities. DataEgo is publicly available in the following link [3].

3.4.2 Predictors Benchmark

The results for our individual predictors are shown on Table 3.1. It can be seen that if we consider methods individually without any type of sensor fusion or extra features such as the ones from optical

[3]Dataset link: `http://sheilacaceres.com/dataego/`

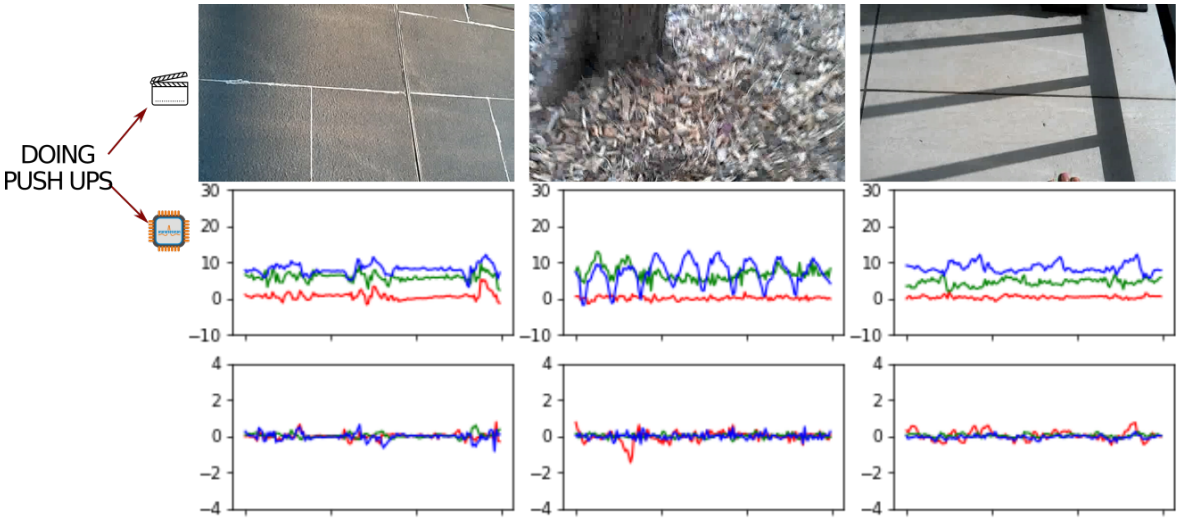


FIGURE 3.3: Different streams of data for a single activity.

flow, our methods have the highest overall accuracy. Our LRCN network has achieved an accuracy of 78.70% which sits very close to the more resource intensive methods from previous work of [62, 91]. Our LSTM also outperforms previous work on motion sensors by almost 10%. We believe that this result is due to the stateful approach we used during training. The idea is to save the hidden states in between batches so as to better capture the temporal structure within the data.

Method	**Dataset**	**Accuracy** (%)
LRCN (vision)	Multimodal	78.70%
LSTM (motion)	Multimodal	61.24%
LRCN (vision)	DataEgo	71%
LSTM (motion)	DataEgo	58%
CNN FBF (vision) [62]	Multimodal	70%
Multi Max (both) [62]	Multimodal	80.5%
Fisher Vector (both) [91]	Multimodal	83.7%
LSTM (motion) [62]	Multimodal	49%

TABLE 3.1: Comparison of motion and vision predictors with previous work shows higher accuracy when comparing to single stream methods.

3.4.3 Convergence of A3C

Training results of the RL framework are shown on Figure 3.4. Convergence was achieved with approximately 600 episodes for each of the actor-learners. The running mean of rewards presents an exponential

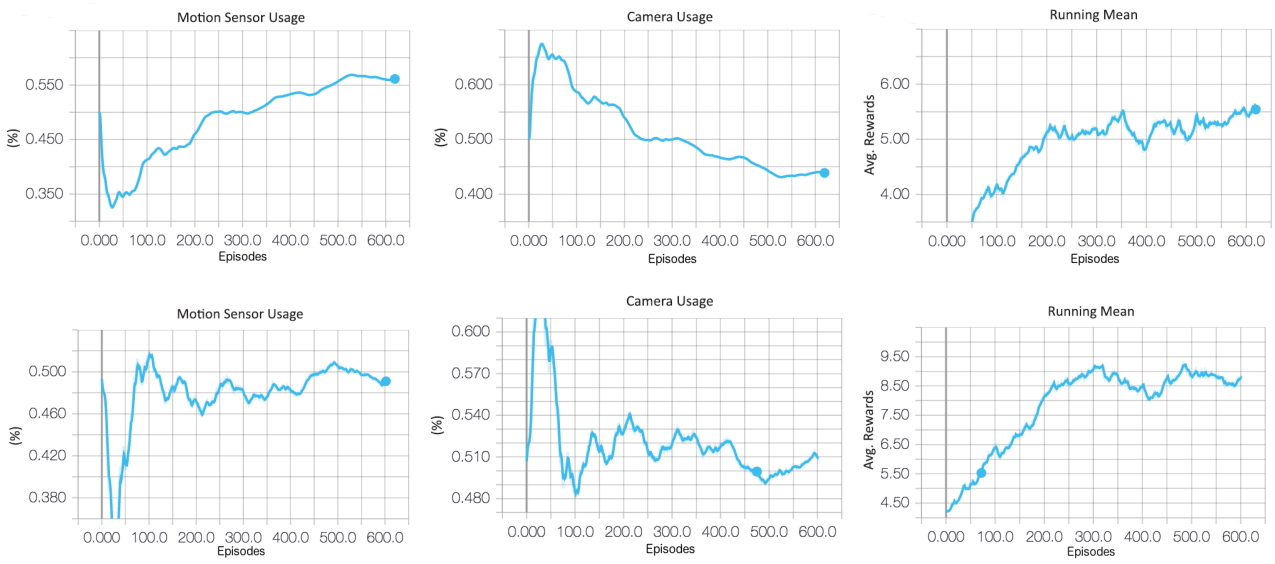


FIGURE 3.4: Convergence of A3C shows small variance on motion/vision usage and average rewards after 600 episodes for both Multimodal (**top**) and DataEgo (**bottom**) datasets.

increase initially while stabilizing with fixed small variance at the end. The results illustrates that our algorithm finds a stable policy for $\lambda = 0.2$ while equally balancing usage of low/high energy consumption predictors.

3.4.4 Motion vs Vision Tradeoff

Figure 3.5 compares the effect of λ on the overall per class results for the Multimodal dataset. Activities such as organizing files, riding elevators and others have greatly improved their accuracy by using the vision predictor. This shows that our policy is in fact learning actions that exploits the different strengths of our predictors.

Validation for the aforementioned results was perfomed through an analysis of how actions were being chosen amongst different activities. We sampled the softmax outputs on the multimodal test dataset while using a learned policy with $\lambda = 0.2$. As can be seen on Figure 3.6, activities such as organizing files and riding elevators/escalators presented higher probabilities on using the vision predictor, while running, doing sit-ups and walking up/downstairs were dominated by the motion predictor. This fact is consistent with a real life setting as activities with higher movement patterns are prone to perform better with motion sensor data.

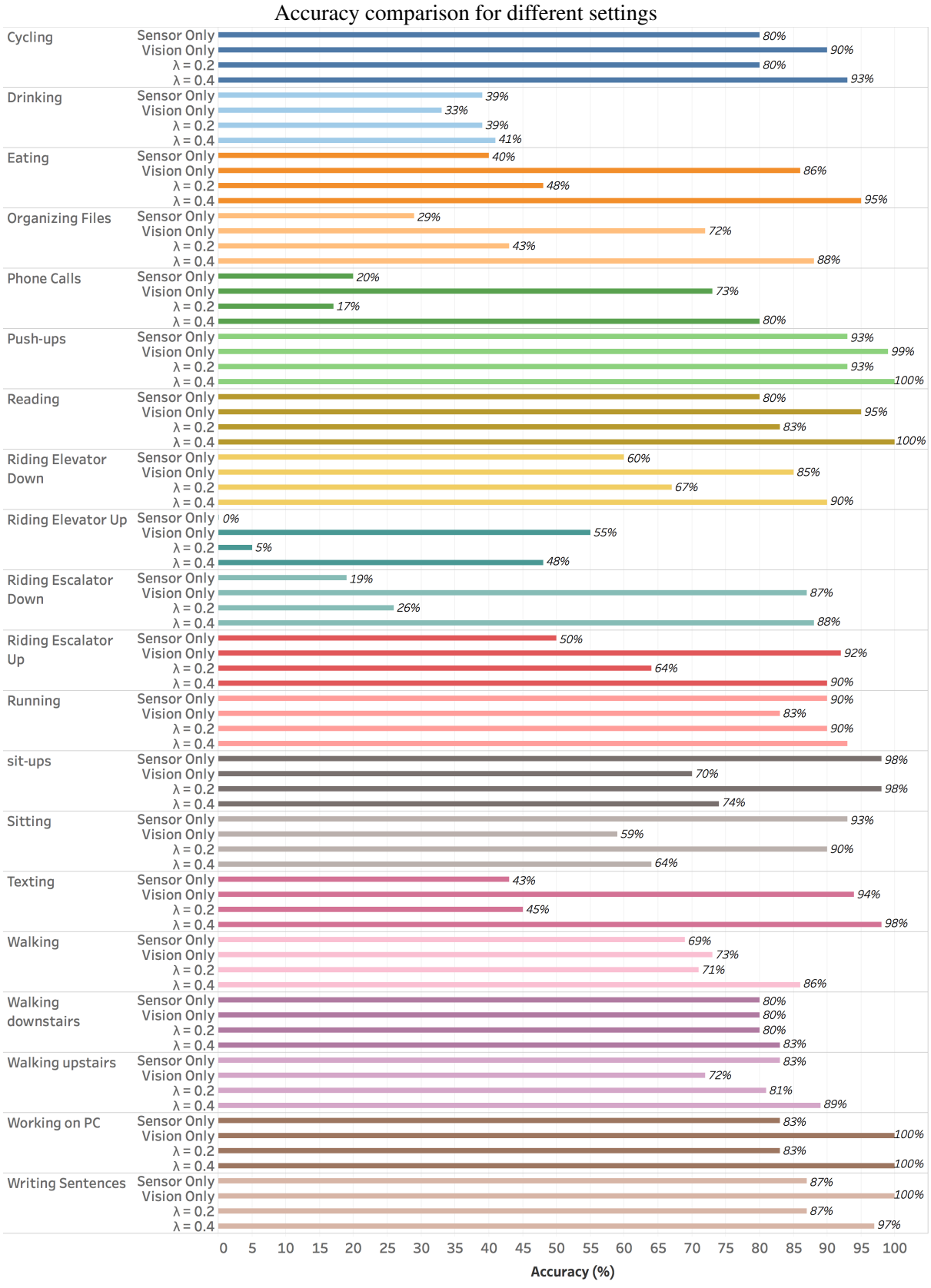


FIGURE 3.5: Individual per-class accuracy on Multimodal dataset shows that activities with low body movement seems to benefit the most from vision predictor.

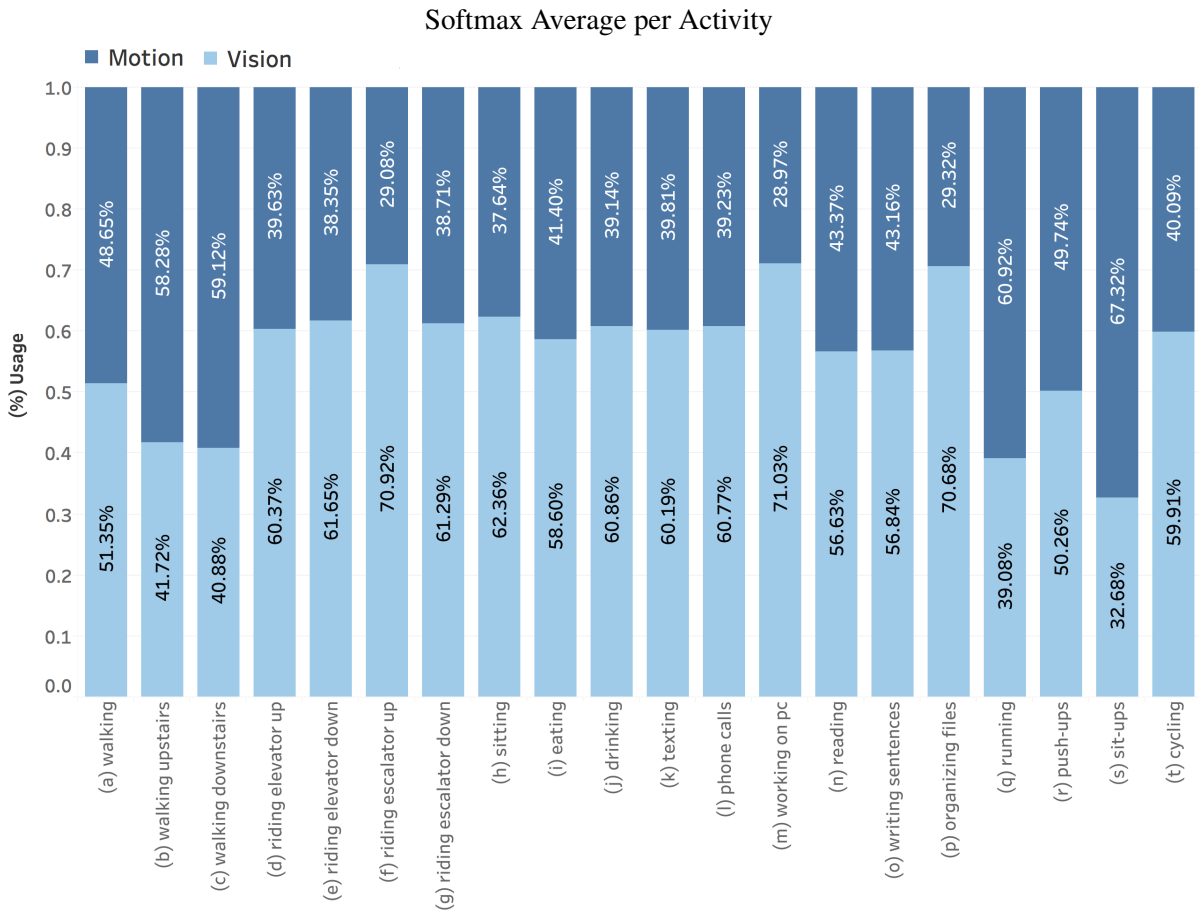


FIGURE 3.6: The softmax average per activity shows synergy between predictors on a $\lambda = 0.2$ policy.

Figure 3.7 shows the energy consumption trend as we increase the use of vision methods. A vision only approach would have an energy cost of 450 mAh, while motion based only costs 150 mAh. As it was mentioned before, motion methods are three times more energy efficient than vision methods. For both datasets, $\lambda = 0.2$ seems to be the best tradeoff between accuracy/energy consumption.

Maximum energy efficiency was achieved allowing minimum use of the vision predictor by making $\lambda = 0$. With only 8% of vision usage the model achieved 64.02% accuracy on the Multimodal dataset as shown on Figure 3.7. This represents an increase of almost 4% if we compare with the motion only predictor. The benefit is that we tradeoff minimum vision usage for a huge leap in accuracy. This is only possible due to the optimal decision making behavior learned by our policy.

Precision and Recall on the DataEgo dataset with different λ values showing Motion / Vision Usage

	$\lambda = 0$ (99% / 1%)		$\lambda = 0.2$ (48% / 52%)		$\lambda = 0.4$ (42 % / 58%)		Max Pooling [62]	
(r)2-3 (r)4-5 (r)6-7 (r)8-9 Activity	Prec.	Recall	Prec.	Recall	Prec.	Recall	Prec.	Recall
Walking	**0.79**	0.84	0.73	**0.88**	0.73	**0.88**	0.68	0.75
Walking Up/downstairs	**0.74**	**0.35**	0.39	0.27	0.67	0.29	0.64	**0.35**
Chopping Food	0.026	0.017	**1.0**	**0.85**	1.0	0.83	0.88	0.80
Riding Elevators	0.18	0.0434	0.42	0.13	0.54	0.17	**0.58**	**0.20**
Brushing Teeth	0.05	0.073	0.14	**1.0**	0.14	**1.0**	**0.20**	0.99
Riding Escalators	0.08	0.03	0.62	0.41	0.71	0.52	**0.78**	**0.60**
Talking with people	0.46	0.70	0.46	0.70	**0.5**	**0.71**	**0.5**	0.70
Watching TV	0.29	0.33	**0.29**	**0.33**	0.28	**0.33**	0.22	0.28
Eating and Drinking	0.49	0.61	0.82	0.97	0.87	**0.98**	**0.88**	0.94
Cooking on Stove	0.19	0.16	**0.94**	**0.94**	0.90	**0.94**	0.90	0.91
Browsing Mobile Phone	0.16	0.28	**0.96**	0.91	0.94	0.92	0.95	**0.97**
Washing dishes	0.21	0.78	**0.81**	0.94	0.76	**1.0**	0.80	1.0
Working on PC	0.30	0.5	0.82	0.58	**0.86**	0.69	0.84	**0.72**
Reading	0.26	0.10	**0.95**	0.85	**0.95**	**0.87**	0.90	0.78
Writing	0.28	0.26	**0.96**	**0.93**	0.90	0.89	0.83	0.83
Lying Down	**0.98**	**0.96**	0.91	**0.96**	0.91	**0.96**	0.76	0.50
Running	0.96	0.92	0.96	0.90	0.94	0.84	**0.97**	**1.0**
Doing push ups	0.69	0.79	0.97	0.89	0.97	0.88	**0.99**	**0.96**
Doing sit ups	0.89	0.82	0.88	0.82	0.88	0.81	**1.0**	**1.0**
Cycling	0.69	0.81	**0.78**	**0.87**	0.64	0.82	0.75	0.83

TABLE 3.2: Results on the DataEgo dataset shows that our method has better performance when compared to previous work.

The parameter λ allows our method to learn policies with both low energy profile and with high predictive capacity. This can be achieved through proper parameter tunning. For instance, predictive performance is improved as we increase the value of λ while energy consumption is reduced as we pick smaller values.

The results on large values of λ have shown promising results. With 95% of vision usage and only 5% of motion, the overall accuracy on the Multimodal dataset was 84.84%. This is the highest accuracy achieved in this dataset to date. The policy outperforms the vision only model by almost 5%. Not only we outperform previous work [62] but we also provide a more energy efficient predictor as 5% of all actions comes from a low energy source. It is worth to note that the benchmarked methods require the camera to be on at all times as they rely on its data to make continuous predictions.

The impact of λ was quite different between datasets as can be observed on Figure 3.7. This is due to a lower predictive capacity of the LRCN on the DataEgo dataset. As the difference of accuracy between predictors decreases, our method requires smaller values of λ in order to balance their usage. Therefore, when choosing a value of λ, the predictors accuracy should be taken into consideration as it is directly related to the magnitude of λ. Moreover, deterministic policies were in fact learned when λ values were too high or too low. For instance, λ = 0 provided 99% of motion based predictions usage and only 1% of vision based ones while using the Multimodal dataset.

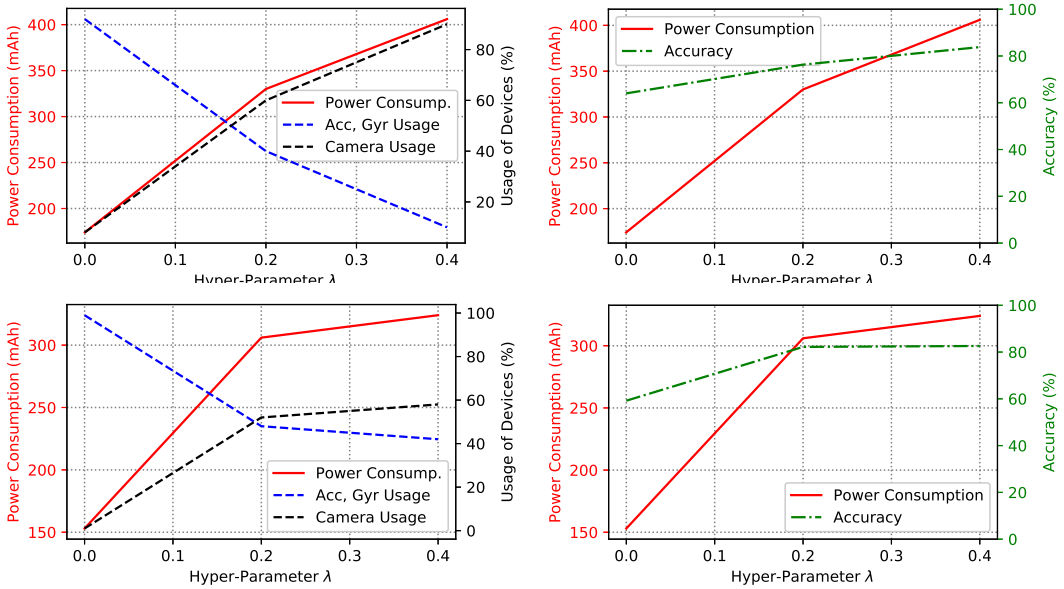


FIGURE 3.7: Summary of Accuracy, Energy Consumption and Sensor/Vision usage for both Multimodal (**top**) and DataEgo (**bottom**) datasets.

Table 3.2 compares the max pooling method from previous work [62] on the DataEgo dataset. The dataset presents a more complicated problem as now the model needs to also learn activity transitions. Benchmarked methods performance is worse than ours with $\lambda = 0.4$. This shows that our framework is more suitable on situations where a more realistic setting is used. Our policy seems to suffer less from activity transition noise as it provides better overall accuracy.

3.5 Summary

We presented a novel RL framework for the Egocentric Activity Recognition problem. Our method was able to achieve state-of-the-art performance on current multi-stream datasets while saving energy by trading off vision-based activity recognition with low power motion based sensor. This approach attempts to be more realistic for practical implementation on wearable devices as these devices have limited computational and energy resources and therefore are not able to not only run expensive deep learning models but also keep the device's camera on for extensive periods of time. Although we have presented promising results for battery optimisation, this would still be limited to real world situations as the battery was designed to operate in its optimal state. For instance, in a day where temperatures are above the manufacture threshold, the discharge rate can be higher due to overheating, hence we would

like the algorithm to be even more conservative. At its current state, we are not incorporating such variables into learning the optimal controller, hence it would make decisions as if it was operating on its standard temperature, leading to possible poor energy performance. In the next chapters we propose a general algorithm to overcome the reality gap problem.

CHAPTER 4

BayesSim: adaptive domain randomization via probabilistic inference[1] [2]

4.1 Introduction

In the previous chapter we have seen that learning effective controllers for use in the real world is dependent on knowing how some of the physics behave. More specifically, in order to calculate battery discharge rate, several environmental factors could affect the actual results, for instance, if the battery gets too hot very quickly it would discharge in a higher rate than what calculated by the standard formula. The controller that optimises battery consumption needs to take into consideration the uncertainty of the inherent system in order to truly learn an optimal policy that generalises well to the real world changes. In order to achieve this, the controller needs to be optimised in the simulated environment where it is exposed to a range of different environment configurations. In this chapter, we introduce BayesSim, a principled framework that allows a full Bayesian treatment for the parameters of a simulator. As simulators become more sophisticated and able to represent the dynamics more accurately, fundamental problems in related areas can be more efficiently transferred to the real world. However, even the most complex simulator might still not be able to represent reality in all its details either due to inaccurate parametrisation or simplistic assumptions in the dynamic models. BayesSim provides a principled framework to reason about the uncertainty of simulation parameters. Given a black box simulator (or generative model) that outputs trajectories of state and action pairs from unknown simulation parameters, followed by trajectories obtained with a physical robot, we develop a likelihood-free inference method that computes the posterior distribution of simulation parameters. This posterior can then be used in problems where Sim2Real is critical, for example in policy search. For benchmarking reasons, we compare the performance of BayesSim in obtaining accurate posteriors in a number of classical control and robotics problems. Results show that the posterior computed from BayesSim can be used for domain randomization outperforming alternative methods that randomize based on uniform priors.

[1] Contents of this Chapter were published in RSS 2019

[2] https://arxiv.org/abs/1906.01728

This chapter proposes a principled Bayesian method that computes full posteriors over simulator parameters, hence, helping to overcome limitations when learning controllers that are deployed in real life environments. The technique is called BayesSim, it leverages recent advances in likelihood-free inference in order to provide updates of posteriors over simulation parameters. This framework helps to reduce the Sim2Real gap when training controllers for a wide range of tasks. In order to capture reality in all its complexity BayesSim collects a small set of observations obtained from the real world, and, then, this information is used to calibrate posteriors that were learned previously in simulation. The main contribution of this chapter is a generic framework for probabilistic inference that can be used in a wide range of real-world simulation that provides a full space of parameters that best fit observed data. We also propose a novel mixture density random Fourier network to approximate the conditional distribution $p(\boldsymbol{\theta}|\mathbf{x}^r)$ directly by learning from pairs $\{\boldsymbol{\theta}_i, \mathbf{x}_i^s\}_{i=1}^N$ generated from the proposal prior and the simulator.

4.2 Related Work

Various methods have been proposed to learn system dynamics from time series data of real systems. Early works include Locally Weighted Regression [95] and Forward Models [96]. Modern "intuitive physics" models often use deep graph neural networks to discover constraints between particles or bodies [97, 98, 99, 100]. Physics-based machine learning approaches introduce a physics-informed inductive bias to the problem of learning models for dynamical systems from data [101, 102, 103, 104].

Simulators accelerate machine learning impact by allowing faster, highly-scalable and low cost data collection. Many other scientific domains such as economics [105], evolutionary biology [57] and cosmology [106] also rely on simulator-based modelling to provide further advancements in research. In robotics, "reality gap" is not only seen in control, robotics vision is also affected by this problem [107]. Algorithms trained on images from a simulation can frequently fail on different environments as the appearance of the world can differ greatly from one system to the other.

Randomizing the dynamics of a simulator while training a control policy has proven to mitigate the reality gap problem [12]. Simulation parameters could vary from physical settings like damping, friction and object masses [12] to visual parameters like objects textures, shapes and etc [107]. Another similar approach is that of adding noise to the system parameters [108] instead of sampling new parameters from a uniform prior distribution. Perturbation can also be seen on robot locomotion [109] where planning is

done through an ensemble of perturbed models. Lastly, interleaving policy roll outs between simulation and reality has also proven to work well on swing-peg-in-hole and opening a cabinet drawer tasks [110].

Originating from traditional physics engines (examples include [111, 112, 113]), differentiable simulators have been introduced that leverage automatic, symbolic or implicit differentiation to calculate parameter gradients through the analytical Newton-Euler equations and contact models for rigid-body dynamics [114, 115, 116, 117, 118]. Simulations for other physical processes, such as light propagation [39], [40] and continuum mechanics [119, 120, 121, 122, 123] have been made differentiable as well. Learning models from simulations of data can leverage one's understanding of the physical world potentially helping to solve the aformentioned problem.

Until recently, Approximate Bayesian Computation [57] has been one of the main methods used to tackle this type of problem. Rejection ABC [124] is the most basic method where parameter settings are accepted/rejected if they are within a certain specified range. The set of accepted parameters approximates the posterior for the real parameters. Markov Chain Monte Carlo ABC (MCMC-ABC) [58] improves over its precedent by perturbing accepted parameters instead of independently proposing new parameters. Lastly, Sequential Monte Carlo ABC (SMC-ABC) [125] leverages sequential importance sampling to simulate slowly-change distributions where the last one is an approximation of the true parameter posterior. In this work, we use a ϵ-free approach [126] for likelihood-free inference, where a Mixture of Density Random Fourier Network estimates the parameters of the true posterior through a Gaussian mixture.

A wide range of complex robotics control problems have been recently solved using Deep Reinforcement Learning (Deep RL) techniques [12, 108, 127]. Classic control problems like Pendulum, Mountain Car, Acrobot and Cartpole have been successfully tackled using policy search with algorithms like Trust Region Policy Optimization (TRPO) [50] and Proximal Policy Optimization (PPO) [55]. More complex tasks in robotics such as the ones in manipulation are still difficult to solve using traditional policy search. Both Push and Slide tasks on the fetch robot [128] were only solved recently using the combination of Deep Deterministic Policy Gradients (DDPG) [47] and Hindsight Experience Replay (HER) [10].

4.3 BayesSim

This section provides the detailed definition of the framework proposed in this chapter. It relies heavily on previous knowledge of likelihood-free inference and Reinforcment Learning as it was provided in the the Literature Review of this thesis.

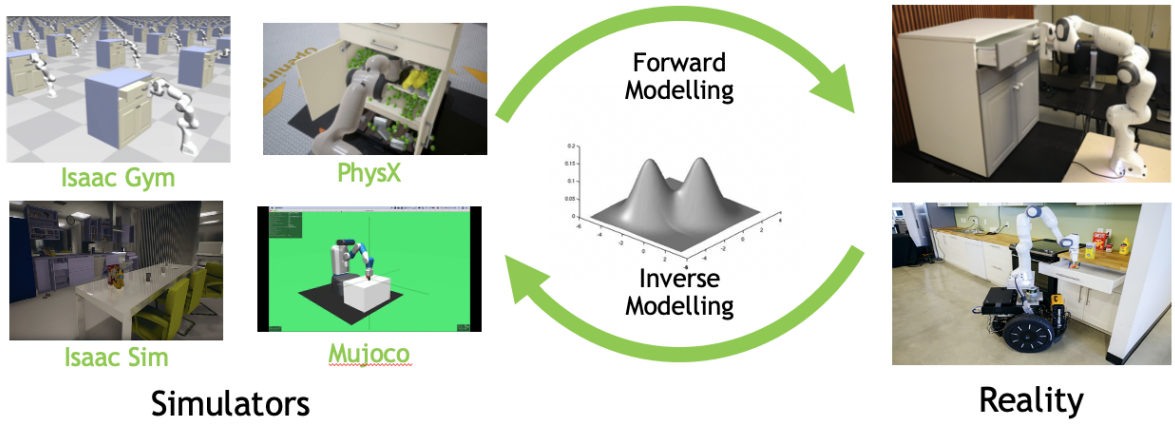


FIGURE 4.1: BayesSim can leverage different simulators at the same time.

4.4 Problem Setup

Following the work in [126], BayesSim approximates the intractable posterior $p(\boldsymbol{\theta}|\mathbf{x} = \mathbf{x}^r)$ by directly learning a conditional density $q_\phi(\boldsymbol{\theta}|\mathbf{x})$ parameterised by parameters ϕ. As we shall see, $q_\phi(\boldsymbol{\theta}|\mathbf{x})$ takes the form of a mixture density random feature network. To learn the parameters ϕ we first generate a dataset with N pairs $(\boldsymbol{\theta}_n, \mathbf{x}_n)$ where $\boldsymbol{\theta}_n$ is drawn independently from a distribution $\tilde{p}(\boldsymbol{\theta})$ referred to as the *proposal prior*. $\mathbf{x}_n$ is obtained by running the simulator with parameter $\boldsymbol{\theta}_n$ such that $\mathbf{x}_n = g(\boldsymbol{\theta}_n)$. In [126] the authors show that $q_\phi(\boldsymbol{\theta}|\mathbf{x})$ is proportional to $\frac{\tilde{p}(\boldsymbol{\theta})}{p(\boldsymbol{\theta})} p(\boldsymbol{\theta}|\mathbf{x})$ when the likelihood $\prod_n q_\phi(\boldsymbol{\theta}_n|\mathbf{x}_n)$ is maximised w.r.t. ϕ. We follow a similar procedure and maximise the log likelihood,

$$L(\phi) = \frac{1}{N} \sum_n \log q_\phi(\boldsymbol{\theta_n}|\mathbf{x}_n) \tag{4.1}$$

to determine ϕ. After this is done, an estimate of the posterior is obtained by

$$\hat{p}(\boldsymbol{\theta}|\mathbf{x} = \mathbf{x}^r) \propto \frac{p(\boldsymbol{\theta})}{\tilde{p}(\boldsymbol{\theta})} q_\phi(\boldsymbol{\theta}|\mathbf{x} = \mathbf{x}^r), \tag{4.2}$$

where $p(\boldsymbol{\theta})$ is the desirable prior that might be different than the proposal prior. In the case when $\tilde{p}(\boldsymbol{\theta}) = p(\boldsymbol{\theta})$, it follows that $\hat{p}(\boldsymbol{\theta}|\mathbf{x} = \mathbf{x}^r) \propto q_\phi(\boldsymbol{\theta}|\mathbf{x} = \mathbf{x}^r)$. When $\tilde{p}(\boldsymbol{\theta}) \neq p(\boldsymbol{\theta})$ we need to adjust the posterior as it will be detailed later.

4.4.1 Mixture density random feature networks

We model the conditional density $q_\phi(\boldsymbol{\theta}|\mathbf{x})$ as a mixture of K Gaussians,

$$q_\phi(\boldsymbol{\theta}|\mathbf{x}) = \sum_k \alpha_k N(\boldsymbol{\theta}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k), \tag{4.3}$$

where $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_K)$ are mixing coefficients, $\{\mu_k\}$ are means and $\{\Sigma_k\}$ are covariance matrices. This is analogous to mixture density networks [13] except that we replace the feedforward neural network with Quasi Monte Carlo (QMC) random Fourier features when computing $\boldsymbol{\alpha}$, $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. We justify and describe these features in the next section.

Denoting $\Phi(\boldsymbol{x})$ as the feature vector, the mixing coeficients are calculated as

$$\boldsymbol{\alpha} = \text{softmax}(\mathbf{W}_{\boldsymbol{\alpha}}\Phi(\mathbf{x}) + \mathbf{b}_{\boldsymbol{\alpha}}). \tag{4.4}$$

Note that the operator $\text{softmax}(\mathbf{z}_i) = \frac{\exp(z_i)}{\sum_{k=1}^K \exp z_k}$ for $i = 1, \ldots, K$ enforces that the sum of coeficients equals to 1 and each coefficient is between 0 and 1.

The means are defined as linear combinations of feature vectors. For each component of the mixture,

$$\boldsymbol{\mu}_k = \mathbf{W}_{\boldsymbol{\mu}_k}\Phi(\mathbf{x}) + \mathbf{b}_{\boldsymbol{\mu}_k}. \tag{4.5}$$

Finally we parametrize the covariance matrices as diagonals matrices with

$$\text{diag}(\boldsymbol{\Sigma}_k) = \text{mELU}(\mathbf{W}_{\boldsymbol{\Sigma}_k}\Phi(\mathbf{x}) + \mathbf{b}_{\boldsymbol{\Sigma}_k}) \tag{4.6}$$

where mELU is a modified exponential linear unit defined as

$$mELU(z) = \begin{cases} \alpha(e^z - 1) + 1 & \text{for } z \leq 0 \\ z + 1 & \text{for } z > 0 \end{cases} \tag{4.7}$$

to enforce positive values. Experimentally this parametrization provided slightly better results than with the exponential function. The diagonal parametrization assumes independence between the dimensions

of the simulator parameters $\boldsymbol{\theta}$. This turns out to be not too restrictive if the number of components in the mixture is large enough.

The full set of parameters for the mixture density network is then,

$$\phi = (\mathbf{W}_{\boldsymbol{\alpha}}, \mathbf{b}_{\boldsymbol{\alpha}}, \{\mathbf{W}_{\boldsymbol{\mu}_k}, \mathbf{b}_{\boldsymbol{\mu}_k}, \mathbf{W}_{\boldsymbol{\Sigma}_k}, \mathbf{b}_{\boldsymbol{\Sigma}_k}\}_{k=1}^{K}). \tag{4.8}$$

4.4.2 Neural Network features

BayesSim can use neural network features creating a model similar to the mixture density network as shown by seminal work of [13]. For a feedforward neural network with two fully connected layers, the features take the form

$$\Phi(\mathbf{x}) = \sigma(\mathbf{W}_2(\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)) + \mathbf{b}_2), \tag{4.9}$$

where $\sigma(\cdot)$ is a sigmoid function; we use $\sigma(\cdot) = \tanh(\cdot)$ in our experiments. This network structure was used in the experiments and compared to the Quasi Monte Carlo random features described below.

4.4.3 Quasi Monte Carlo random features

BayesSim can use random Fourier features [129] instead of neural nets to parameterise the mixture density. There are several reasons why this can be good choice. Notably, 1) random Fourier features – of which QMC features are a particular type – approximate possibly infinite Hilbert spaces with properties defined by the choice of the associated kernel. In this way prior information about properties of the function space can be readily incorporated by selecting a suitable positive semi-definite kernel; 2) the approximation converges to the original Hilbert space with order $O(1/\sqrt{s})$, where s is the number of features, therefore independent of the input dimensionality; 3) experimentally, we verified that mixture densities with random Fourier features are more stable to different initialisations and converge to the same local maximum in most cases.

Random Fourier features approximate a shift invariant kernel $k(\boldsymbol{\tau})$, where $\boldsymbol{\tau} = \mathbf{x} - \mathbf{x}'$, by a dot product $k(\boldsymbol{\tau}) \approx \Phi(\mathbf{x})^T\Phi(\mathbf{x})$ of finite dimensional features $\Phi(\mathbf{x})$. This is possible by first applying the Bochner's theorem [130] stated below:

Theorem 1 *(Bochner's Theorem) A shift invariant kernel $k(\boldsymbol{\tau})$, $\boldsymbol{\tau} \in \mathbb{R}^D$, associated with a positive finite measure $d\mu(\boldsymbol{\omega})$ can be represented in terms of its Fourier transform as,*

$$k(\boldsymbol{\tau}) = \int_{\mathbb{R}^D} e^{-i\boldsymbol{\omega}\cdot\boldsymbol{\tau}} d\mu(\boldsymbol{\omega}). \tag{4.10}$$

The proof can be found in [131]. When μ has density $(\boldsymbol{\omega})$ then K represents the spectral distribution for a positive semi-definite k. In this case $k(\boldsymbol{\tau})$ and $K(\boldsymbol{\omega})$ are Fourier duals:

$$k(\boldsymbol{\tau}) = \int K(\boldsymbol{\omega}) e^{-i\boldsymbol{\omega}\cdot\boldsymbol{\tau}} d\boldsymbol{\omega}. \tag{4.11}$$

Approximating Equation 4.11 with a Monte Carlo estimate with N samples, yields

$$k(\boldsymbol{\tau}) \approx \frac{1}{N} \sum_{n=1}^{N} (e^{-i\boldsymbol{\omega}_n \mathbf{x}})(e^{-i\boldsymbol{\omega}_n \mathbf{x}'}), \tag{4.12}$$

where $\boldsymbol{\omega}$ is sampled from the density $K(\boldsymbol{\omega})$.

Finally, using Euler's formula ($e^{-ix} = \cos(x) - i\sin(x)$) we recover the features:

$$\begin{aligned} \Phi(\mathbf{x}) = \frac{1}{\sqrt{N}} [&\cos(\boldsymbol{\omega}_1 \mathbf{x} + b_1), \ldots, \cos(\boldsymbol{\omega}_n \mathbf{x} + b_n), \\ &-i \cdot \sin(\boldsymbol{\omega}_1 \mathbf{x} + b_1), \ldots, -i \cdot \sin(\boldsymbol{\omega}_n \mathbf{x} + b_n)]. \end{aligned} \tag{4.13}$$

where bias terms $\mathbf{b}_i$ are introduced with the goal of rotating the projection and allowing for more flexibility in capturing the correct frequencies.

This approximation can be used with all shift invariant kernels proving flexibility in introducing prior knowledge by selecting a suitable kernel for the problem. For example, the RBF kernel can be approximated using the features above with $\boldsymbol{\omega} \sim N(0, 2\sigma^{-2} I)$ and $b \sim U[-\pi, \pi]$. σ is a hyperparameter that corresponds to the kernel length scale and is usually set up with cross validation.

We further adopt a quasi Monte Carlo strategy for sampling the frequencies. In particular we use Halton sequences proposed in [132] which has been shown in [133] to have better convergence rate and lower approximation error than standard Monte Carlo.

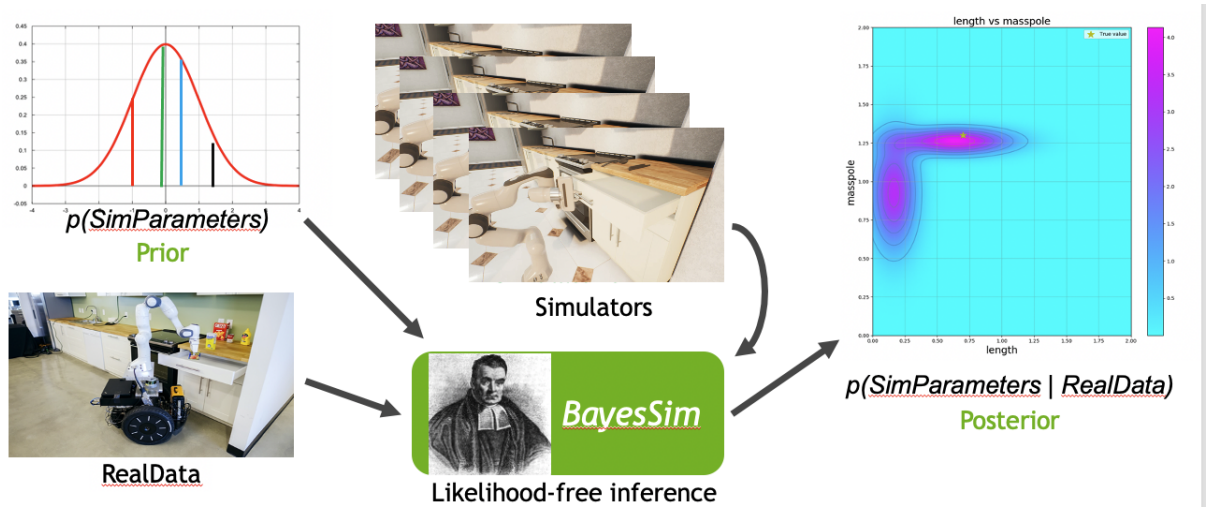


FIGURE 4.2: BayesSim framework for posterior recovery.

4.4.4 Posterior recovery

From Equation 4.2 we note that when the proposal prior is different than the desirable prior, we need to adjust the posterior by weighting it with the ratio $p(\boldsymbol{\theta})/\tilde{p}(\boldsymbol{\theta})$.

In this paper we assume the prior to be uniform, either with finite support – defined within a range and zero elsewhere – or improper, constant value everywhere. Therefore,

$$\hat{p}(\boldsymbol{\theta}|\mathbf{x}=\mathbf{x}^r) \propto \frac{q_\phi(\boldsymbol{\theta}|\mathbf{x}^r)}{\tilde{p}(\boldsymbol{\theta})}. \tag{4.14}$$

When the proposal prior is Gaussian, we can compute the division between a mixture and a single Gaussian analytically. In this case, since $q_\phi(\boldsymbol{\theta}|\mathbf{x})$ is a mixture of Gaussians and $\tilde{p}(\boldsymbol{\theta}) \sim N(\boldsymbol{\theta}|\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)$, the solution is given by

$$\hat{p}(\boldsymbol{\theta}|\mathbf{x}=\mathbf{x}^r) = \sum_k \alpha'_k N(\boldsymbol{\theta}|\boldsymbol{\mu}'_k, \boldsymbol{\Sigma}'_k) \tag{4.15}$$

where,

$$\boldsymbol{\Sigma}'_k = \left(\boldsymbol{\Sigma}_k^{-1} - \boldsymbol{\Sigma}_0^{-1}\right)^{-1} \tag{4.16}$$

$$\boldsymbol{\mu}'_k = \boldsymbol{\Sigma}_k^{-1}\left(\boldsymbol{\Sigma}_k^{-1}\boldsymbol{\mu}_k - \boldsymbol{\Sigma}_0^{-1}\boldsymbol{\mu}_0\right) \tag{4.17}$$

$$\alpha'_k = \frac{\alpha_k \exp(-\frac{1}{2}\lambda_k)}{\sum_{k'} \alpha_{k'} \exp(-\frac{1}{2}\lambda_{k'})}, \tag{4.18}$$

and the coefficients λ_k are given by

$$\lambda_k \;=\; \log\det\boldsymbol{\Sigma}_k \;-\; \log\det\boldsymbol{\Sigma}_0 \;-\; \log\det\boldsymbol{\Sigma}'_k \;+\; \boldsymbol{\mu}_k^T\boldsymbol{\Sigma}_k^{-1}\boldsymbol{\mu}_k \;-\; \boldsymbol{\mu}_0^T\boldsymbol{\Sigma}_0^{-1}\boldsymbol{\mu}_0 \;-\; \boldsymbol{\mu}_k'^T\boldsymbol{\Sigma}_k'^{-1}\boldsymbol{\mu}'_k. \quad (4.19)$$

4.4.5 Sufficient statistics for state-action trajectories

Trajectories of state and action pairs in typical problems can be long sequences making the input dimensionality to the model prohibitive large and computationally expensive. We adopt a strategy commonly used in ABC; instead of inputting raw state and action sequences to the model, we first compute some sufficient statistics. Formally, $\mathbf{x} = \psi(\mathbf{S}, \mathbf{A})$ where $\mathbf{S} = \{\mathbf{s}^t\}_{t=1}^T$ and $\mathbf{A} = \{\mathbf{a}^t\}_{t=1}^T$ are sequences of states and actions from $t = 1$ to T. There are many options in the literature for sufficient statistics for time series or trajectory data. For example, the mean, log variance and autocorrelation for each time series as well as cross-correlation between two time series. Another possibility is to learn these from data, for example with an unsupervised encoder-decoder recurrent neural network [134]. However, such a representation would need to be trained with simulated trajectories and might not be able to capture complexities in the real trajectories. This will be investigated in future work. Here we adopt a simpler strategy and use statistics commonly applied to stochastic dynamic systems such as the Lotka-Volterra model studied in [135].

Defining $\boldsymbol{\tau} = \{\mathbf{s}^t - \mathbf{s}^{t-1}\}_{t=1}^T$ as the difference between immediate future states and current states, the statistics

$$\psi(\mathbf{S}, \mathbf{A}) = (\{\langle\boldsymbol{\tau}_i, \mathbf{A}_j\rangle\}_{i=1,j=1}^{D_s,D_a}, \mathrm{E}[\boldsymbol{\tau}], \mathrm{Var}[\boldsymbol{\tau}]), \quad (4.20)$$

where D_s is the dimensionality of the state space, D_a is the dimensionality of the action space, $\langle\cdot,\cdot\rangle$ denotes the dot product, $\mathrm{E}[\cdot]$ is the expectation, and $\mathrm{Var}[\cdot]$ the variance.

4.4.6 Example: CartPole posterior

We provide a simple example to demonstrate the algorithm in estimating unknown simulation parameters for the famous CartPole problem. In this problem a pole installed on a cart needs to be balanced by applying forces to the left or to the right of the cart. For this example we assume that both the mass and the length of the pole are not available and we use BayesSim to obtain the posterior for these parameters. We assume uniform priors for both parameters and collect 1000 simulations following a

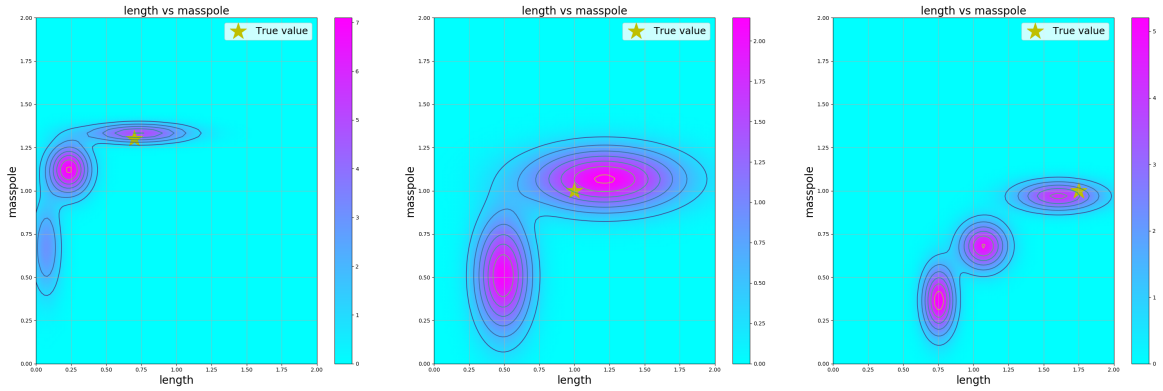


FIGURE 4.3: Example of joint posteriors obtained for the CartPole problem with different parametrizations for `length` and `masspole`. The true value is indicated by a star. Note that the joint posteriors capture the multimodality of the problem when two or more explanations seem likely, for example, a longer pole `length` with a lighter `masspole` or vice versa.

rl-zoo policy [3] to train BayesSim. With the model trained, we collected 10 trajectories with the correct parameters to simulate the real observations. Figure 4.3 shows the posteriors for both problems. As with many problems involving two related variables, `masspole` and pole `length` exhibit statistical dependencies that generate multiple explanations for their values. For example, the pole might have lower mass and longer length, or vice versa. BayesSim is able to recover the multi-modality nature of the posterior providing densities that represent the uncertainty of the problem accurately.

4.4.7 Domain randomization with BayesSim

Here we describe the domain randomization strategy to take full advantage of the posterior obtained by the inference method. Given the posterior obtained from the simulation parameters $\hat{p}(\boldsymbol{\theta}|\mathbf{x} = \mathbf{x}^r)$ we maximize the objective,

$$J(\boldsymbol{\beta}) = \mathbb{E}_{\boldsymbol{\theta}} \left[\mathbb{E}_{\boldsymbol{\eta}} \left[\sum_{t-0}^{T-1} \gamma^{(t)} r(\mathbf{s}_t, \mathbf{a}_t) | \boldsymbol{\beta} \right] \right], \tag{4.21}$$

where $\boldsymbol{\theta} \sim \hat{p}(\boldsymbol{\theta}|\mathbf{x} = \mathbf{x}^r)$ with respect to the policy parameters $\boldsymbol{\beta}$. Since the posterior is a mixture of Gaussians, the first expectation can be approximated by sampling a mixture component following the distribution over $\boldsymbol{\alpha}$ to obtain a component k, followed by sampling the corresponding Gaussian $N(\boldsymbol{\theta}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$.

[3] https://github.com/araffin/rl-baselines-zoo

4.5 Experiments

Experiments are presented in two different cases to demonstrate and assess the performance of BayesSim. In Section 4.5.1 we verify and compare the accuracy of the posterior recovered. In Section 4.5.2 we compare the robustness of policies trained by randomizing following the prior versus posterior distribution over simulation parameters.

4.5.1 Posterior recovery

The first analysis we carry out is the quality of the posteriors obtained for different problems and methods. We use the log probability of the target under the mixture model as the measure, defined as $\log p(\boldsymbol{\theta}_*|\mathbf{x} = \mathbf{x}^r)$, where $\boldsymbol{\theta}_*$ is the actual value for the parameter. We compare Rejection-ABC [124] as the baseline, the recent ϵ-Free [126] which also provides a mixture model as the posterior, and BayesSim using either a two layer neural network with 24 units in each layer, and BayesSim with quasi random Fourier Features. For the later we use the Matern 5/2 kernel [136] and set up the the sampling precision σ by cross validation. Three different simulators were used for different problems; OpenAI Gym [137], PyBullet [4], and MuJoCo [112]. Finally, the following problems were considered; CartPole (Gym), Pendulum (Gym), Mountain Car (Gym), Acrobot (Gym), Hopper (PyBullet), Fetch Push (MuJoCo) and Fetch Slide (MuJoCo). For all configurations of methods and parameters, training and testing were performed 5 times with the log probabilities averaged and standard deviation computed. To extract the real observations, we simulate the environments with the actual parameters 10 times and average the sufficient statistics to obtain $\mathbf{x}^r$. In all cases we collect sufficient statistics by performing rollouts for either a maximum of 200 time steps or until the end of the episode.

Table 5.1 shows the results (means and standard deviations) for the log probabilities. BayesSim with either RFF or Neural Network features provides generally higher log-probabilities and lower standard deviation than Rejection ABC. This indicates that the posteriors provided by BayesSim are more peaked and centered around the correct values for the parameters. Compared to ϵ-Free, the results are equivalent in terms of the means but BayesSim generally provides lower standard deviation across multiple runs of the method, indicating it is more stable than ϵ-Free. Comparing BayesSim with RFF and NN, the RFF features lead to higher log probabilities in most cases but BayesSim with neural networks have lower standard deviation.

[4] https://pypi.org/project/pybullet/

Problem	Parameter	Uniform prior	Rejection ABC	ϵ-Free	BayesSim RFF	BayesSim NN
CartPole	pole length	$[0.1, 2.0]$	-0.342±0.15	**-0.211±0.07**	-0.609±0.39	-0.657±0.25
	pole mass	$[0.1, 2.0]$	0.032±0.21	0.056±0.14	**0.973 ± 0.26**	0.633± 0.52
Pendulum	dt	$[0.01, 0.3]$	2.101±1.04	2.307±0.84	3.192±0.30	**3.199±0.17**
Mountain Car	power	$[0.0005, 0.1]$	3.69±1.21	3.800±1.06	3.863±0.52	**3.901±0.2**
Acrobot	link mass 1	$[0.5, 2.0]$	1.704±0.82	1.883±0.79	**2.046±0.37**	1.331±0.22
	link mass 2	$[0.5, 2.0]$	1.832±0.93	**2.237±0.76**	0.321±1.85	1.513±0.39
	link length 1	$[0.1, 1.5]$	**2.421±0.75**	2.135±0.50	2.072±0.76	1.856±0.18
	link length 2	$[0.5, 1.5]$	-0.521±0.36	-0.703±0.16	**-0.148±0.19**	-0.672±0.09
Hopper	lateral friction	$[0.3, 0.5]$	3.032±0.43	3.154±0.81	2.622±0.64	**3.391±0.08**
Fetch Push	friction	$[0.1, 1.0]$	1.332±0.54	2.013±0.09	**2.423±0.07**	2.404±0.05
Fetch Slide	friction	$[0.1, 1.0]$	1.014±0.38	1.614±0.12	**2.391±0.06**	2.111±0.03

TABLE 4.1: Mean and standard deviation of log predicted probabilities for several likelihood-free methods, applied to seven different problems and parameters.

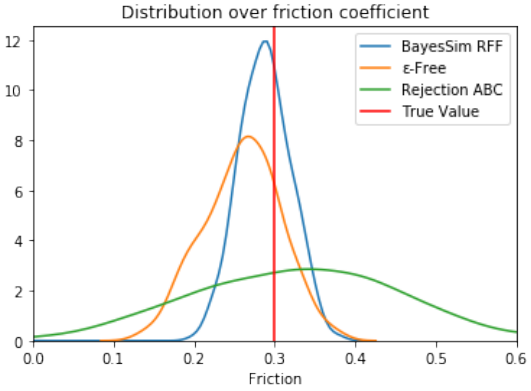


FIGURE 4.4: Posteriors recovered by different methods for the Fetch slide problem. Note that BayesSim with random features provides a posterior that is more peaked around the true value.

These results suggest that BayesSim with either RFF or NN is comparable to the state-of-art, and in many cases superior when estimating the posterior distribution over the simulation parameters. For the robotics problems analyzed in the next section, however, BayesSim with RFF provide significant superior results than the other methods and slightly better than BayesSim with NN. This can be better observed when we plot the posteriors in Figure 4.4. BayesSim RFF is significantly more peaked and centered around the true friction value.

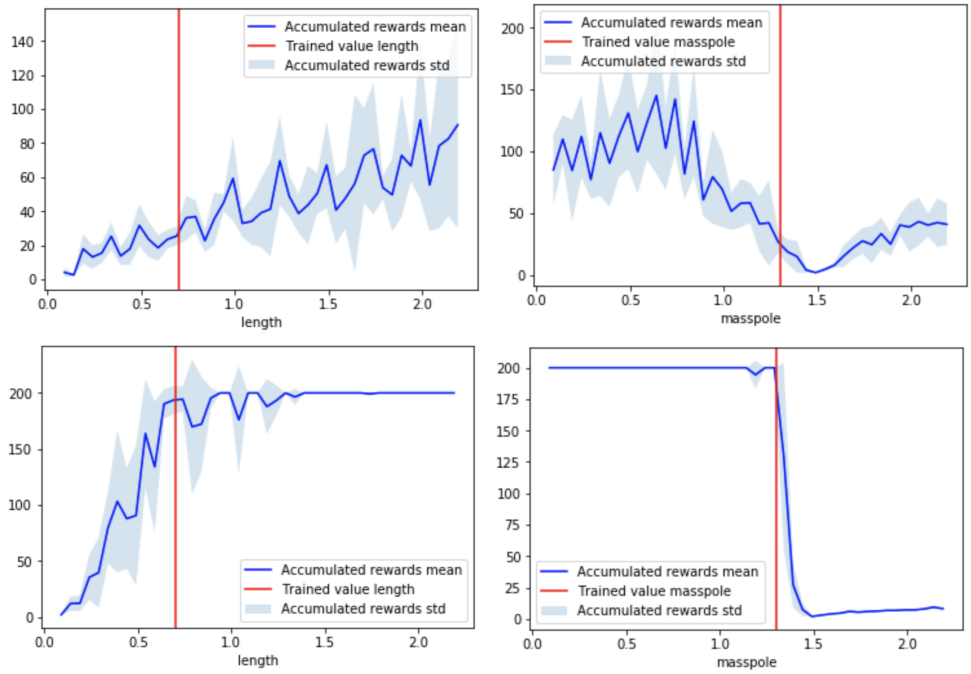


FIGURE 4.5: Accumulated rewards for CartPole policies trained with PPO by randomizing over prior and posterior joint densities. Top left: Performance of the policy trained with the prior, over parameter `length`. `masspole` is set to actual. Top right: Similar to top left, but over multiple `masspole` values. Bottom left: Performance of policy trained with the posterior, over parameter `length`. Bottom right: Similar to bottom left, but over multiple `masspole` values.

4.5.2 Robustness of policies

We evaluate robustness of policies by comparing their performance on the uniform prior and the learned posterior provided by BayesSim. Evaluation is done over a pre-defined range of simulator settings and the average reward is shown for each parameter value.

In the first set of experiments we use the CartPole problem as a simple example to illustrate the benefits of posterior randomization. We trained two policies, the first randomizing with a uniform prior for *length* and *masspole* as indicated in Table 5.1. The second, randomized based on the posterior provided by BayesSim with RFF. In both cases we use PPO to train the policies with 100 samples from the prior and posterior, for 2M timesteps. The results are presented in Figure 4.5, averaged over several runs with the corresponding standard deviations. It can be observed that randomization over the posterior yields a significantly more robust policy, in particular at the actual parameter value. Also noticeable is the reduction in performance for lower *length* values and higher *masspole* values. This is expected as it is more difficult to control the pole position when the length is short due to the increased dynamics of

the system. Similarly, when the mass increases too much, beyond the value it was actually trained on, the controller struggles to maintain the pole balanced. Importantly, the policy learned with the posterior seems much more stable across multiple runs as indicated by the lower variance in the plots.

In the second set of experiments we use a Fetch robot available in OpenAI Gym [128] to perform both push and slide tasks. The first is a closed loop scenario, where the arm is always in range of the entire table and, hence, it can correct its trajectories according to the input it receives from the environment. The second is a more difficult open loop scenario, where the robot has usually only one shot at pushing the puck to its desired target. For both tasks, the friction coefficient of the object and the surface plays a major role in the final result as they are strictly related to how far the object goes after each force is applied. A very low friction coefficient means that the object is harder to control as it slides more easily and a very high one means that more force needs to be applied in order to make the object to move.

Our goal is to recover a good approximation of the posterior over friction coefficients using BayesSim. Initially, we need to learn a policy with a fixed friction coefficient that will be used for data generation purposes. We train this policy using DDPG with experiences being sampled using HER for 200 epochs with 100 episodes/rollouts per epoch. Gradient updates are done using Adam with step size of 0.001. We then run this policy multiple times with different friction coefficients in order to approximate the likelihood function and recover the full posterior over simulation parameters. With the dynamics model in hand, we can finally recover the desired posterior using some data sampled from the environment we want to learn the dynamics from. Training is carried out using the same aforementioned settings but instead of using a fixed friction coefficient, we sample a new one from its respective distribution every time a new episode starts.

The results from both tasks are presented on Figure 5.3. As it has been shown in previously work of [12], the uniform prior works remarkably well on the push task. This happens as the robot has the opportunity to correct its trajectory whether something goes wrong. As it has been exposed to a wide range of scenarios involving different dynamics, it can then use the input of the environment to perform corrective actions and still be able to achieve its goal. However, the results for slide task differ significantly since using a wide uniform prior has led the robot to achieve a very poor performance. This happens as not only the actions for different coefficients in most times are completely different but also because the robot has no option of correcting its trajectory. This is where methods like BayesSim are useful as it recovers a distribution with very high density around the true parameter and, hence, leads

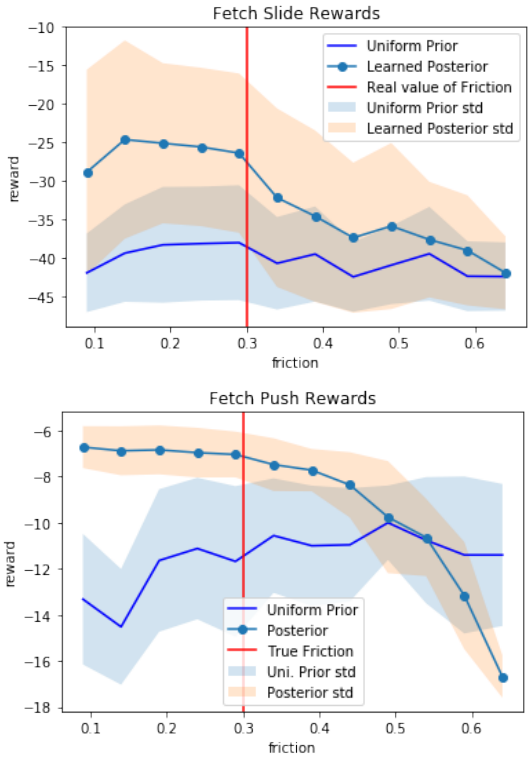


FIGURE 4.6: Comparison between policies trained on randomizing the prior vs BayesSim posterior for different values of the simulation parameter. Top: Fetch slide problem. Bottom: Fetch push problem.

to a better overall control policy. Our results shows that higher rewards are achieved around the true friction value while the uniform prior results are mostly flat throughout all values.

4.6 Summary

The method shown in this chapter represents the first step towards a Bayesian treatment of simulation parameters, combined with domain randomization for policy search. The approach is connected to system identification in that both attempt to estimate dynamic models, but ours uses a black-box generative model, or simulator, totally integrated into the framework. Prior distributions can also be provided and incorporated into the model to compute a full, potentially multi-modal posterior over the parameters. The method proposed here, BayesSim, performs comparably to other state-of-the-art likelihood-free approaches for Bayesian inference but appears more stable to different initialisations, and across multiple runs when recovering the true posterior. Finally, we show that domain randomization with the posterior

leads to more robust policies over multiple parameter values compared to policies trained on uniform prior randomization.

As typical in the likelihood-free inference literature, BayesSim relies on the definition of meaningful sufficient statistics for the trajectories of states and actions. Alternatively, a lower dimensional representation for the trajectories could be created using recent encoder-decoder methods and recurrent neural networks known to perform well for time series prediction such as LSTMs. Hence, the entire framework can be learnt end to end. This idea is explored in the next chapter as we shall see.

CHAPTER 5

Online BayesSim: Sequential Policy Optimisation and Simulator Parameter Inference[1] [2]

5.1 Introduction

Model miss-specification and noisy sensor measurements are a common problem in control systems operating in complex robotics applications. Models need to be robust to such circumstances in order to achieve desirable results. Developing powerful and realistic simulators allows practitioners to gauge the performance of controllers against such variables before its actually deployed to the real robot. Moreover, controllers, nowadays, are relying on different types of cameras to capture streams of images as an extra source of data for decision making.A range of possible improvements in simulation techniques before they can capture reality with all its complexities. "Reality gap" is a term used when the environment model used in a simulator does not represent the targeted system accurately enough so we can achieve the desirable performance when deploying a robot in the real world.

Oversimplified assumptions or insufficient numerical precision in solvers can play a major role in how well a simulator can model reality. Existing prior knowledge about simulation parameters is often incorporated through a series of trial and error experiments until some good approximation is reached. This process is inefficient and time consuming as it involves running non-optimal control strategies on expensive and fragile robots.

Estimating accurate distributions of dynamical systems can enable robust policy learning in both open-loop and closed-loop scenarios as presented in [7]. In [138], the authors explore how using distributions over the model parameters can help training more robust model based controllers when compared to those using maximum-likelihood point estimates.

[1] Contents of this Chapter were published in IROS 2020

[2] https://ieeexplore.ieee.org/document/9341401

In order to work well, there is a strong dependency of the above methods on manual specification of sufficient statistics representing trajectory roll-outs. In several scenarios, the data is either high-dimensional (e.g. images, videos) or highly correlated (e.g. time series) which makes the definition of sufficient statistics challenging. Even state-of-the-art techniques like the ones presented in [61] have still not addressed the issue of dealing with high dimensional input data. Attempts to use sufficient statistics in these cases will not provide a reasonable representation of simulation data which can drastically impact the overall performance. This, unfortunately, limits the usage of such methods to much simpler inputs.

As it has been shown on previous chapter, likelihood-free inference enables a probabilistic approach for the problem of estimating simulation parameters and their uncertainty given sequences of observations. Domain randomization can be performed much more effectively when a posterior distribution provides the correct uncertainty over parameters in a simulated environment. In this chapter, we further extend the method presented previously. We study the integration of simulation parameter inference with both model-free reinforcement learning and model-based control in a novel sequential algorithm that alternates between learning a better estimation of parameters and improving the controller. This approach exploits the interdependence between the two problems to generate computational efficiencies and improved reliability when a black-box simulator is available. Experimental results suggest that both control strategies have better performance when compared to traditional domain randomization methods.

In this Chapter, we build upon the idea of using probabilistic inference to learn distributions over simulation parameters. Previous method had two main caveats: 1) It requires a reasonable initial controller to be able to explore the state space in regions where it can collect meaningful data for distribution approximation and 2) It only works with much simpler data types, as it relies on manual specification of sufficient statistics. We, therefore, present an end-to-end approach for solving the Sim2Real problem where we tie together controller performance with simulation parameter inference. In this new method, there is no requirement of an initial controller to be provided, as it trains both simulation parameters distributions and controller at the same time. We also propose a sufficient-statistics-free method (SSLFI), where Recurrent Neural Networks (RNN) are used to automatically learn a latent representation of roll-outs. This is a more scalable solution as it enables traditional Likelihood-Free Inference (LFI) in other data domains (e.g. images, videos and etc).

5.2 Related Work

State-of-the-art simulators have enabled the development of new robotics methods. These simulators, however, often make many assumptions requiring approximations to solve complex dynamics represented by differential equations. The increased usage of parallel computation has, fortunately, allowed these approximations to happen much more efficiently. A single simulation can be certainly inaccurate, but, multiple simulations, when combined, can capture reality in all its details. As seen in [7] and [110], its generally understood that an ensemble of simulations can provide better approximation to the underlying complexity in a dynamical system.

Lately, Reinforcement Learning (RL) has been successfully used to train controllers in robotics. However, learned policies are quite sensitive to environmental model miss-specification. To overcome this problem, in [110], policies have been trained on distributions of simulators whose parameters are fit to real-world data. The algorithm switches back and forth updating the domain randomization distribution by minimizing the difference between simulated and real world trajectories. Generalizations of these methods have also been attempted by varying simulation parameters in control tasks as shown in [139]. This helps to understand how simulation accuracy impacts control policies. Another possible approach, when the simulator is differentiable, is to use its gradients to make inference of physical properties. This has been used to learn such parameters on a rigid-body system as discussed in [140].

Alternatively, Model Predictive Control (MPC) can also be used as a robust technique in control problems. In [141] a method was developed to iteratively find a solution of an optimisation problem for a receding finite time-horizon using an approximate model of the system . Sampling is more efficient if the internal model of the system reflects well the real environment where the robot is being deployed. Therefore, the use of distributions over simulator parameters can have major applications in several areas of robotics control.

The use of Bayesian inference for estimating parameters of dynamical systems can be borrowed from more traditional statistics methods such as the seminal approximate Bayesian computation (ABC) found in [57]. Improvements over this method such as Rejection ABC has been proposed by [124]. In [58] and [125], Markov Chain Monte Carlo ABC (MCMC-ABC) and Sequential Monte Carlo ABC (SMC-ABC) provides further improvements by approximating sharper distributions more efficiently. More recent work has although been seen in [126], where an ϵ-free approach have enabled Bayesian inference

on a wider range of problems by introducing Neural Networks for approximating a mixture of gaussians. These methods are usually referred to as Likelihood-Free Inference (LFI) in traditional statistics literature. Their main advantage is that they can handle any problem where the simulation is a black-box generative model.

Although several improvements have been done to LFI methods over time, there still one main caveat that hinders its further progress to a wider range of problems: sufficient statistics needs to be manually chosen for each problem. This limitation not only makes it harder to get such methods to work, but also limits the type of data it can work with, as its extremely hard to find meaningful low dimensional statistical representation for highly dimensional data. In several scenarios, the data is either high dimensional (e.g. images, videos) or highly correlated (e.g. time series) which makes the definition of sufficient statistics challenging. Even more recent state-of-the-art techniques like [61] have still not addressed the issue of dealing with high dimensional input data. Attempts to use sufficient statistics in these cases will not provide a reasonable representation of simulation data which can drastically impact the overall performance. This, unfortunately, limits the usage of such methods to simpler time-series inputs. The method discussed in the next sections tries to fix such problem by automating the sufficient statistics calculation using Neural Networks embeddings.

5.3 Online BayesSim

As described in previous chapter the fundamental goal of LFI methods is to approximate the intractable posterior $p(\boldsymbol{\theta}|\mathbf{x} = \mathbf{x}^r)$ by directly learning a conditional density $q_\phi(\boldsymbol{\theta}|\mathbf{x})$ parameterized by parameters ϕ. BayesSim approximates the posterior distribution using a a mixture of density network as described in [13] where its outputs are the parameters of a Mixture of Gaussians. The ultimate goal is to maximize the likelihood $\prod_n q_\phi(\boldsymbol{\theta}_n|\mathbf{x}_n)$ w.r.t. ϕ.

When learning a controller for a robotics system, for instance, the parameters of the policy are learned separately from the simulation model. In that sense, for every iteration of BayesSim we need to fit a new controller that will be optimized according to that new posterior distribution. This process is rather error prone and can be quite long depending on the target task.

We leverage previous work in BayesSim to simultaneously improve a controller and learn a distribution over the simulator parameters.Additionally, we propose a methodology to automate the computation of a low-dimensional representation of state-action trajectories using Recurrent Neural Networks (RNN).

The approach aims to provide a framework that learns in an end-to-end fashion rather than isolating each otpimization step. In order to achieve this, we propose a methodology to automate the computation of a low-dimensional representation of state-action trajectories using Recurrent Neural Networks (RNN).The difficulty in representing high-dimensional time series has been one of the major reasons why LFI methods do not scale well to higher dimensional spaces. We show that with an RNN, latent representations from entire trajectories can be learnt and used directly for the posterior estimation. This removes the need to manually define meaningful summary statistics, which sometimes, can be a quite difficult and complex task.

5.3.1 Automatic Trajectory Embedding

Trajectories of state-action pairs, in control, are long sequences of correlated data that makes difficult the use of traditional statistical such as the one discussed in the previous sections. The input dimensionality can scale to sizes that makes it prohibitively hard and computationally costly to implement. Although sufficient statistics are one way of reducing the dimensions of the input data, it still does not capture all the needed information for achieving reasonable approximation of the desired posterior distribution.

By introducing temporal models such as RNNs for automated sufficient statistics learning we achieve better scalability and automation on feature extraction from trajectory data. Let $\mathbf{z}$ be a latent representation defined as $\mathbf{z} = \psi_\gamma(\mathbf{S}, \mathbf{A})$ where $\mathbf{S} = \{\mathbf{s}^t\}_{t=1}^T$ and $\mathbf{A} = \{\mathbf{a}^t\}_{t=1}^T$ are sequences of states and actions from $t = 1$ to T and γ are the weights of a LSTM denoted by $\psi_\gamma(\cdot)$. We input the latent embedding $\mathbf{z}$ of state and action pairs directly into the posterior estimator, $q_\phi(\boldsymbol{\theta}|\mathbf{z})$ and train both set of parameters γ and $\boldsymbol{\phi}$ jointly. Gradients are propagated throughout the entire network in a single back-propagation operation. From now on, whenever we refer to updates to the model $q_\phi(\boldsymbol{\theta}|\mathbf{z})$, we will be referring to updates in the parameters of both LSTM feature extractor γ and posterior estimator ϕ.

The use of neural networks for learning feature embeddings has allowed the usage of much richer datasets. In our work we show examples with long sequences of time-series data but this field could be explored further. For instance, sequence of images, sounds and other complex data types could still be used with little modifications in the overall framework we propose.

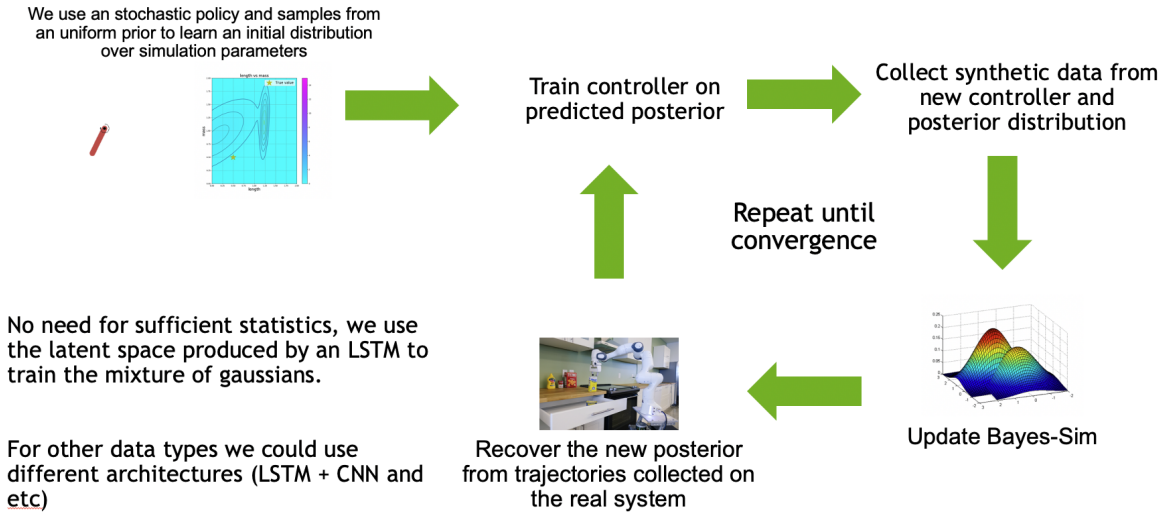


FIGURE 5.1: Sequential policy optimisation and simulator parameter inference.

5.3.2 Sequential Learning

The usual domain randomization approach requires policy optimization and dynamical model learning to happen in different steps. For the likelihood-free inference scenario, where a posterior is approximated through the observations collected in a simulated environment, we shall see that the controllers used to explore the state-action space are not required to be optimal. This allows us to learn both control strategies and environment models in a more iterative and principled way. Alternatively, we also show that the sequential nature of our algorithm can also be used online in MPC methods like [138] to update its internal model to reflect variations in the environment, e.g. adjust friction coefficients, compensate for extra weights added to the robot and etc.

Online BayesSim is an iterative process where at each step t we use the data generated on simulation configurations conditioned on the distribution over parameters $\boldsymbol{\theta}$ learned in the previous step $t-1$. The approach can be seen as a domain randomisation method where the distribution is updated during training by receiving feedback of the current interactions with the environment.

Formally, we start with a stochastic controller $\pi_\beta(\boldsymbol{a_t}|\boldsymbol{s_t})$ and no prior knowledge of the true parameters represented by an uniform prior $p(\boldsymbol{\theta})$. In the first iteration $\pi_\beta(\boldsymbol{a_t}|\boldsymbol{s_t})$ is initialised with samples from the uniform prior $p(\boldsymbol{\theta})$. Trajectories $\mathbf{S}^s, \mathbf{A}^s$ are collected using current $\pi_\beta(\boldsymbol{a_t}|\boldsymbol{s_t})$ which are then used to update our Mixture of Gaussians model $q_\phi(\boldsymbol{\theta}|\mathbf{z})$. New data $\mathbf{S}^r, \mathbf{A}^r$ is then collected in the target

system (e.g. real environment, proxy simulator and etc) using the same controller which is subsequently used to recover a new posterior and update the control strategy. $p(\boldsymbol{\theta}|\mathbf{S}, \mathbf{A} = \mathbf{S}^r, \mathbf{A}^r)$. The prior $p(\boldsymbol{\theta})$ is then replaced by the new posterior and the algorithm iterates until we achieve the desired controller performance. As we shall see in the experiments, there are two possible interpretations for Algorithm 1; When used on a RL setting, improving the controller means performing a gradient step in the direction that maximizes a given reward function. Alternatively in a MPC setup, the controller is improved by replacing its internal environment model to one that better represents the multi-modality of a physical world. A better representation of the environment for such methods means more efficient trajectory sampling that ultimately results in minimizing the involved cost function. In both scenarios, line 13 in the algorithm can be seen as improving our current control strategy to further collect data for our next step of posterior estimation. More details of the entire workflow can be seen in Algorithm 1.

Algorithm 1 Online BayesSim

//observed and real trajectories: $\mathbf{S}^s, \mathbf{A}^s$ *and* $\mathbf{S}^r, \mathbf{A}^r$
//RNN Mixture of Gaussians (MoG) estimator: $q_\phi(\boldsymbol{\theta}|\mathbf{z})$
//RL Policy: $\pi_\beta(\boldsymbol{a_t}|\boldsymbol{s_t})$
Inputs: *total_steps*, *policy_train_steps*, *mog_train_steps*, *num_sampled_params*, $p(\boldsymbol{\theta}_0)$
Outputs: $q_\phi(\boldsymbol{\theta}|\mathbf{z}), \pi_\beta(\boldsymbol{a_t}|\boldsymbol{s_t})$
Initialize weights $\boldsymbol{\beta}$ and $\boldsymbol{\phi}$ randomly
$t \leftarrow 1$
repeat
  $\boldsymbol{\theta}_t \sim p(\boldsymbol{\theta}_{t-1})$
  $\mathbf{S}^s, \mathbf{A}^s \leftarrow$ Run $\pi_{\beta_t}(\boldsymbol{a_t}|\boldsymbol{s_t})$ in sim with $\boldsymbol{\theta}_t$.
  $\boldsymbol{\beta}_t \leftarrow \boldsymbol{\beta}_{t-1} + \lambda \nabla \pi_{\beta_t}(\boldsymbol{a_t}|\boldsymbol{s_t})$
  $\phi_t \leftarrow \phi_{t-1} + \lambda \nabla q_\phi(\boldsymbol{\theta}_t|\psi_{\gamma_t}(\mathbf{S}^s, \mathbf{A}^s))$
  $\mathbf{S}^r, \mathbf{A}^r \leftarrow$ Run $\pi_{\beta_t}(\boldsymbol{a_t}|\boldsymbol{s_t})$ on real env.
  $p(\boldsymbol{\theta}_t|\mathbf{S}, \mathbf{A}) \leftarrow q_\phi(\boldsymbol{\theta}_t|\psi_{\gamma_t}(\mathbf{S}^r, \mathbf{A}^r))$
  $p(\boldsymbol{\theta}_t) \leftarrow p(\boldsymbol{\theta}_t|\mathbf{S}, \mathbf{A})$
  $t \leftarrow t + 1$
until $t < total\_steps$ or convergence reached

5.4 Experiments

In this section we provide the experimental results of Online BayesSim. Perfomance is evaluated in two scenarions: Sim2Sim, where the second simulator uses a different physics engine and on the actual Sim2Real where a Skid-Steer Robot is used. We show that accurate posterior estimation is not entirely required to achieve state of the art control. As we shall see, in some cases an optimal controller can converge without a fully trained dynamics model. This highlights the advantage of using full distributions rather than single point estimates.

Experiments were done in a wide range of control problems. For more simple tasks we have chosen Pendulum, Cartpole, Acrobot and Hopper. In a more intermediate scenario, we have evaluated against sparse rewards setups using both push and slide tasks in the OpenAI Gym framework. Lastly, for completness, we validate against a real world robot using IT-MPC techniques where posteriors are update in simulation with data collected from its real world counterpart and then used to replace the internal model of the controller in real time.

Benchmarking is done with both ϵ-Free [126] and RFF (Random Fourier Features) [7] methods. On the real robot setup, we show that our work can assist IT-MPC methods like DISCO [138] to react to changes in the environment by performing iterative and real time posterior updates. For all experiments we have collected data from multiple attempts in order to better capture the variance of different initializations.

5.4.1 Classic control tasks

Pendulum, Carpole and Acrobot are evaluated using different instances of the same simulation, in other words, using the same physics engine. Hopper, on the other hand, is tested using a different simulator/-physics engine. The "real world" is then represented by MuJoCo while the controller is trained entirely on PyBullet. All RL Policies used are from stable-baselines package with default parameters. The mixture of gaussians for posterior estimation uses 5 components to allow for the recovery of multimodal distributions.

On each inner-loop step of Online BayesSim we trained a PPO2 (Cartpole, Acrobot) [55] and SAC (Pendulum) [142] policy $\pi_{\boldsymbol{\beta}}(a_t|s_t)$ for 300 steps where each episode has 50 steps and $q_\phi(\boldsymbol{\theta}|\mathbf{z})$ for 200 epochs with 200 trajectories generated by the current policy. In both tasks, there is no initial training from the uniform prior, the policy used in the first iteration has random weights. We run the outer-loop for a total of 30 epochs and we achieve state of the art log-likelihood on posterior estimation for all tasks while also achieving maximum reward with the learned controller. Results on figure 5.2 shows that the same multi-modality recovered in previous work [7] is very similar to the one achieved here. However, the distribution is more peaked due to the higher log-likelihood as depicted on table 5.1.

In the Hopper problem, we train a PPO2 policy with the same configuration as above but the only difference is that we run the outer-loop for longer (50 epochs). Although highest log-likelihood value is not achieved in this problem, values are close enough to consider successful transfer between two different simulation engines. It is worth to note that previous work has evaluated the same problem

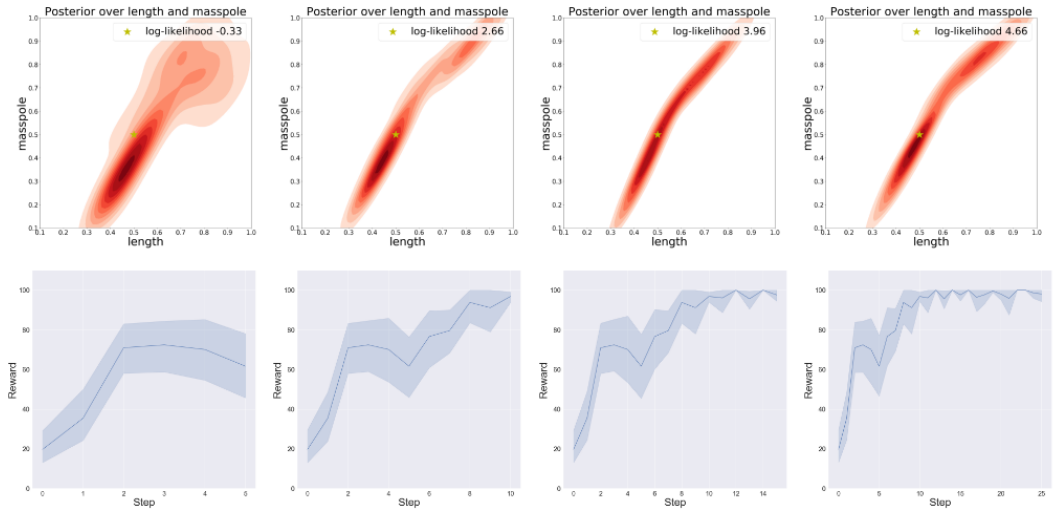


FIGURE 5.2: Accumulated rewards and recovered posterior at steps 5, 10, 15 and 25 shows that a policy for PPO2 can be trained while we approximate the distribution over true parameters (length and masspole)

Problem	Parameter	Prior	Online BayesSim	BayesSim RFF	ϵ-Free
CartPole	Length / Mass	$[0.1, 1.0]$	**4.66±0.22**	2.68±0.08	2.88±0.15
Pendulum	Length / Mass	$[0.1, 1.0]$	**4.076±0.12**	3.89±0.34	3.332±0.41
Fetch Push	Friction	$[0.1, 2.0]$	2.09±0.12	**2.18±0.19**	2.10±0.27
Fetch Slide	Friction	$[0.1, 2.0]$	**3.24±0.21**	3.12±0.08	2.55±0.26
Hopper	Lat. Friction	$[0.1, 0.5]$	3.25±0.33	3.134±0.11	**3.384±0.25**
Acrobot	Link Mass 1 & 2	$[0.5, 2.0]$	**2.85±0.12**	1.534±0.22	1.210±0.32
	Link Length 1 & 2	$[0.1, 1.5]$	**2.25±0.13**	1.426±0.11	1.012±0.21

TABLE 5.1: Mean and standard deviation of log-likelihood of the joint distribution for offline and online likelihood-free methods, applied to different problems and combination of parameters

using the same engine on both ends. Our evaluation is more challenging as each simulation engine have specific ways of implementing their differential equations.

Lastly, the highest difference in posterior evaluation can be seen on the Acrobot task. As we increase the dimensionality of the parameters being estimated, methods with sufficient statistics will not capture the correct latent space of state-action pairs. This shows that using RNNs for automatic trajectory embedding works better on problems where we estimate a higher number of simulation parameters.

5.4.2 Sparse rewards robotics tasks

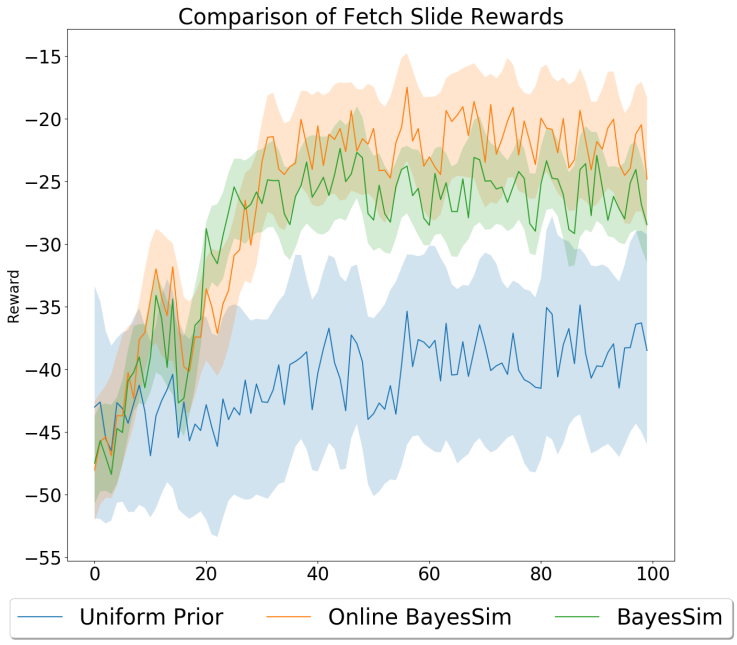
Both experiments with the Fetch robot required a different setup from the ones above. While in the classical control tasks we start with a completely random controller, in this set of experiments we need to train the initial controller on a uniform prior for 10 epochs before we start the main method loop. This happens as we have to firstly learn basic kinematics to then be able to cause the perturbation on the state space that is required to learn the dynamics model.

In order to solve the sparse reward problem of both tasks we have used DDPG coupled with Hindsight Experience Replay. In the first iteration we train the policy on uniform prior for 15 epochs and for the remaining of the loop we train the controller for 5 epochs before fitting a new posterior. In order to work well, we were also required to reduce the size of the HER replay buffer. Since the replay buffer stores past experiences and our simulator is changing at every step, we require a buffer that is able to replace past experiences quickly so as to accommodate the constant changes in simulation.

Results depicted on Figure 5.3 shows that higher reward is achieved for both experiments when compared to different domain randomization techniques. While on the push task the uniform domain randomization is able to achieve similar reward at the end of training when compared to other methods, in the fetch slide there is no visible reward increase. The log-likelihood presented on Table 5.1 shows that our method outperforms previous work on the more challenging slide task while reaches similar log-likelihood on the push task with lower variance and higher overall reward. Variance in rewards for both experiments is higher as we use non-converged dynamics distributions along the way, therefore it causes the robot to experiment more uncertainty while optimizing its controller.

5.4.3 Experiments on a physical robot

This section presents experimental results with a physical robot equipped with a skid-steering drive mechanism (Figure 5.4). Kinematics of the robot were modelled based on a modified unicycle model, which accounts for skidding via an additional parameter as it has been presented in [143]. Online BayesSim estimates the following parameters: robot's wheel radius, axial distance, and the displacement of the robot's instant centre of rotation (ICR) from the robot's centre. A non-zero value on the latter affects turning by sliding the robot sideways. The control task was defined as following a circular path at a constant tangential speed.



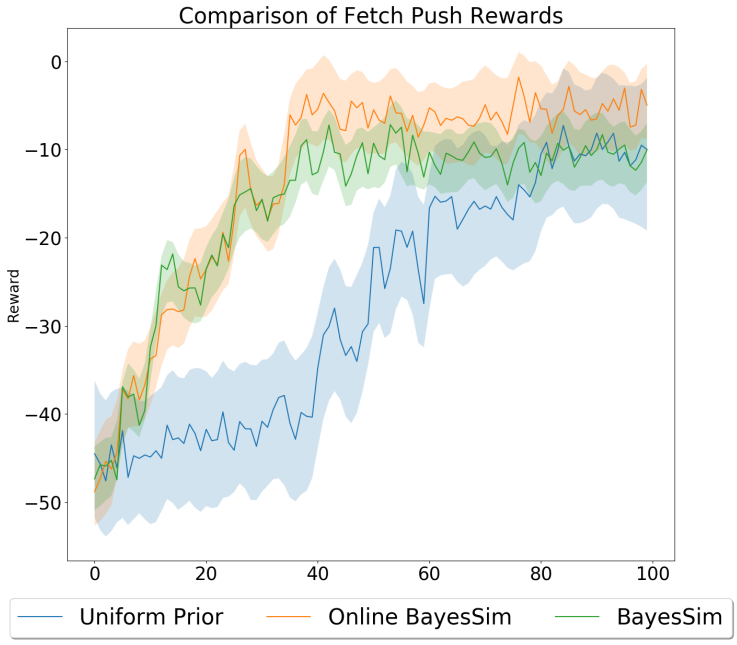


FIGURE 5.3: (Top) Comparison of different DR techniques in a open loop task shows that Online BayesSim outperforms its offline counterpart while uniform DR does not seem to work. (Bottom) Comparison on a Closed Loop task, uniform DR seems to work better but still would require more steps to reach the same reward as the two other methods.

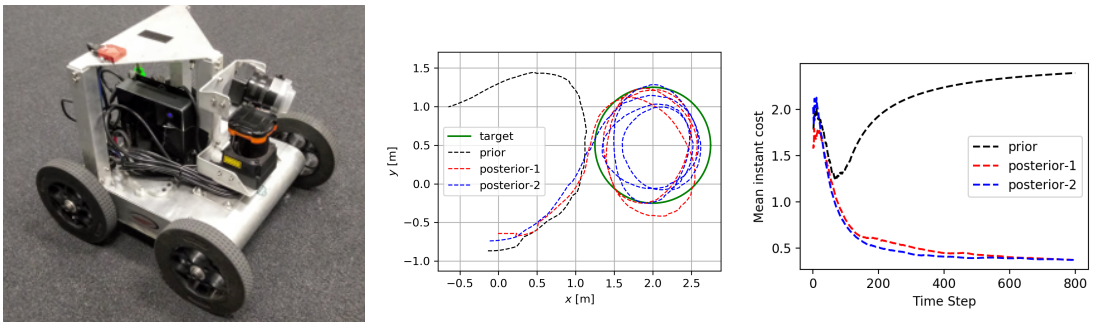


FIGURE 5.4: (Left) Skid-steer Robot. (Center) As posteriors are refined the controller has fewer overshoots on the circular trajectory. (Right) Cumulative average of the cost over time.

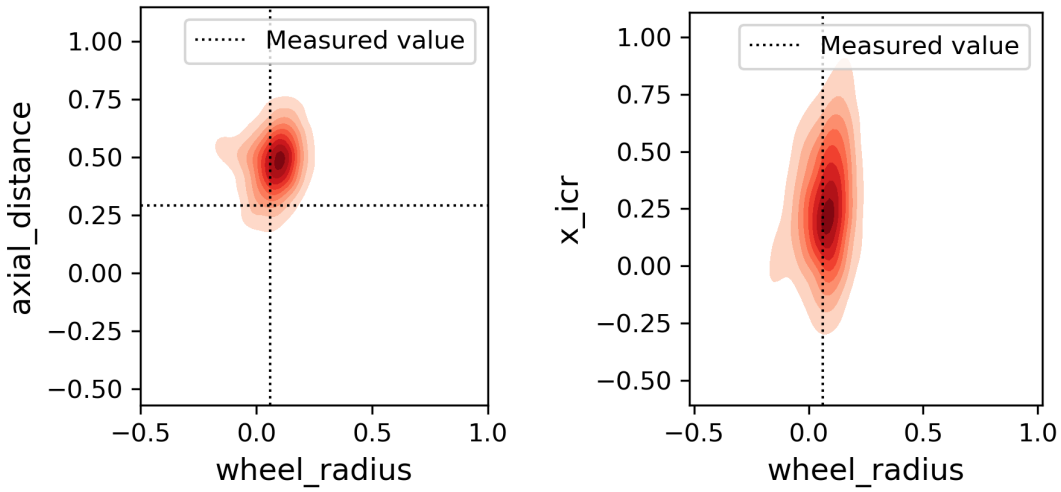


FIGURE 5.5: Posterior distribution for the *first* iteration of online BayesSim on the experiments with a physical robot. Available measured values are indicated by a dashed line.

We have used DISCO from [138] as the robot's controller, a stochastic non-linear MPC based on MPPI. Such method considers the uncertainty in the system's parameters and, hence, can be used together with the posteriors recovered in each step of Online BayesSim. The main advantage of using Online BayesSim instead of its offline counterpart is that we can continuously update the internal model of the controller to reflect changes in the environment in real time. As the robot experiences different scenarios, the data is then incorporated in the main loop where multimodal posteriors can arise naturally to reflect changes such as increase/decrease in robot weight, different friction coefficients, etc.

In order to evaluate our method in the above task, we first initialize the controller with a wide uniform prior and iteratively update it with trajectory data through BayesSim. The resulting posterior is then

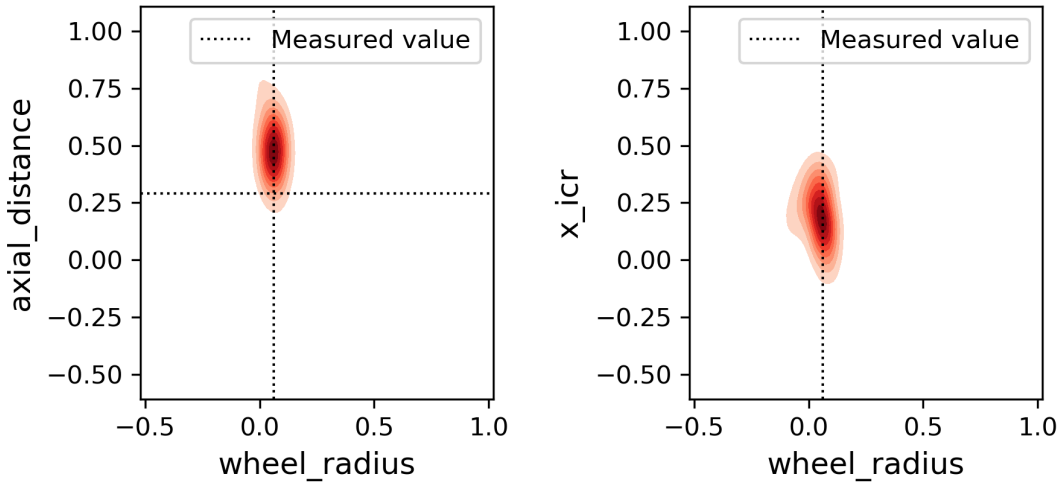


FIGURE 5.6: Posterior distribution for the *second* iteration of online BayesSim on the experiments with a physical robot. Available measured values are indicated by a dashed line.

updated in the controller for the next batch of data collection. This process is repeated until there are no considerable changes in the estimated posterior or reasonable performance is achieved by the controller.

The results presented in Figure 5.4 and Table 5.2 show the qualitative and quantitative improvement in the control task as the posterior distribution is refined. As expected, once the robot is able to collect data from the real environment and refine its knowledge of the world, the results improve significantly. After that, the posterior distribution keeps adjusting to the idiosyncrasies of the environment, as reflected by the further improvement, albeit not as substantially. The main benefit of this setup, however, is the fact that refining the posterior distribution provides a way of adjusting to the non-stationary characteristic of the environment with a fast response to the sudden changes.

Figure 5.5 presents the first posterior, which was estimated from data obtained by running DISCO with the uniform prior. The second posterior estimated by running DISCO with the previous posterior is shown in Figure 5.6. As evident from the plots, the second posterior distribution is more concentrated, providing a more informative guess of the true physical parameters to the controller. Another evidence to highlight is that, although one can measure the wheel radius and the axial distance without many difficulties, the parameter indicating the robot's ICR is hard to measure, involving an elaborate weighing process. The estimate from Online BayesSim then again highlights the usefulness of Bayesian inference for robotic problems with unknown physical parameters.

Sampling distribution for θ	Mean cost
Prior	2.39 ± 0.48
Posterior after first iteration	0.37 ± 0.35
Posterior after second iteration	$\mathbf{0.36 \pm 0.32}$

TABLE 5.2: Average task cost for the robotic experiment with different types of sampling distribution for the model parameters.

5.5 Summary

In this chapter we presented a generalisation over the BayesSim algorithm for sequential posterior estimation. The method aims to solve the "reality gap" problem in robotics simulators and it combines parameter estimation with controller improvement in an end to end approach. Sequential improvements in controller performance are used to estimate better simulation parameters with its associated uncertainty. Online BayesSim outperforms previous methods both in terms of accumulated reward and quality of the recovered posterior. Besides, it can be considered a more general approach as it does not need an optimal controller to start with. Moreover, we have also shown that our method connects traditional statistics literature with more recent ideas on learning latent representations for dynamical systems with recurrent neural networks. The main takeaway of this chapter is to show that sequential optimisation of both policy and simulator parameter inference can reduce the amount of calculations and iterations required to achieve optimal convergence of such functions.

CHAPTER 6

Conclusions

This thesis addressed the problem of energy efficient policy optimisation along with bridging the sim-to-real gap using uncertainty estimation through likelihood-free inference methods. The general framework of LFI has been applied and improved to be used in policy optimisation and simulation parameter inference. Improvements over the current state-of-the-art methods were developed and extended the use of traditional LFI models to more complex and highly dimensional data. This chapter summarises the thesis' contributions and presents possible future research directions.

6.1 Contributions

The following reviews the contributions of this thesis organised per topic.

6.1.1 Energy Efficient Policies

Chapter 3 developed a method for learning a control policy that trades-off energy consumption by classification accuracy. In an egocentric setting, our model learns a reinforcement learning policy that chooses actions between different sensor inputs (motion and vision) in order to reduce the overall energy consumption of the attached battery. The method highlights the importance of taking into the consideration external factors when optimising a policy such as the different discharge rates a battery can have when deployed in different environments (e.g. a battery in higher temperature environment will have higher discharge rate).

6.1.2 Uncertainty estimation in bridging the sim-to-real gap

Chapter 4 presented a novel method to estimate distributions over simulation parameters. Given a simulated environment with several parameters, the method uses a collection of state and action pairs for a

specific task and learns a distribution over the configuration parameters of a simulator. The main advantage over traditional Domain Randomisation methods is that it can not only take prior information about an environment but it also refines the initial distributions with correct measurements of their deviation and multi-modality. This can substantially improve controllers performance as they now will be able to generalise better over the most likely states of the environment their are currently deployed.

6.1.3 End to End policy optimisation and simulation parameter estimation

Chapter 5 addressed the problem of optimising both the policy and the simulation parameter estimation. In an end-to-end framework, the method alternates between performing optimisation steps in both the controller and the simulation model. Besides, it also makes uses of recurrent neural networks to compress temporal and highly dimensional data inputs in order to scale LFI methods to a set of more interesting problems. The method speeds up considerably the time to fully train a policy that can be deployed in a real world with comparable performance to that obtained during simulation.

6.2 Future work

The combination of Bayesian inference for black-box generative models and policy search opens a myriad of possibilities for applications in robotics when a simulator is available and a controller needs to be optimised for further deployment into a real system. The automatic trajectory embedding method developed in this thesis can be further extended to receive sequences of images as inputs, leveraging powerful simulators with realistic graphics capabilities. In this manner, even appearance parameters such as textures and illumination could be estimated from sequences of images. Moreover, multi-streams of different inputs could be used, to mimic how human-beings actually make decisions. For instance, its well known that we depend on different types of data (sound, images, etc) to be able to make coherent decision making. In the same way, a robot could make use of such data to understand how the environment around it looks like. Incorporating multi-streams of data for environment understanding and optimising their use for decision making can be a major future direction of this work as it brings us closer and closer to create truly autonomous systems.

Creating randomisable simulation environments is a time-consuming tasks, and initial guess for the domain parameters are usually inaccurate. This process, therefore, needs to be grounded by real-world data. For instance, one could record an environment with a RGBD camera, and use the information

to reconstruct the scene in simulation. This collected data is then used to infer all required domain parameters. Another problem is that Deep NNs are by far the most common policy type, as they are not only flexible function approximators but they also have a lot of expressiveness. They are, however, brittle w.r.t changes in their input. The inevitable domain shift in sim-to-real scenarios magnifies this sensitivity. Pairing the expressiveness of deep NNs with physically-grounded prior knowledge leads to controllers that achieve high performance and suffer less from transferring to the real world.

Finally, we highlight the importance of treating the estimation of simulation parameters from a probabilistic point of view since inferring posterior distributions as opposed to point estimates can lead to more robust, safe and reliable controllers. As discussed, our world is inherently uncertain and we cannot expect to make point wise estimation of physical behaviors. Dynamical systems are too complex and sometimes differential equations cannot capture all of their singularities. Therefore, by estimating uncertainty around such processes we are able to account for all possible differences that can occur.

Bibliography

[1] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE, 2017, pp. 23–30.

[2] M. A. Nielsen, "Neural networks and deep learning," 2018. [Online]. Available: http://neuralnetworksanddeeplearning.com/

[3] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional neural networks for speech recognition," *IEEE/ACM Transactions on audio, speech, and language processing*, vol. 22, no. 10, pp. 1533–1545, 2014.

[4] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[5] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[6] V. Verma, A. Jónsson, R. Simmons, T. Estlin, and R. Levinson, "Survey of command execution systems for nasa spacecraft and robots," in *Plan Execution: A Reality Check. Workshop at The International Conference on Automated Planning & Scheduling, ICAPS*, 2005, pp. 92–99.

[7] F. Ramos, R. C. Possas, and D. Fox, "Bayessim: adaptive domain randomization via probabilistic inference for robotics simulators," *arXiv preprint arXiv:1906.01728*, 2019.

[8] P. Abbeel, A. Coates, and A. Y. Ng, "Autonomous helicopter aerobatics through apprenticeship learning," *The International Journal of Robotics Research*, vol. 29, no. 13, pp. 1608–1639, 2010.

[9] P. Abbeel, A. Coates, M. Quigley, and A. Y. Ng, "An application of reinforcement learning to aerobatic helicopter flight," in *Advances in neural information processing systems*, 2007, pp. 1–8.

[10] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba, "Hindsight experience replay," in *Advances in Neural Information Processing Systems*, 2017, pp. 5048–5058.

[11] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition,*, 2016. [Online]. Available: http://arxiv.org/abs/1512.00567

[12] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-real transfer of robotic control with dynamics randomization," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 1–8.

[13] C. M. Bishop, "Mixture density networks," Citeseer, Tech. Rep., 1994.

[14] K. P. Murphy, *Machine Learning A Probabilistic Perspective*. Cambridge, MA: The MIT Press, 2012.

[15] L. Bottou, "Stochastic gradient descent tricks," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 421–436.

[16] G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent," *Cited on*, vol. 14, no. 8, p. 2, 2012.

[17] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[18] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[19] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[20] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[22] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[23] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Icml*, 2010.

[24] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.

[25] R. Possas, S. Pinto Caceres, and F. Ramos, "Egocentric activity recognition on a budget," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 5967–5976.

[26] A. Toshev and C. Szegedy, "Deeppose: Human pose estimation via deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 1653–1660.

[27] H. Pirsiavash and D. Ramanan, "Detecting activities of daily living in first-person camera views," in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, 2012.

[28] J. Fu, J. Liu, H. Tian, Y. Li, Y. Bao, Z. Fang, and H. Lu, "Dual attention network for scene segmentation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 3146–3154.

[29] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[30] S. Team, "Deep learning for siri's voice: On-device deep mixture density networks for hybrid unit selection synthesis," *Apple Machine Learning J*, vol. 1, no. 4, 2017.

[31] A. Graves, "Generating sequences with recurrent neural networks," *arXiv preprint arXiv:1308.0850*, 2013.

[32] K. Mülling, J. Kober, O. Kroemer, and J. Peters, "Learning to select and generalize striking movements in robot table tennis," *The International Journal of Robotics Research*, vol. 32, no. 3, pp. 263–279, 2013.

[33] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and autonomous systems*, vol. 57, no. 5, pp. 469–483, 2009.

[34] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[35] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[36] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial intelligence*, vol. 101, no. 1-2, pp. 99–134, 1998.

[37] R. Bellman, "On the theory of dynamic programming," *Proceedings of the National Academy of Sciences*, vol. 38, no. 8, pp. 716–719, 1952.

[38] L. C. Baird III, "Advantage updating," WRIGHT LAB WRIGHT-PATTERSON AFB OH, Tech. Rep., 1993.

[39] M. E. Harmon and L. C. Baird III, "Multi-player residual advantage learning with general function approximation," *Wright Laboratory, WL/AACF, Wright-Patterson Air Force Base, OH*, pp. 45 433–7308, 1996.

[40] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581*, 2015.

[41] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, 2016, pp. 1928–1937.

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

[43] M. P. Deisenroth, G. Neumann, J. Peters *et al.*, "A survey on policy search for robotics," *Foundations and Trends® in Robotics*, vol. 2, no. 1–2, pp. 1–142, 2013.

[44] F. Gomez and J. Schmidhuber, "Evolving modular fast-weight networks for control," in *International Conference on Artificial Neural Networks*. Springer, 2005, pp. 383–389.

[45] J. Koutník, G. Cuccu, J. Schmidhuber, and F. Gomez, "Evolving large-scale neural networks for vision-based reinforcement learning," in *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM, 2013, pp. 1061–1068.

[46] D. Wierstra, A. Förster, J. Peters, and J. Schmidhuber, "Recurrent policy gradients," *Logic Journal of the IGPL*, vol. 18, no. 5, pp. 620–634, 2010.

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

[48] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *Advances in neural information processing systems*, 2000, pp. 1008–1014.

[49] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

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

[51] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[52] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine learning*, vol. 8, no. 3-4, pp. 293–321, 1992.

[53] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[54] T. Schaul, D. Horgan, K. Gregor, and D. Silver, "Universal value function approximators," in *International Conference on Machine Learning*, 2015, pp. 1312–1320.

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

[56] P. Diggle and R. Gratton, "Monte Carlo methods of inference for implicit statistical models," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 2, no. 46, pp. 193–227, 1984.

[57] M. A. Beaumont, W. Zhang, and D. J. Balding, "Approximate bayesian computation in population genetics," *Genetics*, vol. 4, no. 162, pp. 2025–2035, 2002.

[58] P. Marjoram, J. Molitor, V. Plagnol, and S. Tavaré, "Markov chain Monte Carlo without likelihoods," *Proceedings of the National Academy of Sciences*, vol. 100, no. 26, pp. 15 324–15 328, 2003.

[59] S. Sisson, Y. Fan, and M. Tanaka, "Sequential Monte Carlo without likelihoods," *Proceedings of the National Academy of Sciences*, vol. 104, no. 6, pp. 1760–1765, 2007.

[60] J. A. Vrugt and M. Sadegh, "Toward diagnostic model calibration and evaluation: Approximate bayesian computation," *Water Resources Research*, vol. 49, no. 7, pp. 4335–4345, 2013.

[61] G. Papamakarios, D. C. Sterratt, and I. Murray, "Sequential neural likelihood: Fast likelihood-free inference with autoregressive flows," *arXiv preprint arXiv:1805.07226*, 2018.

[62] S. Song, V. Chandrasekhar, B. Mandal, L. Li, J. H. Lim, G. S. Babu, P. P. San, and N. M. Cheung, "Multimodal multi-stream deep learning for egocentric activity recognition," in *IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2016.

[63] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[64] N. Ballas, L. Yao, C. Pal, and A. Courville, "Delving Deeper into Convolutional Networks for Learning Video Representations," pp. 1–11, 2016. [Online]. Available: http://arxiv.org/abs/1511.06432

[65] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 2625–2634.

[66] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-Scale Video Classification with Convolutional Neural Networks," *2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1725–1732, 2014. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6909619{%}5Cnpapers3://publication/doi/10.1109/CVPR.2014.223

[67] J. Y. H. Ng, M. Hausknecht, S. Vijayanarasimhan, O. Vinyals, R. Monga, and G. Toderici, "Beyond short snippets: Deep networks for video classification," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 07-12-June-2015, pp. 4694–4702, 2015. [Online]. Available: https://arxiv.org/pdf/1503.08909.pdf

[68] K. Simonyan and A. Zisserman, "Two-stream convolutional networks for action recognition in videos," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, pp. 568–576. [Online]. Available: http://dl.acm.org/citation.cfm?id=2968826.2968890

[69] S. Ji, W. Xu, M. Yang, and K. Yu, "3d convolutional neural networks for human action recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 1, pp. 221–231, Jan 2013.

[70] P. J. Hsieh, Y. L. Lin, Y. H. Chen, and W. Hsu, "Egocentric activity recognition by leveraging multiple mid-level representations," in *2016 IEEE International Conference on Multimedia and Expo (ICME)*, July 2016, pp. 1–6.

[71] Y. Li, A. Fathi, and J. M. Rehg, "Learning to predict gaze in egocentric video," in *2013 IEEE International Conference on Computer Vision*, Dec 2013, pp. 3216–3223.

[72] M. Ma, H. Fan, and K. M. Kitani, "Going deeper into first-person activity recognition," in *Computer Vision and Pattern Recognition (CVPR)*, 2016.

[73] K. Matsuo, K. Yamada, S. Ueno, and S. Naito, "An attention-based activity recognition for egocentric video," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2014 IEEE Conference on*, June 2014, pp. 565–570.

[74] Y. Li, Z. Ye, and J. M. Rehg, "Delving into egocentric actions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 287–295.

[75] P. Morerio, L. Marcenaro, and C. S. Regazzoni, "Hand detection in first person vision," in *Proceedings of the 16th International Conference on Information Fusion*, July 2013, pp. 1502–1507.

[76] H. Wang, A. Kläser, C. Schmid, and C.-L. Liu, "Dense trajectories and motion boundary descriptors for action recognition," *Int. J. Comput. Vis.*, vol. 103, no. 1, pp. 60–79, mai 2013.

[77] K. Ogaki, K. M. Kitani, Y. Sugano, and Y. Sato, "Coupling eye-motion and ego-motion features for first-person activity recognition," in *2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, June 2012, pp. 1–7.

[78] M. S. Ryoo, B. Rothrock, and L. Matthies, "Pooled motion features for first-person videos," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 896–904.

[79] K. Zhan, S. Faux, and F. Ramos, "Multi-scale conditional random fields for first-person activity recognition," in *Pervasive Computing and Communications (PerCom), 2014 IEEE International Conference on*, March 2014, pp. 51–59.

[80] L. Wang, Y. Xiong, Z. Wang, Y. Qiao, D. Lin, X. Tang, and L. Van Gool, "Temporal segment networks: Towards good practices for deep action recognition," in *Computer Vision – ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part VIII*, vol. 9912. Cham: Springer, 2016, pp. 20–36. [Online]. Available: https://doi.org/10.1007/978-3-319-46484-8_2

[81] A. Betancourt, P. Morerio, C. Regazzoni, and M. Rauterberg, "The evolution of first person vision methods: A survey," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.

[82] L. Bao and S. S. Intille, "Activity recognition from user-annotated acceleration data," in *Pervasive Computing*, 2004.

[83] N. D. Lane, P. Georgiev, and L. Qendro, "Deepear: Robust smartphone audio sensing in unconstrained acoustic environments using deep learning," in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, ser. UbiComp '15. New York, NY, USA: ACM, 2015, pp. 283–294. [Online]. Available: http://doi.acm.org/10.1145/2750858.2804262

[84] X. Zhang, J. Wang, Q. Gao, X. Ma, and H. Wang, "Device-free wireless localization and activity recognition with deep learning," in *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, March 2016, pp. 1–5.

[85] X. Li, Y. Zhang, I. Marsic, A. Sarcevic, and R. S. Burd, "Deep learning for rfid-based activity recognition," in *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys '16. New York, NY, USA: ACM, 2016, pp. 164–175. [Online]. Available: http://doi.acm.org/10.1145/2994551.2994569

[86] J. Wu, A. Osuntogun, T. Choudhury, M. Philipose, and J. M. Rehg, "A scalable approach to activity recognition based on object use," in *2007 IEEE 11th International Conference on Computer Vision*, Oct. 2007, pp. 1–8.

[87] M. Riedmiller, T. Gabel, R. Hafner, and S. Lange, "Reinforcement learning for robot soccer," *Autonomous Robots*, vol. 27, no. 1, pp. 55–73, 2009.

[88] S. Mathe, A. Pirinen, and C. Sminchisescu, "Reinforcement learning for visual object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2894–2902.

[89] S. Karayev, M. Fritz, and T. Darrell, "Anytime recognition of objects and scenes," in *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*, 2014. [Online]. Available: https://doi.org/10.1109/CVPR.2014.80

[90] S. Song, V. Chandrasekhar, N.-M. Cheung, S. Narayan, L. Li, and J.-H. Lim, "Activity recognition in egocentric life-logging videos," in *Computer Vision - ACCV 2014 Workshops*, ser. Lecture Notes in Computer Science, C. V. Jawahar and S. Shan, Eds. Springer International Publishing, 2015, vol. 9010, pp. 445–458.

[91] S. Song, N. M. Cheung, V. Chandrasekhar, B. Mandal, and J. Liri, "Egocentric activity recognition with multimodal fisher vector," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, vol. 2016-May, 2016, pp. 2717–2721.

[92] R. S. Sutton and A. G. Barto, *Reinforcement Learning : An Introduction*. MIT Press, 1998.

[93] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[94] R. J. Williams and J. Peng, "Function optimization using connectionist reinforcement learning algorithms," *Connection Science*, vol. 3, no. 3, pp. 241–268, 1991.

[95] S. Schaal and C. G. Atkeson, "Robot juggling: implementation of memory-based learning," *IEEE Control Systems Magazine*, vol. 14, no. 1, pp. 57–71, 1994.

[96] A. W. Moore, "Fast, robust adaptive control by learning only forward models," in *Advances in neural information processing systems*. Citeseer, 1992, pp. 571–578.

[97] P. W. Battaglia, R. Pascanu, M. Lai, D. Rezende, and K. Kavukcuoglu, "Interaction networks for learning about objects, relations and physics," *arXiv preprint arXiv:1612.00222*, 2016.

[98] Z. Xu, J. Wu, A. Zeng, J. B. Tenenbaum, and S. Song, "Densephysnet: Learning dense physical object representations via multi-step dynamic interactions," *arXiv preprint arXiv:1906.03853*,

2019.

[99] S. He, Y. Li, Y. Feng, S. Ho, S. Ravanbakhsh, W. Chen, and B. Póczos, "Learning to predict the cosmological structure formation," *Proceedings of the National Academy of Sciences*, vol. 116, no. 28, pp. 13 825–13 832, 2019.

[100] M. Raissi, H. Babaee, and P. Givi, "Deep learning of turbulent scalar mixing," *Physical Review Fluids*, vol. 4, no. 12, p. 124501, 2019.

[101] O. Chapelle and V. Vapnik, "Model selection for support vector machines," *Advances in neural information processing systems*, vol. 12, pp. 230–236, 1999.

[102] Z. Long, Y. Lu, X. Ma, and B. Dong, "Pde-net: Learning pdes from data," in *International Conference on Machine Learning*. PMLR, 2018, pp. 3208–3216.

[103] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," *Journal of Computational Physics*, vol. 378, pp. 686–707, 2019.

[104] M. Lutter, C. Ritter, and J. Peters, "Deep lagrangian networks: Using physics as model prior for deep learning," *arXiv preprint arXiv:1907.04490*, 2019.

[105] C. Gourieroux, A. Monfort, and E. Renault, "Indirect inference," *Journal of applied econometrics*, vol. 8, no. S1, pp. S85–S118, 1993.

[106] C. M. Schafer and P. E. Freeman, "Likelihood-free inference in cosmology: Potential for the estimation of luminosity functions," in *Statistical Challenges in Modern Astronomy V*. Springer, 2012, pp. 3–19.

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

[108] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke, "Sim-to-real: Learning agile locomotion for quadruped robots," *arXiv preprint arXiv:1804.10332*, 2018.

[109] I. Mordatch, K. Lowrey, and E. Todorov, "Ensemble-cio: Full-body dynamic motion planning that transfers to physical humanoids," in *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*. IEEE, 2015, pp. 5307–5314.

[110] Y. Chebotar, A. Handa, V. Makoviychuk, M. Macklin, J. Issac, N. Ratliff, and D. Fox, "Closing the sim-to-real loop: Adapting simulation randomization with real world experience," *arXiv preprint arXiv:1810.05687*, 2018.

[111] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning," 2016.

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

[113] J. Lee, M. X. Grey, S. Ha, T. Kunz, S. Jain, Y. Ye, S. S. Srinivasa, M. Stilman, and C. K. Liu, "Dart: Dynamic animation and robotics toolkit," *Journal of Open Source Software*, vol. 3, no. 22, p. 500, 2018.

[114] M. Giftthaler, M. Neunert, M. Stäuble, M. Frigerio, C. Semini, and J. Buchli, "Automatic differentiation of rigid body dynamics for optimal control and estimation," *Advanced Robotics*, vol. 31, no. 22, pp. 1225–1237, 2017.

[115] J. Carpentier and N. Mansard, "Analytical derivatives of rigid body dynamics algorithms," in *Robotics: Science and systems (RSS 2018)*, 2018.

[116] F. de Avila Belbute-Peres, K. Smith, K. Allen, J. Tenenbaum, and J. Z. Kolter, "End-to-end differentiable physics for learning and control," *Advances in neural information processing systems*, vol. 31, pp. 7178–7189, 2018.

[117] T. Koolen and R. Deits, "Julia for robotics: Simulation and real-time control in a high-level programming language," in *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 2019, pp. 604–611.

[118] E. Heiden, D. Millard, H. Zhang, and G. S. Sukhatme, "Interactive differentiable simulation," *arXiv preprint arXiv:1905.10706*, 2019.

[119] Y. Hu, J. Liu, A. Spielberg, J. B. Tenenbaum, W. T. Freeman, J. Wu, D. Rus, and W. Matusik, "Chainqueen: A real-time differentiable physical simulator for soft robotics," in *2019 International conference on robotics and automation (ICRA)*. IEEE, 2019, pp. 6265–6271.

[120] J. Liang, M. Lin, and V. Koltun, "Differentiable cloth simulation for inverse problems," 2019.

[121] Y. Hu, L. Anderson, T.-M. Li, Q. Sun, N. Carr, J. Ragan-Kelley, and F. Durand, "Difftaichi: Differentiable programming for physical simulation," *arXiv preprint arXiv:1910.00935*, 2019.

[122] Y.-L. Qiao, J. Liang, V. Koltun, and M. C. Lin, "Scalable differentiable physics for learning and control," *arXiv preprint arXiv:2007.02168*, 2020.

[123] J. K. Murthy, M. Macklin, F. Golemo, V. Voleti, L. Petrini, M. Weiss, B. Considine, J. Parent-Lévesque, K. Xie, K. Erleben *et al.*, "gradsim: Differentiable simulation for system identification and visuomotor control," in *International Conference on Learning Representations*, 2020.

[124] J. K. Pritchard, M. T. Seielstad, A. Perez-Lezaun, and M. W. Feldman, "Population growth of human y chromosomes: a study of y chromosome microsatellites." *Molecular biology and evolution*, vol. 16, no. 12, pp. 1791–1798, 1999.

[125] F. V. Bonassi, M. West *et al.*, "Sequential Monte Carlo with adaptive weights for approximate bayesian computation," *Bayesian Analysis*, vol. 10, no. 1, pp. 171–187, 2015.

[126] G. Papamakarios and I. Murray, "Fast ε-free inference of simulation models with bayesian conditional density estimation," in *Advances in Neural Information Processing Systems*, 2016, pp. 1028–1036.

[127] M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray *et al.*, "Learning dexterous in-hand manipulation," *arXiv preprint arXiv:1808.00177*, 2018.

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

[129] A. Rahimi and B. Recht, "Random features for large-scale kernel machines," in *Advances in Neural Information Processing Systems 20*, J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, Eds. Curran Associates, Inc., 2008, pp. 1177–1184.

[130] M. L. Stein, *Interpolation of Spatial Data*. Springer-Verlag, New york, 1999.

[131] I. I. Gihman and A. V. Skorohod, *The Theory of Stochastic Processes*. Springer Verlag, Berlin, 1974, vol. 1.

[132] E. Braaten and G. Weller, "An improved low-discrepancy sequence for multidimensional quasi-Monte Carlo integration," *Journal of Computational Physics*, vol. 33, pp. 249–258, Nov. 1979.

[133] H. Avron, V. Sindhwani, J. Yang, and M. W. Mahoney, "Quasi-Monte Carlo feature maps for shift-invariant kernels," *Journal of Machine Learning Research*, vol. 17, no. 120, pp. 1–38, 2016.

[134] N. Srivastava, E. Mansimov, and R. Salakhudinov, "Unsupervised learning of video representations using lstms," in *Proceedings of the 32nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, F. Bach and D. Blei, Eds., vol. 37. Lille, France: PMLR, 07–09 Jul 2015, pp. 843–852.

[135] D. J. Wilkinson, *Stochastic Modelling for Systems Biology, Second Edition*, ser. Chapman & Hall/CRC Mathematical and Computational Biology. Taylor & Francis, 2011.

[136] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[137] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[138] L. Barcelos, R. Oliveira, R. Possas, L. Ott, and F. Ramos, "DISCO: Double Likelihood-free Inference Stochastic Control." [Online]. Available: http://arxiv.org/abs/2002.07379

[139] C. Packer, K. Gao, J. Kos, P. Krähenbühl, V. Koltun, and D. Song, "Assessing generalization in deep reinforcement learning," *arXiv preprint arXiv:1810.12282*, 2018.

[140] E. Heiden, D. Millard, H. Zhang, and G. S. Sukhatme, "Interactive differentiable simulation," 2019.

[141] G. Williams, P. Drews, B. Goldfain, J. M. Rehg, and E. A. Theodorou, "Information-Theoretic Model Predictive Control: Theory and Applications to Autonomous Driving," vol. 34, no. 6, pp. 1603–1622.

[142] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *arXiv preprint arXiv:1801.01290*, 2018.

[143] K. Kozłowski and D. Pazderski, "Modeling and Control of a 4-wheel Skid-steering Mobile Robot," *Int. J. Appl. Math. Comput. Sci.*, vol. 14, no. 4, pp. 477–496, 2004.