ABSTRACT

BATES, DYLAN. Virtual Reinforcement Learning for Balancing an Inverted Pendulum in Real Time. (Under the direction of Hien Tran).

Using a variety of policy-based reinforcement learning techniques, we train a single-hidden-layer artificial neural network to balance a physically accurate simulation of a single inverted pendulum. The trained weights and biases of the neural network are then transferred to a real agent where they can be used to control and balance a real inverted pendulum, in real time. The virtually trained model is robust to sensor noise, model errors, and motor play, and is able to recover from physical disturbances to the system.

We also train a radial basis function network using PILCO, a model-based technique that uses Gaussian processes to propagate uncertainties through a probabilistic dynamics model to reduce model bias for long-term predictions. This technique is significantly more data-efficient, requiring an order of magnitude fewer samples than previous state-of-the art algorithms, and is better able to recover from large disturbances to the system. Its primary limitation is being significantly more computationally expensive and memory intensive.

This hybrid approach of training a simulation allows thousands of trial runs to be completed orders of magnitude faster than would be possible in the real world, resulting in greatly reduced training time and more iterations, producing a more robust model. Using a physically accurate dynamics model allows the system to overcome the sim-real gap without requiring domain randomization or a variational autoencoder to create a domain-invariant latent space. When compared with existing reinforcement learning methods, the resulting control is smoother, learned faster, and able to withstand forced disturbances.

Virtual Reinforcement Learning for Balancing an Inverted Pendulum in Real Time

by
Dylan Bates

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Applied Mathematics

Raleigh, North Carolina

2021

APPROVED BY:

Kevin Flores	Kazufumi Ito
Ryan Murray	Hien Tran Chair of Advisory Committee

DEDICATION

To my cat.

BIOGRAPHY

Dylan Bates was born in Toronto, Canada. He moved to South Carolina for college, and stayed there for his Master's degree, before moving to North Carolina to obtain his PhD, just to confuse everybody back home who kept mixing up the two states. This doesn't really tell you anything about me. Shoot me an email if you'd like to know more.

ACKNOWLEDGEMENTS

I would like to thank COVID-19 for keeping me inside doing research and writing, and J. R. R. Tolkien for providing an inspirational fantasy world to escape to while theatres were closed. My biggest thanks go to my advisor, Dr. Hien Tran, whose weekly meetings kept me on track and who held me accountable to the overambitious goals I often set. There were times when my progress was fueled entirely by your contagious optimism and positive feedback. Finally, I would like to thank the Center for Research in Scientific Computation and the NCSU Mathematics Department for their financial support these last few years.

Oh, and my cat. Thanks, Sara.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Machine learning is a the decades-old study of algorithms that let a computer learn from experience, instead of from manually coding the individual steps. Despite the major breakthroughs that took place in the 1960s, like neural networks [27], perceptrons [54], and backpropagation [11, 30], there has been a recent explosion of results in the field stemming from the rapid increase in computer processing power (specifically graphics processing units and tensor processing units) in the last 10 years. This second renaissance of machine learning has allowed computers to do incredible things like diagnose cancer patients better than doctors [39], generate works in the style of Mozart [49], Shakespeare [9, 10], or Van Gogh [23], and play games like Go [61] or Dota 2 [44] better than professionals. The last two are examples of *reinforcement learning*, and use some of the same algorithms discussed in Chapter 2.

Machine learning is traditionally split into three broad categories:

- Supervised Learning: input data with desired labeled outputs are fed to the computer with the goal of correctly predicting the labels,
- Unsupervised Learning: unlabeled data is presented to the computer with the goal of discovering the underlying structure or generating a model,
- Reinforcement Learning (RL): a computer interacts with an environment by taking actions and receiving rewards, with the goal of maximizing the rewards in order to achieve a predefined goal.

Chapter 1 of this dissertation will focus on the basics of RL, and the history of how it has been applied to our main problem: balancing an inverted pendulum. Chapter 2 will cover the implementation of several policy-based model-free RL algorithms in detail. Chapter 3 will discuss PILCO, a probabilistic model-based algorithm that differs significantly from

the previously mentioned algorithms. Finally, Chapter 4 will cover the results of each implementation, both in a simulation and in the real world.

1.1 Machine Learning

Machine learning is a subset of artificial intelligence (AI) describing programs that are able to learn from experience, making predictions or decisions without being programmed to do so. Paraphrasing Arthur Samuel [55] from 1959:

> How can computers learn to solve problems without being explicitly programmed? In other words, how can computers be made to do what needs to be done, without being told exactly how to do it? [34]

In 1959, Samuels was able to write a program that played Checkers better than he could after 8–10 hours of machine-playing time. In the 60+ years since then, this field has only grown, with state-of-the-art-versions of fundamentally similar techniques (tree searches and neural networks) being implemented on hilariously superior hardware, allowing programs to beat the world champions of Checkers with Chinook in 1994 (64 spaces and 5×10^{20} possible moves) [56], Chess with Deep Blue in 1997 (64 spaces and $< 10^{46}$ possible positions) [12], and Go with AlphaGo in 2016 (361 spaces and $> 2 \times 10^{170}$ possible positions) [61]. Importantly, that latter number is larger than the number of elementary particles in the observable universe (usually estimated around 10^{80}) meaning that the game will *never* be completely solved. AlphaGo and its successors (AlphaGo Zero [62], AlphaZero [63], and MuZero [57]) use reinforcement learning techniques, similar to those in Chapter 2.

1.1.1 Neural Networks

One of the main tools used for this type—and indeed in most types—of machine learning is a neural network. Originally intended to represent the connections between biological neurons in the brain, an artificial neural network consists of layers of neurons connected to every other neuron in the adjacent layers, as in Figure 1.1.

For a typical feedforward neural network, neurons a_j^ℓ in layer ℓ ($1 \leq j \leq N_\ell$) are connected to neurons $a_j^{\ell+1}$ in layer $\ell + 1$ ($(1 \leq j \leq N_{\ell+1})$, along with biases b_j. Neuron a_j^ℓ sums the inputs from $a_k^{\ell-1}$, $1 \leq k \leq N_{\ell-1}$, which are multiplied by weight w_{jk}^ℓ, added to bias b_j^ℓ, and feeds them forward through a continuous, nonlinear activation function $\sigma : \mathbb{R} \to \mathbb{R}$ that sends its output to all the neurons $a_j^{\ell+1}$, $1 \leq j \leq N_{\ell+1}$ in the next layer, as follows:

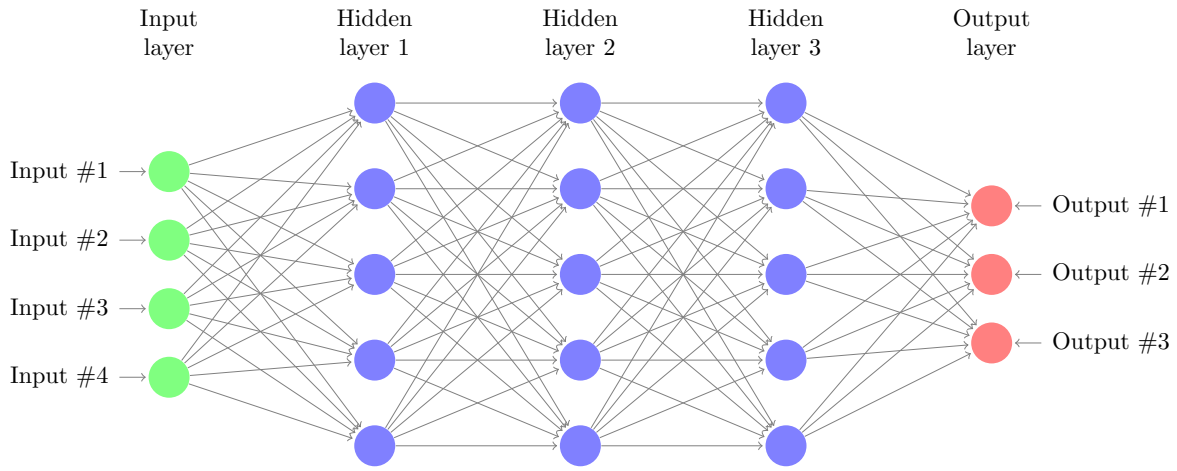


Figure 1.1 Artificial neural network containing 3 hidden layers with 5 neurons each. This network has 103 trainable parameters, as 85 weight parameters (represented by arrows) and 18 bias parameters (represented by non-output nodes).

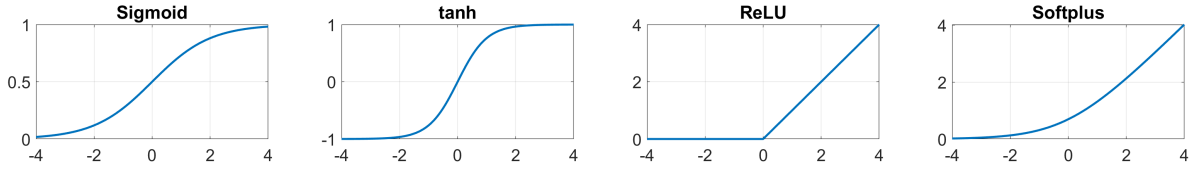


Figure 1.2 Some common activation functions $\sigma(z)$.

$$a_j^\ell = \sigma\left(\sum_{k=1}^{N_\ell} w_{jk}^\ell a_k^{\ell-1} + b_j^\ell\right). \tag{1.1}$$

This feedforward process is repeated for all neurons a_j^ℓ $(1 \leq j \leq N_\ell)$ in all layers $1 \leq \ell \leq L$ until the final values are presented in the output or *top layer* of the network. By defining $w_{j0} = b_j$ in every layer, we can simplify Equation 1.1 to

$$a_j = \sigma\left(\sum_{k=0}^{N} w_{jk} a_k\right), \tag{1.2}$$

for all layers ℓ, which can speed up computation. By overloading $\sigma(x)$ as a function from $\mathbb{R}^n \to \mathbb{R}^n$ such that $\sigma(x)_j = \sigma(x_j)$, we can simplify this even further as

$$a^\ell = \sigma\left(w^\ell \cdot a^{\ell-1}\right). \tag{1.3}$$

Common choices of the activation function σ include the sigmoid, hyperbolic tangent, `ReLU`, and Softplus functions, seen in Figure 1.2.

Intended to mimic biological neurons firing when the electrical potential reaches a certain threshold, the sigmoid function is any continuous function such that $\lim_{z\to-\infty} = 0$ and $\lim_{z\to\infty} = 1$. For example,

$$\sigma(z) = \frac{1}{1+e^{-z}}. \tag{1.4}$$

This has a few benefits over the discrete step function in neurons in the brain:

- $\sigma(z)$ is continuous,
- $\sigma(z)$ is differentiable, with derivative $\sigma'(z) = \sigma(z)\,(1-\sigma(z))$,
- $\sigma(z)$ is *discriminatory*.

Definition 1.1. Let I_n be the n-dimensional unit cube $[0,1]^n$. We say that a function $\sigma(z)$ is *discriminatory* if for a measure $\mu \in M(I_n)$,

$$\int_{I_n} \sigma\left(y^\top x + \theta\right)\, d\mu(x) = 0 \tag{1.5}$$

for all $y \in \mathbb{R}^m$ and $\theta \in \mathbb{R}$ implies that $\mu \equiv 0$ [13].

However, it is only able to output values between 0 and 1. As a result, large negative inputs tend to get squished towards 0, which can inhibit learning. Alternate functions like the hyperbolic tangent function solve this problem by outputting values in the range $[-1,1]$, while remaining differentiable.

$$\tanh(z) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.6}$$

This function maps negative inputs to negative outputs, and positive inputs to positive outputs, which can make learning faster. Additional common activation functions include

$$\texttt{ReLU}(z) = \max(z, 0) \tag{1.7}$$

and

$$\text{Softplus}(z) = \ln(1+e^x), \tag{1.8}$$

both of which are used in this dissertation, as see in the next chapter.

What makes neural networks so powerful is that they are universal function approximators, being able to approximate *any* continuous or Lebesgue integrable function to arbitrary accuracy, given a few conditions.

Theorem 1.2 (Universal Approximation Theorem—Bounded Depth [13])
Let σ be a continuous discriminatory function. Then finite sums of the form

$$F(x) = \sum_{j=1}^{N} \alpha_j \sigma\left(w_j^\top x + b_j\right) \tag{1.9}$$

are dense in (I_n). That is, given any $f \in C(I_n)$ and $\varepsilon > 0$, there is a neural network $F(x)$ for which

$$|F(x) - f(x)| < \varepsilon \text{ for all } x \in I_n.$$

Proof. Let $S \subset C(I_n)$ be the set of functions in the form $F(x)$ as in Equation 1.9. S is a linear subspace of $C(I_n)$; we claim that the closure of S is all of $C(I_n)$.

In order to reach a contradiction, assume that the closure of S is *not* all of $C(I_n)$. Then $\overline{S}$—the closure of S—is a closed proper subspace of $C(I_n)$. By the Hahn-Banach theorem, there is a bounded linear functional L on $C(I_n)$, with the property that $L(\overline{S}) = L(S) = 0$, but $L \neq 0$.

By the Riesz Representation theorem, this bounded linear functional is of the form

$$L(h) = \int_{I_n} h(x)\, d\mu(x) \tag{1.10}$$

for some $\mu \in M(I_n)$, for all $h \in C(I_n)$. Since $\sigma(w^\top x + b) \in \overline{S}$ for all w and b, we must have that

$$\int_{I_n} \sigma(w^\top x + b)\, d\mu(x) = 0 \tag{1.11}$$

for all w and b. However, since σ is discriminatory, this implies $\mu \equiv 0$, contradicting our assumption. Hence, S is dense in $C(I_n)$; therefore continuous functions can be arbitrarily closely approximated by functions in the form of $F(x)$. □

A similar proof can show that for all $\varepsilon > 0$,

$$\|F - f\|_{L^1} = \int_{I_n} |F(x) - f(x)|\ dx < \varepsilon \tag{1.12}$$

and

$$\sup_{x \in K} \|F(x) - f(x)\| < \varepsilon \tag{1.13}$$

for every compact subset $K \subset \mathbb{R}^n$ if and only if σ is continuous and not a polynomial [50]. This can trivially be extended to functions $f \in L^p(\mathbb{R}^n)$ since $C_c(\mathbb{R}^n)$ is dense in $L^p(\mathbb{R}^n)$.

This says that we can approximate continuous functions with arbitrary precision in the L^p $(1 \leq p < \infty)$ norm using neural networks with a single hidden layer, given only mild conditions on the activation function, as long as there are no limits on the number of neurons or bounds on the weights.

Definition 1.3. A function $f(x) : \mathbb{R}^n \to \mathbb{R}$ is *Lebesgue integrable* if

$$\int_{\mathbb{R}^n} |f(x)| \ dx < \infty. \tag{1.14}$$

Theorem 1.4 (Universal Approximation Theorem—Bounded Width [38])
For any Lebesgue integrable function $f(x)$ and any $\varepsilon > 0$, there exists a fully connected network $F(x)$ with width $N \leq n + 4$, such that $F(x)$ satisfies

$$\int_{\mathbb{R}^n} |F(x) - f(x)| \ dx < \varepsilon. \tag{1.15}$$

The proof can be found in the supplementary material of [38], which is beyond the scope of this section, although the constructive outline of a "visual proof" can be found at [43]. We can approximate Lebesgue integrable functions with arbitrary precision in the L^1 norm with `ReLU` neural networks of width $N \leq n + 4$. For L^p integrable functions $f : \mathbb{R}^n \to \mathbb{R}^m$, the bound on the width has been improved to $\max(n + 1, m)$ [48].

Corollary 1.5
There exists a deep neural network with constant width that cannot be approximated by a shallower neural network without at least an exponential increase in width.

However, in order to approximate a single hidden layer neural network with a narrower fixed-width deep neural network, there is only a polynomial lower bound on the number of layers required [38].

Unfortunately, neither Theorems 1.2 and 1.4 nor Corollary 1.5 provide any way of determining precisely how many neurons or layers, respectively are needed for a given bound on ε, and provide no guidance on how to assign weights and biases in such a way to minimize the error between a function and its neural network approximation.

1.1.2 Backpropagation with Gradient Descent

Although the Universal Approximation Theorems are unable to tell us exactly how to determine the weights and biases (collectively referred to as the *paramters* θ) of a neural

network), if we have a *loss function* quantifying the error between our neural network function $F(\theta)$ and desired function, we can roughly minimize this error function by computing the gradients of the loss function using an algorithm called *backpropogation*, and stepping down the error function surface until we find a local (or ideally global) minimum using gradient descent.

Backpropagation is a computationally efficient method of computing the gradients of a loss function of a neural network with respect to the parameter values. Conceptually, this is just repeated application of the chain rule from calculus. Given a set of weights w^ℓ $(1 \leq \ell \leq L)$, activation functions σ^ℓ $(1 \leq \ell \leq L)$, and a loss function $J(\theta)$, the neural network can be defined as

$$F(x) = \sigma^L \left(w^L \sigma^{L-1} \left(w^{L-1} \dots \sigma^1 \left(w^1 x \right) \dots \right) \right). \tag{1.16}$$

For each pair (x_i, y_i) such that $y_i = f(x_i)$, we can define our loss function as whatever metric best fits our needs (e.g. mean squared error, crossentropy loss, etc.). Then

$$J\left(y_i, F(x_i)\right) = J\left(y_i, \sigma^L \left(w^L \sigma^{L-1} \left(w^{L-1} \dots \sigma^1 \left(w^1 x \right) \dots \right) \right) \right), \tag{1.17}$$

so that the total derivative with respect to the inputs is

$$\frac{dJ}{dx} = \frac{dJ}{da^L} \cdot \left(\sigma^L\right)' \cdot w^L \cdot \left(\sigma^{L-1}\right)' \cdot W^{L-1} \dots \left(\sigma^1\right)' \cdot w^1. \tag{1.18}$$

For example, computing the mean squared error between data y_i and network approximations $F(x_i) = \hat{y}_i$ $(1 \leq i \leq M)$, we have

$$J(\theta) = \frac{1}{M} \sum_{i=1}^{M} \left(y_i - \hat{y}_i\right)^2, \tag{1.19}$$

where we often multiply by a factor of $1/2$ to simplify the gradients.

In order to compute the gradient of $J(\theta)$ with respect to a weight w_{jk}^ℓ, we first define an *auxiliary error*

$$\delta^\ell = \left(\sigma^\ell\right)' \cdot \left(w^{\ell+1}\right)^\top \dots \left(\sigma^{L-1}\right)' \cdot \left(w^L\right)^\top \cdot \nabla_{a^L} J, \tag{1.20}$$

from the chain rule (with the transposes coming from the calculation of the gradient), so that we can say

$$\nabla_{w^\ell} J = \delta^\ell \left(a^{\ell-1}\right)^\top. \tag{1.21}$$

In practice, we calculate this recursively from the top layer of the network. For example,

$$\begin{aligned}\nabla_{w^L} J &= \delta^L \left(a^{L-1}\right)^\top \\ &= \left(\sigma^L\right)' \cdot \frac{1}{M}(a^L - y)\left(a^{L-1}\right)^\top \\ &= \sigma^L(z^L)\left(1 - \sigma^L(z_L)\right) \cdot \frac{2}{M}(a^L - y)\left(a^{L-1}\right)^\top,\end{aligned} \tag{1.22}$$

given a sigmoid activation in layer L and mean squared error loss. If we want to calculate the gradients with respect to weights in earlier layers, we proceed recursively, having already calculated important values:

$$\begin{aligned}\nabla_{w^{L-1}} J &= \delta^{L-1}\left(a^{L-2}\right)^\top \\ &= \left(\sigma^{L-1}\right)' \cdot \left(w^L\right)^\top \delta^L \left(a^{L-2}\right)^\top,\end{aligned} \tag{1.23}$$

where we have already calculated and stored δ^L in Equation 1.22. By computing $\delta^{\ell-1}$ in terms of δ^ℓ, backpropagation allows us to compute the gradients in each layer as a few matrix-vector products, avoiding duplicate multiplications in layer ℓ and beyond, and preventing the more computationally expensive matrix-matrix products that would arise from naïvely computing the gradients forward. This process is handled automatically through a technique called automatic differentiation or *autodiff* built into the library being used.

Once we know the gradients, the simplest way to update the weights is by shifting them slightly in opposite direction the direction of the gradients in order to decrease the error. That is,

$$\begin{aligned}w^\ell &:= w^\ell - \eta \nabla_{w^\ell} J \\ &= w^\ell - \eta \delta^\ell \left(a^{\ell-1}\right)^\top \\ &= w^\ell - \frac{\eta}{M} \sum_{i=1}^{M} \delta^{x_i,\ell}\left(a^{x_i,\ell-1}\right)^\top, \\ &= w^\ell - \frac{\eta}{M} \sum_{i=1}^{M} \nabla_{w^\ell}\left(\hat{y}_i - y_i\right)^2,\end{aligned} \tag{1.24}$$

for some positive learning rate η. This is known as *stochastic gradient descent*.

Algorithms that use backpropogation generally fall into one of three categories [36]:

- steepest descent (stochastic gradient descent, Adam)

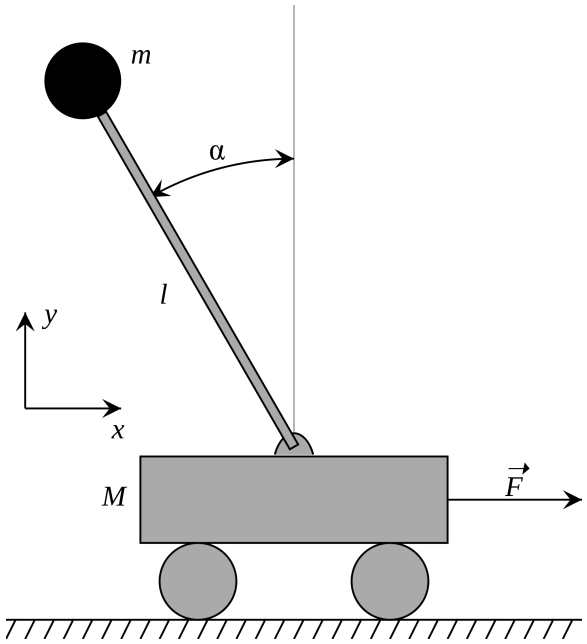


Figure 1.3 Single inverted pendulum: the cart can only move with 1 degree of freedom.

- quasi-Newton (Broyden-Fletcher-Goldfarb-Shanno, limited-memory BFGS)
- conjugate gradient (Fletcher-Reeves, Polak-Ribière)

The algorithms in Chapter 2 use stochastic gradient descent, while PILCO uses limited-memory BFGS.

1.2 Single Inverted Pendulum

The primary goal of this thesis is to use specific machine learning techniques to balance an inverted pendulum. Also known as a *cartpole*, our agent is a single inverted pendulum attached to a cart on a one-dimensional track, with a four-dimensional continuous state space: $[x, \alpha, \dot{x}, \dot{\alpha}]$, corresponding to the x-position (right positive), angle (counterclockwise positive from 0 vertical), x-velocity, and angular velocity, as indicated in Figure 1.3.

The action space is the continuous range of voltages available to the motor, ranging from -10V to $+10$V, moving the cart left or right, respectively. The cart will also move according to the physics of the system, even if no voltage is applied. Our goal is to apply the appropriate voltage to the motor in order to balance the pendulum vertically above the cart, without running the cart off the track. The model is virtually trained, entirely through a simulation in Python. The trained model is then transferred to the real inverted

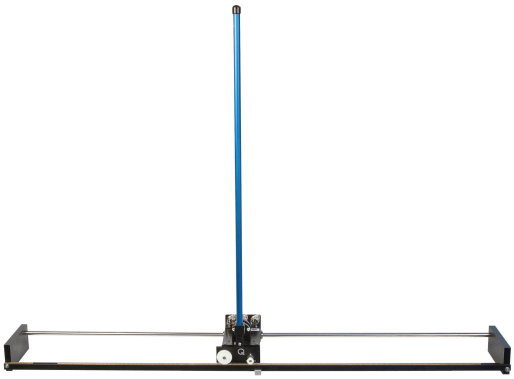


Figure 1.4 Real inverted pendulum with an IP02 servo plant.

pendulum seen in Figure 1.4, where the pole is balanced over the physical cart. The pendulum in the lab was provided by Quanser Consulting Inc.[1]

1.2.1 Environment

The environment used was modified from OpenAI Gym[2] Cartpole, which has the same setup, except the motor can only apply a force of -1 or $+1$, moving the cart left or right, with no consideration for how fast or how far to move. The simplistic physical model uses forward Euler's method for computing future states, which is inherently unstable, adding energy to the system over long time periods. The lack of important physical concepts like friction make this an interesting toy problem for testing algorithms, but a poor representation of the real world.

The first thing we did was update the equations of motion to better reflect reality, using the equations for horizontal and angular acceleration from [31] that were created to be as physically accurate as possible, by incorporating friction, the electrodynamics of the motor, and the viscous damping coefficient as seen at the motor pinion and pendulum axis. Additionally, the action space was updated to allow a continuous voltage between -10V and $+10$V. Finally, the update scheme was changed to a modified version of semi-implicit Euler's method, which almost conserves energy, and prevents the system from becoming unstable over long time intervals.

The equations of motion are recreated from Section 2.2 of [31] here:

[1] 119 Spy Court Markham, Ontario, L3R 5H6, Canada.
[2] `https://gym.openai.com/`

$$
\begin{aligned}
\ddot{x}(t) = &-\frac{3r_{mp}^2 B_p \cos(\alpha(t))\dot{\alpha}(t)}{\ell_p D(\alpha)} - \frac{4M_p \ell_p r_{mp}^2 \sin(\alpha(t))\dot{\alpha}(t)^2}{D(\alpha)} \\
&- \frac{4(R_m r_{mp}^2 B_{eq} + K_g^2 K_t K_m)\dot{x}(t)}{R_m D(\alpha)} + \frac{3M_p r_{mp}^2 g \cos(\alpha(t))\sin(\alpha(t))}{D(\alpha)} \\
&+ \frac{4r_{mp} K_g K_t V_m}{R_m D(\alpha)},
\end{aligned} \tag{1.25}
$$

$$
\begin{aligned}
\ddot{\alpha}(t) = &-\frac{3(Mr_{mp}^2 + M_p r_{mp}^2 + J_m K_g^2) B_p \dot{\alpha}(t)}{M_p \ell_p^2 D(\alpha)} - \frac{3M_p r_{mp}^2 \cos(\alpha(t))\sin(\alpha(t))\dot{\alpha}(t)^2}{D(\alpha)} \\
&- \frac{3(R_m r_{mp}^2 B_{eq} + K_g^2 K_t K_m)\cos(\alpha(t))\dot{x}}{R_m \ell_p D(\alpha)} + \frac{3(Mr_{mp}^2 + M_p r_{mp}^2 + J_m K_g^2) g \sin(\alpha(t))}{\ell_p D(\alpha)} \\
&+ \frac{3r_{mp} K_g K_t \cos(\alpha(t)) V_m}{R_m \ell_p D(\alpha)},
\end{aligned} \tag{1.26}
$$

where $D(\alpha) = 4Mr_{mp}^2 + M_p r_{mp}^2 + 4J_m K_g^2 + 3M_p r_{mp}^2 \sin^2(\alpha(t))$, and the position is updated via semi-implicit Euler's method:

$$
\begin{aligned}
\dot{x}_{t+1} &= \dot{x}_t + h\ddot{x}_t, \\
\dot{\alpha}_{t+1} &= \dot{\alpha}_t + h\ddot{\alpha}_t, \\
x_{t+1} &= x_t + h\dot{x}_t, \\
\alpha_{t+1} &= \alpha_t + h\dot{\alpha}_t,
\end{aligned} \tag{1.27}
$$

where $h = \frac{1}{50}$ is the time step-size used for implicit Euler.

Parameter values were either measured directly from the pendulum or calibrated using parameter identification. A complete list of values can be found in Appendix A.

1.2.2 Background

With the current generation of graphics cards offering the "greatest generational leap ever," [26] advances in the speed and size of graphics processing units (GPUs) and tensor processing units (TPUs) have made it possible for current top-of-the-line graphics cards to have similar compute power as the world's fastest supercomputers up until 2004 (Table 1.1). This has ushered in an era of massive growth in machine learning, with the two-year doubling of compute, consistent with Moore's law abruptly switching to a 3.4-month doubling time around 2012 [4], starting with AlexNet's highly-optimized GPU implementation of 2D convolutions [35].This coincided with the release of the GeForce 600

Table 1.1 Modern graphics cards compared to the fastest supercomputers from 2000–2004.

Supercomputer	Cores	TFLOPS (Max)	Precision	Year
IBM ASCI White	8,192	7.226	FP64	2000
NEC Earth Simulator	5,120	35.86	FP64	2002
Consumer GPU	**Shaders**	**TFLOPS (Max)**	**Precision**	**Year**
NVIDIA A100	6,912	9.75	FP64	2020
NVIDIA GeForce RTX 3090	10,496	35.58	FP32	2020
AMD Radeon RX 6900 XT	5,120	46.08	FP16	2020

series graphics cards and emerging mainstream use of GPUs as highly parallel multi-core systems for big data. Since then, training the largest AI training runs have used more than $600,000\times$ more compute than training AlexNet [10], even as algorithmic efficiency has resulted in a $44\times$ reduction in compute needed to obtain the same performance [25].

These massive gains in compute have not extended to robotic reinforcement learning problems though, with the primary method of improving training being the development of new algorithms, and collecting more data—an incredibly slow and expensive process. Google used Q-learning with deep neural network function approximators called QT-Opt to collect 800 hours of data, parallelized over 7 grasping robots for 4 months [28]. That is impractical for most applications. Therefore, the novel approach proposed here is to leverage the enormous compute provided by modern GPUs to train models entirely through simulation, after which the trained model can be transferred to a real world agent, and fine-tuned if necessary.

Using reinforcement learning to train a single inverted pendulum is a proof of concept that could make other reinforcement learning applications more efficient, letting computers quickly and efficiently train models before implementing them in the real world. As most cartpole implementations are either simulations with simplified models or model-free implementations in the real world, this research is unique in that it we can balance real inverted pendulums using neural networks trained entirely in a computer.

1.3 Reinforcement Learning

Reinforcement learning (RL) is an area of machine learning focused on using trial-and-error to interact with a system, performing actions and obtaining rewards quantifying how good that action is. These rewards encourage the agent to take the optimal action when presented with the state of its environment, with the goal of maximizing cumulative rewards, known as the *return*. RL is often used to optimize and automate physical

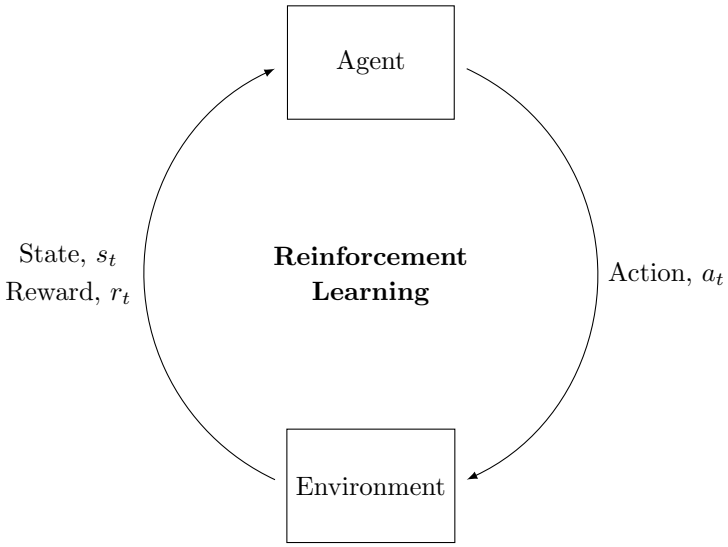


Figure 1.5 Agent-environment interaction loop typical of reinforcement learning.

processes, but most of the time ends up being used for purely virtual tasks.

Most RL problems can be formulated as a type of Markov Decision Process (MDP) described by a set of states, actions, and a process for describing transition behaviour. Formally, an MPD is a tuple (S, A, p, R, γ), where S is a finite set of states, A is a finite set of actions, p is a state transition probability matrix describing the probability of moving from state s to s' given action a such that

$$p(s, a, s') = \mathbb{P}\left[S_{t+1} = s' | S_t = s, A_t = a\right], \tag{1.28}$$

R is a reward function giving the expected rewards from state s given action a so

$$R(s, a) = \mathbb{E}\left[R_{t+1} | S_t = s, A_t = a\right], \tag{1.29}$$

and γ is a discount factor between 0 and 1.

The goal is to find a deterministic policy $\pi : S \to A$ such that $\pi(s) = a$, or a stochastic policy $\pi : S \times A \to [0, 1]$ such that

$$\pi(a|s) = \mathbb{P}\left[A_t = a | S_t = s\right]. \tag{1.30}$$

An optimal policy π^* is one that maximizes expected rewards or minimizes expected costs over any further steps, sometimes called the *reward-to-go*. Then the optimal action-value

function maximizes this over all policies so that

$$Q^*(s,a) = \max_\pi Q_\pi(s,a) \tag{1.31}$$

and all optimal policies achieve the optimal action-value function,

$$Q_{\pi^*}(s,a) = Q^*(s,a). \tag{1.32}$$

In classical Q-learning, this can be done by defining an update rule in the form of a Bellman equation

$$\begin{aligned} Q_{k+1}(s,a) &= Q_k(s,a) + \eta \left(R(s,a) + \gamma \max_a Q_k(s',a) - Q_k(s,a) \right) \\ &= (1-\eta) Q_k(s,a) + \eta R(s,a) + \eta\gamma \max_a Q_k(s',a), \end{aligned} \tag{1.33}$$

for the state-action value function, describing the maximum expected future rewards for any (state, action) pair. For finite state and action-spaces, a table of Q-values be made and updated using this equation. When S is continuous, the process is called least-squares fitted Q-iteration, and an error function like $J = \left(Q(s,a) - \left(R(s,a) + \gamma \max_a Q_k(s',a)\right)\right)^2$ can be minimized using backpropagation and gradient descent [53].

Algorithm 1: Training with classical Q-learning.

randomly initialize $Q(s,a)$ for each $s \in S$ and $a \in A$;
repeat
  initialize s;
  choose $a = \arg\max_a Q(s,a)$;
  observe R;
  compute s';
  update $Q_{k+1}(s,a) = Q_k(s,a) + \eta \left(R(s,a) + \gamma \max_a Q_k(s',a) - Q_k(s,a)\right)$;
until $Q(s,a)$ *converges or* s *is terminal*;

Theorem 1.6

Given a finite MDP (S, A, p, R, γ) *and a (possibly variable)* $0 \leq \eta_t < 1$*, the Q-learning algorithm in Equation 1.33 converges to the optimal Q-function with probability 1, if*

$$\sum_t \eta_t = \infty \quad \textit{and} \quad \sum_t \eta_t^2 < \infty. \tag{1.34}$$

The proof can be found in [68]. Theorem 1.6 guarantees that $Q_k(s,a) \to Q^*(s,a)$ as $k \to \infty$ with probability 1. However, this requires a decreasing learning rate η_t, and that each (state, action) pair is visited infinitely often. Classical Q-learning has the downside of requiring infinite iterations until an optimal policy is found, or possibly tens of thousands to find one that is "good enough," which is unrealistic for many applications. Simulation, and larger learning rates can help alleviate this issue. In practice, a small constant learning rate η is used in Algorithm 1 with an ε-greedy policy that takes random actions with probability ε instead of optimal ones, in order to explore the space and find a near-optimal solution more quickly [65].

A variation of Equation 1.33 involves creating a second Q-function to estimate the value of $\max_a Q_k(s', a)$, so that one policy is used for value evaluation and one is used to select the next action. This generally speeds up convergence, especially in noisy environments. However, this algorithm only works when both S and A are finite, so will not suffice for our implementation on an inverted pendulum unless we discretize the spaces.

1.4 Literature Review

Single inverted pendulums are ubiquitous benchmark problems used for testing various controls, due to their nonlinear equations of motion, underactuated pole, and unstable target equilibrium. They represent one of the real-world control problems listed by the International Federation of Automatic Control (IFAC) Theory Committee for testing new and existing control methods [15]. Because the model can be simulated to a high degree of accuracy, computer simulations are often used to test these controls instead of real pendulums, as they can be expensive to obtain, time-consuming to set up, and labourious to maintain. Due to their ubiquity , there have been a plethora of classic control techniques used to balance them. These techniques generally fall into three categories.

1.4.1 Traditional Controls

It is essentially impossible to exactly recreate a perfect simulation of a real inverted pendulum in a computer. While things like friction, motor electrodynamics, and viscous damping coefficients can be included to make the simulation match reality better, various attempts to do this still neglect the nonlinear Coulomb friction applied to the cart, and the force on the cart due to the pendulum's action [20, 31]. In addition, variances in manufacturing tolerances, imbalances in the setup of the cart, play in the motor, and general wear and tear on the gears all serve to add unpredictable noise that may cause

simulated controls to fail, or to perform worse than the simulation predicts. Finally, nonlinear controllers need to be evaluated quickly for online implementation, posing a challenge for computationally intensive controllers. Because of this significant hurdle, far fewer inverted pendulums are constructed and balanced in real life.

A non-comprehensive list of traditional controllers include PID with a Kalman Filter [1, 42], LQR [33, 47], State-Dependent Riccati Equation [14], and power series approximation of the HJB equation [31].

Proportional-integral-derivative (PID) controllers are a staple of industrial process control. By first linearizing the space state equations about the unstable equilibrium, the error $e(t)$ between the current and target states can be fed into the PID-controller

$$u(t) = k_P e(t) + k_I \int_0^t e(\tau)\, d\tau + k_D \frac{de(t)}{dt}, \tag{1.35}$$

where k_P, k_I, and k_D are the proportional, integral, and derivative gain constants, respectively. The performance of the controller is directly dependent on the ability to find values of these constants that work for this specific system. If any of the gain constants are too high or low, the system can become unstable, oscillate, or fail to balance the pole at all.

Linear-quadratic regulator (LQR) controllers also begin by linearizing the nonlinear state equations the same way, and using the solution P to the algebraic Riccati equation to minimize the cost functional

$$J(x_0, u) = \int_0^\infty x^T Q x + R u^2\, dt. \tag{1.36}$$

Then the optimal control is $u = -R^{-1} B^T P x$. Like the PID controller, the performance of LQR depends directly on the values chosen in Q and R; this can lead to oscillations or unstable performance if they are off.

In both PID and LQR, a Kalman filter is used to smooth sensor inputs, and provide an approximation of $\dot{x}$ and $\dot{\alpha}$ in the state, although a low-pass derivative filter may work better if the optimal Kalman parameter values cannot be obtained [6]. The downside of these controllers is that the linearized state equations differ from the true nonlinear equations increasingly as the state deviates the unstable equilibrium at the origin. While they perform well when the pole is essentially straight up, a nonlinear controller may perform better over a wider range of values, such as during a disturbance.

One solution to this is to use a nonlinear formulation like the state-dependent Riccati

equation (SDRE). Much like LQR, SDRE starts by solving

$$A(x)^T P(x) + P(x)A(x) - P(x)B(x)R(x)^{-1}B(x)^T P(x) + Q(x) = 0 \tag{1.37}$$

to get $P(x) \geq 0$, which is used for the control $u = -R(x)^{-1}B(x)^T P(x)x$. The main difference between LQR and SDRE is that the design matrices $Q(x)$ and $R(x)$ and the plant matrices $A(x)$ and $B(x)$ are state-dependent instead of constant. This allows the same controller to be used to swing-up and balance the pendulum, instead of switching controllers, as the linearized models would require. The major downside of this technique is that it requires solving the algebraic Riccati equation at every time step, which can be computationally expensive and limit the sampling rate. As long as the online compute resources are fast enough for real time implementation, this results in a controller that is more stable and robust.

1.4.2 Reinforcement Learning Controls

For a control method to be effective, it must be robust to model errors, sensor noise, and potential disturbances of the system. Reinforcement learning algorithms can be used as controllers because they can generalize to new environments, and do not require domain specific knowledge of the system to achieve satisfactory results; the same algorithm can be used to balance a pole, drive a car, or play Pong. Due to the costs associated with working in a real environment, most reinforcement learning algorithms are trained virtually. There have been several attempts at using various reinforcement learning techniques to control a real inverted pendulum, but they all suffer from the same issue: most reinforcement algorithms require a lot of data to perform adequately, and getting data from a real pendulum is a slow and expensive process.

1.4.2.1 Virtual Training

Through the development of virtual environments like OpenAI Gym, it is now possible to test reinforcement learning algorithms on a collection of standardized environments. OpenAI Gym's Cartpole[3] provides a simplified simulation of an inverted pendulum environment, lacking friction, and featuring a discrete left/right action-space. The original environment was defined in 1983 [5] and used an Adaptive Critic method to greatly improve on the Boxes algorithm used to teach a computer-analogue to play noughts and crosses (tic-tac-toe).

[3] `https://gym.openai.com/envs/CartPole-v0/`

The Boxes algorithm [41] starts by discretizing the state space into 225 regions or "boxes," creating several hundred independent sub-problems that could each be solved, finding the optimal action to take any time the pole was in a specific box. The algorithm is remarkably simple:

Algorithm 2: Boxes algorithm used for Tic-Tac-Toe or Cartpole.

initialize state s_t and identify which box $B(s_t)$ it is in;
while *the agent survives* **do**
    choose action a_t such that lifetime $Q(B(s_t), a_t)$ is maximized;
    proceed to the next state s_{t+1} of the environment;
end
Update $Q(B(s_t), a_t)$ for every action taken in box $B(s_t)$

This is just Q-learning from Algorithm 1, with $\eta = 1$ and $\gamma = 0$. This can be repeated until the algorithm becomes optimal (i.e. the algorithm can consistently balance the pole). The performance is understandably poor, as the discretization requires taking the same action when presented with different states in the same box. This ends up being a lose-lose trade-off between algorithmic accuracy and computation time, as refining the boxes results in a more computationally challenging problem due to the curse of dimensionality, but further discretizing the boxes erodes performance.

In the decades since, the emergence of reinforcement learning has lead to several other techniques, including vanilla Policy Gradient, Actor-Critic, and Deep Q-Learning being used with varying degrees of success. However, basic reinforcement learning algorithms like these tend to be slow and data-intensive, requiring thousands of trials for the policies to converge. Even so, the ease of use and implementation of OpenAI Gym has lead to the environment being used as the *de-facto* way of benchmarking the success of new algorithms like Deep Deterministic Policy Gradient [37], Trust Region Policy Optimization [59], and Proximal Policy Optimization [60]. These algorithms are benchmarked on a variety of OpenAI environments in [19]. However, the discrete action space and simplified dynamics of the system make it difficult to replicate the results in the real world. To use reinforcement learning to balance a real pendulum, the model is usually trained from scratch on a real pole.

1.4.2.2 Real World Training

Generally, the algorithms used to train a model in a simulation perform poorly in the real world. Running thousands of trials on a real pole with physical resets is excruciatingly slow, making many these same algorithms intractable in practice. They all suffer from the same issue: most reinforcement algorithms require a lot of data to perform adequately, and getting data from a real pendulum is a slow and expensive process. Although some data-efficient algorithms are better than others, requiring only a few seconds or minutes of *online* time interacting with the pendulum, there is a severe computational burden: the total time training can be orders of magnitude higher.

One technique that performs well is called Neural Fitted Q-Iteration [53], which does not require a model and is relatively data-efficient. This technique is a modification of a tree-based regression method using neural networks with an enhanced weight update that can learn directly from real-world interactions; it is a Markov Decision Process. Unlike traditional Q-learning, which requires an overwhelming number of iterations until a near-optimal policy is found, Neural Fitted Q-Iteration uses offline learning, enabling more advanced batch supervised learning algorithms like Rprop [52] to use the entire set of transition experiences when updating the neural network parameters. This drastically reduces the amount of data required, proportionally reducing training time on a real agent or in simulation. As a bonus, Rprop is insensitive to the learning parameters, meaning time-consuming hyperparameter optimization is not necessary to achieve quick learning.

Alternative data-efficient model-based strategies like Probabilistic Inference for Learning Control (PILCO) [16] are also able to learn effective policies with very little data by incorporating model uncertainty with a probabilistic dynamics model for long-term planning. Essentially, this algorithm creates a probabilistic model using very few data points, and refines the model as more data is collected. Ultimately, this results in a drastic reduction in required training to achieve satisfactory results.

1.4.3 Virtual Training

While reinforcement learning algorithms can be benchmarked in a virtual environment, unless care is taken to closely match the environment to reality, it can be difficult to replicate a model's success on a real agent. There is already work being done to bridge the *simulation-reality gap* of training agents in a simulation for real-world deployment.

Researchers at OpenAI are using two interconnected long short-term memory neural networks defining a value function and agent policy to be trained in simulation and implemented in the real world to solve a Rubik's cube using a robotic hand [45]. They are

using Proximal Policy Optimization to train the control policies and use automatic domain randomization to train on a large distribution over randomized environments. Despite not being able to model accurate physics in the simulations, domain randomization allows them to successfully solve a cube 60% of the time.

Researchers at Microsoft are using AirSim to generate simulated training data for drones to navigate obstacle courses by imitating expert behavior [8]. Specifically, they encode images with an 8-layer Resnet and decocde them with six transpose convolutional layers. By using a cross-modal variational autoencoder, they are able to learn a rich low-dimensional state representation from simulated or real data, which enables the drones to learn a control policy that directly transfers to real-world environments. The learned policy is robust to changes in weather, lighting, and background changes.

Researchers from deepsense.ai collaborated with Volkswagen AG to train a parallelized architecture with CARLA, a high-fidelity model-based driving simulator, to generate up to 100 years of synthetic semantic segmentation data for training 10 virtual models [46]. The models were trained using a parallelized version of Proximal Policy Optimization on GPUs, and were implemented in 9 driving scenarios over the course of over 400 test drives with a real car. This paper also contains an excellent review of existing reinforcement learning methods that use synthetic data for both simulated and real-world robotics.

Researchers from MIT are using standard Policy Gradient techniques with a multi-layer convolutional neural network to train a car to drive up to 10km [3]. They used a variety of different simulation techniques as baselines, including imitation learning from an expert human and training in CARLA, using domain randomization and domain adaptation in order to bridge the simulation-reality gap. Finally, they trained the same agent on a data-driven simulator VISTA, which uses images from real driving data to simulate the camera views from new trajectories along the same route. This provided unprecedented levels of realism to the simulation, allowing the incredibly basic control algorithm to achieve near-human performance, generalizing across changes in lighting, weather, and road type. The models trained in VISTA required no human interventions, even on unfamiliar roads, and showed the ability to recover when placed in near-crash situations.

1.5 Discussion

Our agent is a single inverted pendulum, which we teach to balance by using reinforcement learning to train a parametrized policy π_θ in a simulation. Once training is finished, we can recreate π_θ in MATLAB in a Simulink block. At this point, the model is built with

Quanser QUARC, loaded to the controller, and tested on a real inverted pendulum.

The idea of using virtual reinforcement learning to train a single inverted pendulum is just a proof of concept that could make other machine learning applications more efficient. If successful, these algorithms could be used for complex robotics, reusable rockets, and other expensive or dangerous agents where it would be prohibitively difficult to test new controls on a real plant. The goals are twofold:

- find the algorithm that will train a virtual pole to balance as quickly as possible,
- find the algorithm that is the most robust for a real inverted pendulum.

These goals are necessarily at odds with one another, as increased training time often leads to increased robustness. By training in a simulation we can drastically speed up the training process, as we can spend more time learning, and less time resetting the experiment and running slow trials. This extends the lifetime of the agent, reducing the amount of wear and tear it experiences, and allows it to experience important edge-case environmental states that would be rare or dangerous in the real world—like a car avoiding a collision.

The rest of this dissertation focuses on the implementation of four reinforcement learning algorithms trained with this virtual paradigm, and the results when transferred to the real world.

Chapter 2

REINFORCEMENT LEARNING

One of the ways we learn is by interacting with our environment and observing the outcome of such an interaction. Depending on the task, there may be an implicit "reward" associated with the action, like the distance travelled for a child learning to ride a bike, or the score achieved while playing a video game. Reinforcement learning is the process of an agent exploring an environment, and using trial-and-error to maximize the numerical reward associated with the actions it takes. Put simply: we drop a robot into an unfamiliar environment, and it learns to take the optimal actions through repeated efforts, without being told which actions to take. The most common reinforcement learning problems can be formulated as Markov Decision Processes (MDPs). MDPs are described by a set of states S, actions A, rewards R and a process $p(s_t, a_t, s_{t+1})$ for describing transition behaviour. The goal is to find an optimal policy $\pi^* : S \rightarrow A$ that maximizes expected rewards (or minimizes expected costs) for each state. While some MDPs can be solved using dynamic programming or linear programming techniques, reinforcement learning presents two key advantages: sampling and function approximation.

By sampling actions from the action-space, we can simplify large-dimensional problems and optimize performance, since we do not have to find optimal trajectories for every possible action. For continuous action-spaces, this can make difficult problems easy, and impossible problems possible. Additionally, due to the universal function approximation properties of neural networks mentioned in Chapter 1, it is possible to solve problems in large state spaces, where it would be infeasible to obtain an analytical solution due to the curse of dimensionality.

As seen in Figure 2.1, there are myriad reinforcement learning algorithms that we can use to attempt to navigate unfamiliar environments, each of which has its own advantages, disadvantages, and quirks. In the rest of this chapter, we will be focusing on model-free implementations, where we observe the states and rewards as they occur, instead of

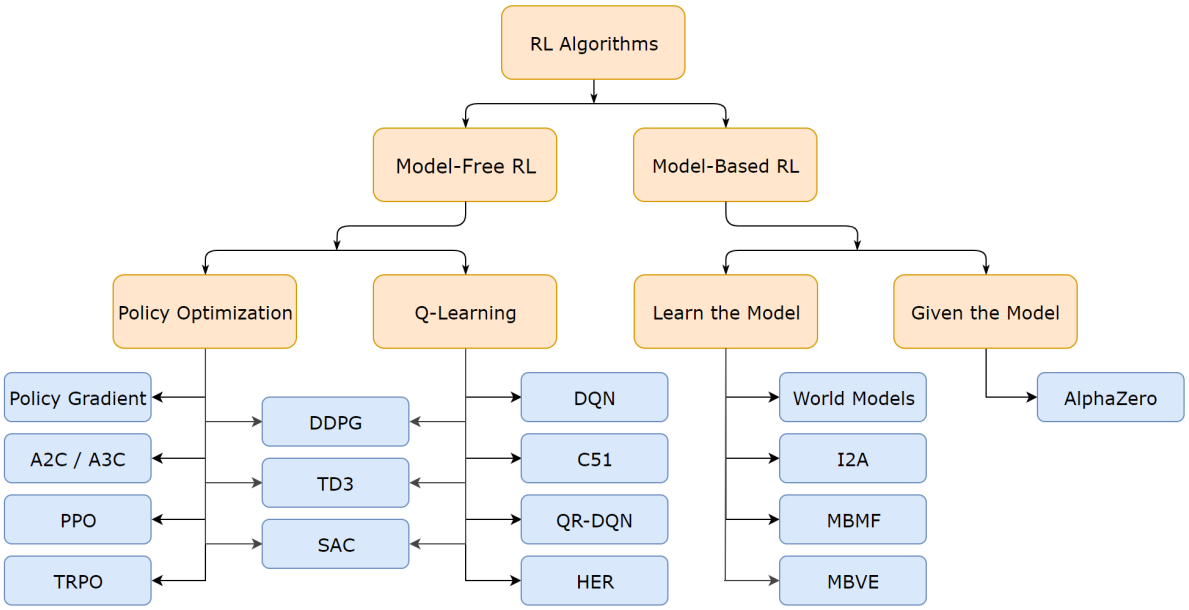


Figure 2.1 A non-exhaustive taxonomy of some Reinforcement Learning algorithms from [2].

assuming we have a full set of transition probabilities to optimize. This lets us perform *online* learning, observing states directly from an agent (or simulation). Additionally, since we have continuous state and action-spaces, we will primarily focus on policy-based algorithms, where we directly optimize the same policy we use to select actions (called an *on-policy* algorithm), as opposed to optimizing the value function like in Q-learning.

This dissertation will give a detailed description of 4 of these algorithms, presented in order of increasing complexity.

2.1 Preliminaries

The goal of reinforcement learning is to make an agent choose the optimal policy to maximize the rewards it will receive. In our case, the agent is the inverted pendulum, the policy is an artificial neural network with one hidden layer that decides how much voltage to apply to the motor, and the rewards correspond to how long the pole stays balanced.

2.1.1 Policies

We define a deterministic policy $\pi : S \to A$, defining what action to take given the state of the environment. We also define a stochastic policy $\pi : S \times A \to [0, 1]$, defining a

distribution over actions for each state, such that

$$\pi(a|s) = P\left[A_t = a|S_t = s\right]. \tag{2.1}$$

In this section, we will use a neural network parametrized by weights and biases collectively referred to as θ to describe a policy that inputs the state of the environment and a probability distribution of actions actions. We define π_θ as a one-hidden-layer artificial neural network seen in Figure 2.2 so that

$$\pi_\theta(a|s) = P\left[a|s, \theta\right]. \tag{2.2}$$

The input layer consists of four neurons, corresponding to the state vector $[x, \alpha, \dot{x}, \dot{\alpha}]$. This feeds forward to the 32-neuron hidden layer, with `ReLU` activation. 32 neurons were chosen somewhat arbitrarily, as it was more than enough to regularly balance the pole. While more neurons (or more layers) may result in faster training speed, neither are necessary for the agent to learn to balance itself. See Section 4.1.2 for further discussion on this.

The top layer consists of two neurons defining the normal probability distribution $N(\mu, \sigma)$ the actions are sampled from. The first neuron uses linear activation and corresponds to the mean of the distribution, while the second neuron uses Softplus ($\ln(1 + e^x)$) activation (guaranteeing it will be positive) and corresponds to the standard deviation of the normal distribution.

Definition 2.1. A *trajectory* τ is a sequence of states and actions

$$\tau = (s_0, a_0, s_1, a_1, \dots). \tag{2.3}$$

The first state s_0 is sampled from the starting distribution ρ_0, then at each time step $t > 0$, the actions are sampled from the policy π and the state is updated from the Equations of Motion 1.25–1.27.

Let π_θ a policy with parameters θ. Given a trajectory of states and actions $\tau = (s_0, a_0, \dots, s_{T+1})$, we define the probability of τ given actions sampled from π_θ to be

$$P\left(\tau|\theta\right) = \rho_0(s_0) \prod_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t, s_t). \tag{2.4}$$

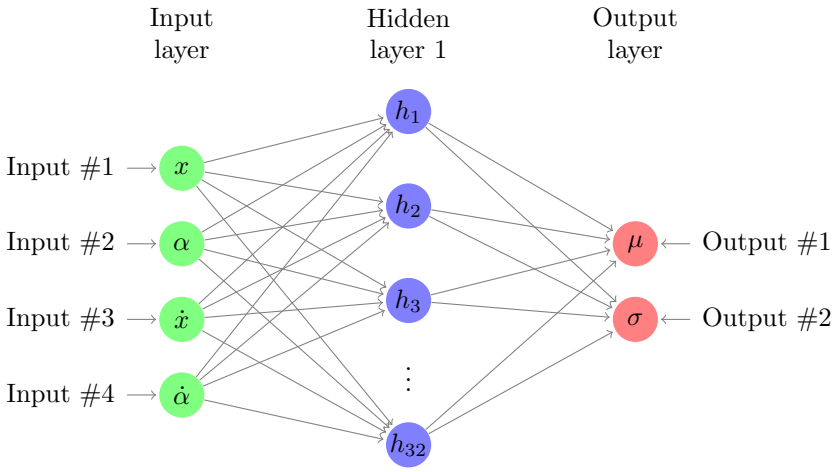


Figure 2.2 Artificial neural network approximation of π_θ.

2.1.2 Rewards

Formally, the reward function $r_t = R(s_t, a_t, s_{t+1})$ can depend on the current state, action, and future state of the environment. For an inverted pendulum, it is much simpler. After initializing the reward function as an empty vector $r = [\,]$, for every time step t that the pendulum remains upright, we set $r_t = 1$. Specifically, if the pole remains between $\pm 12°$ and the cats stays within ± 0.4m of the centre of the track, we set $r_t = 1$.

Given a trajectory τ, our goal is to choose the policy π_θ that allows the pole to remain balanced by maximizing

$$R(\tau) = \sum_{t=0}^{T} r_t. \tag{2.5}$$

This sum diverges over infinite timescales. A common technique to prevent this (and to emphasize getting rewards now rather than later in the future), involves the idea of a *discounted* reward. That is, we define $\gamma \in (0, 1]$ so that our cumulative discounted reward at time step t is

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}. \tag{2.6}$$

This is known as the *rewards-to-go*, as it gives us the sum of discounted future rewards. In addition to guaranteeing the sum converges, discounting has the additional benefit of prioritizing rewards now over rewards later. The value of γ can be tuned to provide the optimal combination of current and future rewards. Small values of γ prioritize actions resulting in rewards now, while larger values prioritize rewards over a longer time frame. If we normalize R by subtracting the mean and dividing by the standard deviation, we

can simultaneously encourage and discourage roughly half of the possible actions [29].

Definition 2.2. The *expected return* for a policy π is

$$\begin{aligned} J(\pi) &= \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)] \\ &= \int_{\tau} P(\tau|\pi)R(\tau), \end{aligned} \tag{2.7}$$

judging the performance of the given policy.

This is the quantity that we want to maximize. Under the policy-based algorithms in this chapter, that means we are looking for the *optimal policy* π^* such that

$$\pi^* = \arg\max_{\pi} J(\pi). \tag{2.8}$$

However, since these algorithms are model-free, we do not have complete knowledge of the set of transition probabilities allowing us to simply maximize the function. By computing the gradient of this policy performance, also known as the *policy gradient*, we can use gradient descent (or technically gradient *ascent*) to find the set of parameters θ that maximize this quantity:

$$\theta_{k+1} = \theta_k + \eta\nabla_\theta J(\pi). \tag{2.9}$$

2.1.3 Training

For each of the algorithms in this section, the general outline is roughly the same: the agent interacts with the environment, acting according to a policy π_θ until an exit condition is met—in our case running off the track ($|x| > 0.4$) or falling ($|\alpha| > 12°$). When that happens, we use our normalized discounted rewards and loss function specific to that algorithm to update the parameters θ using backpropagation with gradient ascent. This encourages the agent to take actions that increase $J(\pi_\theta)$, and discourages the agent from taking actions that decrease it, hopefully balancing the pole for longer next time.

As seen in Algorithm 3, the model is fed the current state s_t and outputs a probability distribution of actions to be taken. The policy π_θ samples an action a_t from the distribution, clips it to $[-10, 10]$ due to the physical limitations of the motor, and uses the Equations of Motion 1.25–1.27 to step forward in time, generating the new state s_{t+1}. If the pole remains balanced and the cart is still on the track, then $r_t = 1$ is appended to the rewards vector and training continues. If the model fails to meet these criteria, training stops, the rewards are discounted and normalized, and the parameters θ are updated using gradient

Algorithm 3: Pseudocode for training all the algorithms in this chapter.

initialize state s_0 so $\{x, \alpha, \dot{x}, \dot{\alpha}\} \in [-0.05, 0.05]$;
initialize $t = 0$ **repeat**
  choose action by sampling $a_t \sim \pi(\,\cdot\,|s_t)$;
  clip a_t to $[-10, +10]$;
  calculate s_{t+1} using Equations of Motion;
  if $|x| > 0.4$ *or* $|\alpha| > 12^\circ$ **then**
    Crashed;
    compute log-likelihoods $\ln \pi_\theta(a_t|s_t)$;
    normalize discounted rewards $R_t = \sum_k \gamma^k r_{t+k}$;
    compute loss;
    update parameters using with gradient ascent or Adam optimizer;
    $r \equiv 0$, $t = 0$;
  else
    $r_t = 1$, $t := t + 1$;
  end
until $\sum_t r_t = 500$ *and training is finished*;

ascent with the Adam optimizer [32]; we found a learning rate of $\eta = 0.01$ tended to perform the best.

Training is considered finished when the undiscounted finite-horizon return exceeds 500. In practice, it would be better to keep track of a running reward value

$$R_r := (1 - \varepsilon)R_r + \varepsilon R(\tau), \tag{2.10}$$

initialized at 0 that will exceed say $R_r > 475$ when the agent consistently balances for 500 time steps (balancing for 10 seconds) without crashing. Depending on the hyperparameters chosen for the learning rate and γ, this can take anywhere from 19 individual trials—taking well under a minute on Google Colab's[1] Tesla GPUs—to well over 2000 trials, testing the free resource limits and my patience. At this point, optional additional training with a smaller learning rate can help make the model more robust.

Once training is finished, the weights and biases of π_θ can be loaded into MATLAB and the neural network recreated in a single block in Simulink. At this point, the model can be built with Quanser QUARC, loaded to the controller, and tested on a real inverted pendulum.

[1] `https://research.google.com/colaboratory/faq.html`

2.2 Policy Gradient

Vanilla Policy Gradient is one of the simplest methods in reinforcement learning [2]. Sometimes called the REINFORCE algorithm [65], it maximizes the expected return for a policy by reinforcing actions that lead to longer lifetimes. It does this by shifting the parameters θ in such a way that increases $J(\pi_\theta)$ with the goal of maximizing Equation 2.7. Therefore, we compute the analytical policy gradient with respect to each of the weights and biases in θ, so we can shift them in the direction that maximizes the expected return.

To this end, the gradient of $J(\pi_\theta)$ for a trajectory $\tau = (s_0, a_0, \ldots, s_{T+1})$ is

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)] \\
&= \nabla_\theta \int_\tau P(\tau|\theta) R(\tau) \\
&= \int_\tau \nabla_\theta P(\tau|\theta) R(\tau) \\
&= \int_\tau \frac{P(\tau|\theta)}{P(\tau|\theta)} \nabla_\theta P(\tau|\theta) R(\tau) \\
&= \int_\tau P(\tau|\theta) \frac{\nabla_\theta P(\tau|\theta)}{P(\tau|\theta)} R(\tau) \\
&= \int_\tau P(\tau|\theta) \nabla_\theta \ln P(\tau|\theta) R(\tau) \\
&= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\nabla_\theta \ln P(\tau|\theta) R(\tau)] .
\end{aligned} \tag{2.11}$$

Expanding the gradient of $\ln P(\tau|\theta)$ in the last step, we get

$$\begin{aligned}
\nabla_\theta \ln P(\tau|\theta) &= \nabla_\theta \left[\ln \rho_0(s_0) + \sum_{t=0}^{T} \left(\ln P(s_{t+1}|s_t, a_t) + \ln \pi_\theta(a_t|s_t) \right) \right] \\
&= \nabla_\theta \ln \rho_0(s_0) + \sum_{t=0}^{T} \left(\nabla_\theta \ln P(s_{t+1}|s_t, a_t) + \nabla_\theta \ln \pi_\theta(a_t|s_t) \right) \\
&= \sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(a_t|s_t),
\end{aligned} \tag{2.12}$$

where the gradients of any term not depending on θ are 0, so that

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(a_t|s_t) R(\tau) \right] . \tag{2.13}$$

This is the gradient of the cost function used for gradient ascent in Algorithm 4.

Algorithm 4: Training with vanilla Policy Gradient

input : a parametrized policy $\pi_\theta(a_t|s_t)$
Initialize parameters θ and state $s_0 \sim \rho_0$;
repeat
  sample a trajectory τ from $\pi_\theta(a_t|s_t)$;
  $\nabla_\theta J(\pi_\theta) = \sum_t \nabla_\theta \ln \pi_\theta(a_t|s_t) R(\tau)$;
  $\theta := \theta + \eta \nabla_\theta J(\pi_\theta)|_\theta$;
until *training is complete*;

At this point, we can choose any arbitrary advantage function, relative to the current policy, depending on the current action and state, giving us

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t) \right]. \tag{2.14}$$

In general, we should choose our advantage function $A^{\pi_\theta}(s_t, a_t)$ so that the gradient term $\nabla_\theta \ln \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t)$ points in the direction of increased $\pi_\theta(a_t|s_t)$ if and only if $A^{\pi_\theta}(s_t, a_t) > 0$. Therefore, for vanilla policy gradient we choose A^{π_θ} to be the normalized version of our discounted rewards-to-go R_t from Equation 2.6. This has three benefits: regularity, causality, and performance.

2.2.0.1 Regularity

By normalizing the discounted rewards, we are able to simultaneously encourage actions that lead to longer lifetimes, while discouraging the actions that directly lead to the pendulum failing. Tweaking the value discount factor γ plays a big role in which actions should be discouraged, which plays a big role in training speed and reliability. See Section 4.1.1 for details.

2.2.0.2 Causality

If we replaced the rewards-to-go with the finite-horizon return

$$\hat{R}(\tau) = \sum_{t=0}^{T} \gamma^t r_t, \tag{2.15}$$

then each step increases the log-probabilities of each action proportional to $\hat{R}(\tau)$, the discounted sum of all rewards ever obtained. Due to causality, an action at time t has no bearing on rewards obtained at any times $t' < t$. By discarding terms that reinforce actions proportional to past rewards, we reduce the variance in the estimates of the policy gradient, therefore reducing the number of trajectories needed.

For completeness, we can rewrite the policy gradient in Equation 2.14 as

$$\begin{aligned} \nabla_\theta J(\pi_\theta) &= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t)\right] \\ &= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} R_{t'}(s_{t'}, a_{t'}, s_{t'+1})\right], \end{aligned} \tag{2.16}$$

where $R_{t'}(s_{t'}, a_{t'}, s_{t'+1})$ is the normalized version of the discounted rewards in Equation 2.6.

2.2.0.3 Performance

The calculated loss function $J(\pi_\theta)$ does not estimate our performance metric that we wish to maximize (expected return) even in expectation. Rather, given the current parameters and actions sampled from the current parameters, the loss function has a local positive gradient of performance. It offers no guarantees that gradient ascent will continue to improve performance from the updated parameters. In fact, the loss function can explode to infinity while performance craters.

Unlike Q-learning, which converges to the optimal Q-function under the right conditions, vanilla Policy Gradient is only guaranteed to converge to a *local* optimum, given a small enough learning rate η [65]. Experimentally, smaller step sizes lead to more consistent convergence, while larger step sizes learn faster, but with higher variance.

2.3 Actor-Critic

One of the largest problems that Policy Gradient faces is the large variance and noisy gradients. Due to the random sampling of actions, there is a high variability in log probabilities of the actions and therefore rewards. Given an initial sample, the possible subsequent trajectories can deviate wildly, with actions sampled from the same distribution producing different future states that end the run at different times. During backpropagation this can lead to noisy gradients, causing learning to take excruciatingly long, or unstable performance if training is ended too soon. The goal of Actor-Critic is to reduce the variance by subtracting a baseline from the rewards, while keeping the value in expectation the

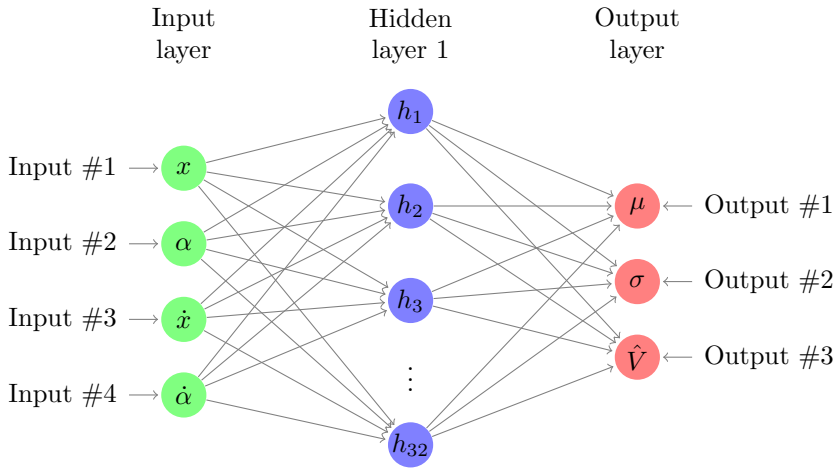


Figure 2.3 Neural network approximation of π_θ, representing Actor and Critic.

same, avoiding bias.

Therefore, we want to subtract a baseline $b(s_t)$ that depends only on the state so that

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(a_t|s_t)\left(\sum_{t'=t}^{T} R_{t'} - b(s_t)\right)\right]. \tag{2.17}$$

We begin the same way as Policy Gradient but with two neural networks: actor and critic. The actor is our policy π_θ, parametrized by the same neural network as Figure 2.2 and predicting the optimal actions to take, and the critic is a value function $V(s_t)$ judging the quality of the corresponding states. Since we do not know the value of $V(s_t)$, we approximate it by adding an additional head to the top layer of π_θ, as seen in Figure 2.3, and use gradient descent to minimize the difference between the predicted value and actual value of the normalized discounted rewards.

This results in an auxiliary function $\hat{V}^\pi(s_t)$ approximating the value function $V(s_t)$ used in our calculations of the advantage function $A^{\pi_\theta}(a_t, s_t)$ and for our parameter update.

Lemma 2.3 (Expected Grad-Log-Prob (EGLP))
Given a parameterized probability distribution P_θ over a random variable x we have that

$$\mathop{\mathbb{E}}_{x\sim P_\theta}\left[\nabla_\theta \ln P_\theta(x)\right] = 0. \tag{2.18}$$

Proof. Recall that all probability distributions are normalized so that

$$\int_x P_\theta(x) = 1. \tag{2.19}$$

Then taking the gradient of both sides with respect to θ gives

$$\nabla_\theta \int_x P_\theta(x) = 0. \tag{2.20}$$

.

Using the same log-derivative trick from Equation 2.11, we get

$$\begin{aligned} 0 &= \nabla_\theta \int_x P_\theta(x) \\ &= \int_x \nabla_\theta P_\theta(x) \\ &= \int_x P_\theta(x) \frac{\nabla_\theta P_\theta(x)}{P_\theta(x)} \\ &= \int_x P_\theta(x) \nabla_\theta \ln P_\theta(x) \\ &= \mathop{\mathbb{E}}_{x \sim P_\theta} \left[\nabla_\theta \ln P_\theta(x) \right]. \end{aligned} \tag{2.21}$$

□

Corollary 2.4
For any baseline function b that depends only on the current state,

$$\mathop{\mathbb{E}}_{a_t \sim \pi_\theta} \left[\nabla_\theta \ln \pi_\theta(a_t|s_t) b(s_t) \right] = 0. \tag{2.22}$$

In order to find best baseline, we begin by defining the state-action value function,

$$Q^{\pi_\theta}(s,a) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} \left[R(\tau) | s_0 = s, a_0 = a \right], \tag{2.23}$$

which gives the expected return if you start in state s, take an arbitrary action a and follow the policy π_θ to calculate all future actions and states. We also define the state value function,

$$V^{\pi_\theta}(s) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} \left[R(\tau) | s_0 = s \right], \tag{2.24}$$

which gives the expected return if you start in a state s and act according to policy π_θ.

The subtle difference between these two formulae can be expressed by the equation

$$V^{\pi_\theta}(s) = \mathop{\mathbb{E}}_{a \sim \tau} \left[Q^{\pi_\theta}(s,a) \right]. \tag{2.25}$$

We can exploit this relationship to calculate an advantage function

$$A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s), \tag{2.26}$$

describing how much better it is to take a specific action a in state s, over the expected value from randomly selecting an action according to the policy's default behaviour [58].

Then we have

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t,a_t)\right] \\
&= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(a_t|s_t) \left(Q^{\pi_\theta}(s_t,a_t) - V^{\pi}(s_t)\right)\right] \\
&= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(a_t|s_t) Q^{\pi_\theta}(s_t,a_t) - \sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(a_t|s_t) V^{\pi}(s_t)\right] \\
&= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(a_t|s_t) Q^{\pi_\theta}(s_t,a_t)\right] - \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(a_t|s_t) V^{\pi}(s_t)\right] \\
&= \nabla_\theta J(\pi_\theta) - 0,
\end{aligned} \tag{2.27}$$

due to the fact that

$$\begin{aligned}
\mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\nabla_\theta \ln \pi_\theta(a_t|s_t) Q^{\pi}(s_t,a_t)\right] &= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\nabla_\theta \ln \pi_\theta(a_t|s_t) \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[Q^{\pi_\theta}(s_t,a_t)\right]\right] \\
&= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\nabla_\theta \ln \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t,a_t)\right] \\
&= \nabla_\theta J(\pi_\theta),
\end{aligned} \tag{2.28}$$

and

$$\begin{aligned}
\mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\nabla_\theta \ln \pi_\theta(a_t|s_t) V^{\pi}(s_t)\right] &= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\mathop{\mathbb{E}}_{a_t\sim\pi_\theta}\left[\nabla_\theta \ln \pi_\theta(a_t|s_t) V^{\pi_\theta}(s_t)\right]\right] \\
&= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\mathop{\mathbb{E}}_{a_t\sim\pi_\theta}\left[\nabla_\theta \ln \pi_\theta(a_t|s_t)\right] V^{\pi_\theta}(s_t)\right] \\
&= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[0 \cdot V^{\pi_\theta}(s_t)\right] \\
&= 0,
\end{aligned} \tag{2.29}$$

using Corollary 2.4 of the EGLP Lemma.

We choose this to be our advantage function, inserting it into Equation 2.14, so that

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[-\sum_{t=0}^{T}\nabla_\theta \ln \pi_\theta(a_t|s_t)\left(Q^{\pi_\theta}(s_t,a_t) - V^{\pi_\theta}(s_t)\right)\right], \tag{2.30}$$

where we make the entire quantity negative so we can minimize the expected costs (negative expected rewards) with gradient descent. In fact, Equation 2.26 is the *best* possible choice of baseline [58], as subtracting the expected return minimizes the variance in the gradients. Therefore, getting a good approximation of $V^{\pi_\theta}(s)$ is fundamental to reducing variance.

Unfortunately, we cannot calculate $V^{\pi_\theta}(s)$ analytically in practice, so we estimate it using the critic from the additional node on the top layer of our neural network, or create an auxiliary neural network π_ϕ explicitly to estimate the expected return. This critic provides an estimate $\hat{V}^\pi_\phi$ of the state value function such that $\hat{V}^\pi_\phi(s) \approx V^\pi(s)$, and this error is minimized through gradient descent with mean squared error loss (MSE). In this case, we have an auxiliary critic loss function

$$\mathcal{L}(\phi) = \mathop{\mathbb{E}}_{s_t,R_t\sim\pi_\theta}\left[\left(\hat{V}^\pi_\phi(s_t) - R_t\right)^2\right], \tag{2.31}$$

that we can simply add to our actor loss from Equation 2.30 which we are also trying to minimize, if the networks predicting π_θ and $\hat{V}^\pi$ share parameters as in Figure 2.3. Since we are only estimating $V(s_t)$, the advantage function $A^{\pi_\theta}(s_t,a_t)$ that minimizes variance is also an estimation, meaning variance is still reduced at the expense of introducing some bias into the model.

Algorithm 5: Training with Actor-Critic

input : a parametrized policy $\pi_\theta(a_t|s_t)$
Initialize parameters θ and state $s_0 \sim \rho_0$;
repeat
  sample a trajectory τ from $\pi_\theta(a_t|s_t)$;
  fit $\hat{V}^{\pi_\theta}$ to rewards through MSE;
  $Q^{\pi_\theta}(s_t,a_t) = \sum_k \gamma^k r_{t+k}$;
  $\hat{A}^{\pi_\theta}(s_t,a_t) = Q^{\pi_\theta}(s_t,a_t) - \hat{V}^{\pi_\theta}(s_t)$;
  $\nabla_\theta J(\pi_\theta) = -\sum_t \nabla_\theta \ln \pi_\theta(a_t|s_t)\hat{A}^{\pi_\theta}(s_t,a_t)$;
  $\theta := \theta - \eta\nabla_\theta J(\pi_\theta)|_\theta$
until *training is complete*;

2.4 Proximal Policy Optimization

Although Actor-Critic reduces the large variance of Policy Gradient, it is still subject to hyperparameter optimization on things like the learning rate η and discount factor γ. This optimization process is meant to speed up training time, finding the hyperparameter set that can train the quickest and most efficiently. Unfortunately, this can be a time-consuming and computationally expensive process, ironically slowing down the same training it is meant to speed up. Based on the ideas from Trust Region Policy Optimization [59], Proximal Policy Optimization (PPO) seeks to achieve the same goal, while being easier to implement and producing better results. Intuitively, we want to take the largest step possible in the right direction, implying a large learning rate. However, in addition to large learning rates being unstable or converging to suboptimal solutions, the loss function $J(\pi_\theta)$ does not estimate our performance metric that we wish to maximize (expected rewards) even in expectation. Rather, given the current parameters and actions sampled from the current parameters, the loss function will move the parameters θ in the direction that locally improves performance, but offers no guarantees that gradient descent will continue to improve performance from the updated parameters. PPO addresses this by taking the largest step it can while restricting policy updates to a stable region around the current policy.

Much like Actor-Critic, PPO contains both an actor and a critic. The actor is identical to the Policy Gradient neural network in Figure 2.2, deciding actions and the critic is a neural network seen in Figure 2.4 judging the quality of those actions. As such, it inputs both the state s_t and sampled action a_t, and outputs the estimation of the value function $Q_\phi(s_t, a_t)$. The critic's loss is identical to actor-critic, minimizing mean-squared error between the predicted and calculated normalized discounted rewards as in Equation 2.31, however the actor's loss is entirely different.

We start by calculating a ratio between current and previous policy

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, \tag{2.32}$$

such that $r_t(\theta_{\text{old}}) = 1$. Since we are evaluating two different policies on the same actions and states, $r_t(\theta) > 1$ if the actions taken are more likely under the new policy than the old one. This lets us define a surrogate objective function

$$\hat{\mathcal{L}}(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[r_t(\theta)A^{\pi_\theta}\right]. \tag{2.33}$$

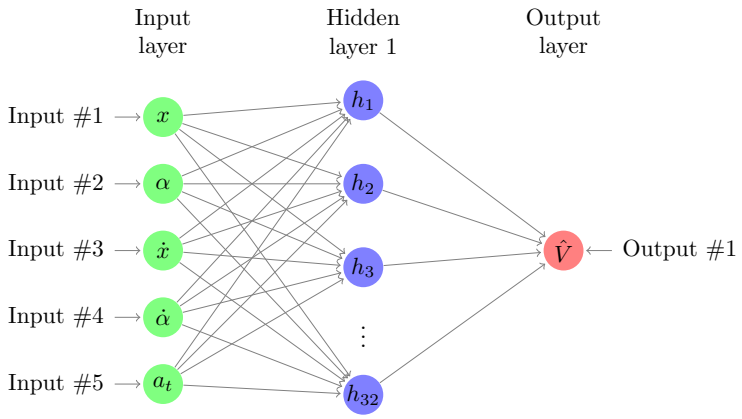


Figure 2.4 Neural network approximation of $\hat{Q}_\phi(s_t, a_t)$, representing Critic network only.

This is an equivalent metric to our actor loss, encouraging actions that perform better than the expected value, with the caveat that maximizing the surrogate objective will lead to excessively large policy updates if $r_t(\theta)$ is large. To remedy this, we introduce a parameter ε that tells us how far from the current policy we are able to step.

Then the final surrogate objective function is

$$\mathcal{L}(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta} \left[\min\left(r_t(\theta)A^{\pi_\theta}, \text{clip}\left(r_t(\theta), 1-\varepsilon, 1+\varepsilon\right)A^{\pi_\theta}\right)\right]. \tag{2.34}$$

This has the property such that if $A^{\pi_\theta}(s_t, a_t) > 0$, that is if the action performed better than the critic expected it to, the objective will increase with $\pi_\theta(a_t|s_t)$ only until $\pi_\theta(a_t|s_t) > (1+\varepsilon)\pi_{\theta_{\text{old}}}(a_t|s_t)$, at which point the objective will update no further than $(1+\varepsilon)A^{\pi_\theta}(s_t, a_t)$.

Similarly, if $A^{\pi_\theta}(s_t, a_t) < 0$ and the action performed worse than the critic expected it to, the objective will increase as $\pi_\theta(a_t|s_t)$ decreases only until $\pi_\theta(a_t|s_t) < (1-\varepsilon)\pi_{\theta_{\text{old}}}(a_t|s_t)$, at which point the objective will update no further than $(1-\varepsilon)A^{\pi_\theta}(s_t, a_t)$. In both of these cases, this ensures that the new policy remains reasonably close to the old policy, instead of performing dangerously worse. We used $\varepsilon = 0.2$ based on the suggested value in [60].

2.5 Improving Vanilla Policies

There are a number of techniques we can use in order to potentially improve training speed and robustness, however most of them are at odds with each other. For example, changing some of the hardcoded constants in Algorithm 3 can drastically change the

Algorithm 6: Training with Proximal Policy Optimization

input : a parametrized policy $\pi_\theta(a_t|s_t)$ and critic $Q_\phi(s_t, a_t)$
Initialize parameters θ, ϕ, and state $s_0 \sim \rho_0$;
repeat
  sample a trajectory τ from $\pi_\theta(a_t|s_t)$;
  fit $\hat{Q}_\phi$ to sampled rewards $R_t = \sum_k \gamma^k r_{t+k}$;
  $\hat{A}^{\pi_\theta}(s_t, a_t) = R_t - \hat{Q}_\phi(s_t, a_t)$;
  Optimize $\mathcal{L}(\theta)$ using $\theta := \arg\max_\theta \mathcal{L}(\theta)$ with Adam;
until *training is complete*;

performance.

While we tested some of these techniques, the various permutations between them result in an exploding number possible configurations to test. For the results reported in Chapter 4, we tried to standardize as much as possible, changing only one variable at a time. One of the things we test is how well the pendulum is able to respond to a disturbance, like getting tapped. Although the pendulums are not disturbed while training in the simulation they are initialized at an angle of $\alpha \in [-0.05, 0.05]$, which is up to 2.86° from vertical. This was default to the environment in Gym's Cartpole, based on the original environment defined in [5]. Increasing the initialization angles would force the model to learn how to overcome them; this would likely take many more trials, increasing training time. Training the pendulum until it can balance for *more* than 10 seconds will ensure the trained model is more robust, but this takes longer, even in a simulation. Similarly, adding noise to the measurements could help overcome sensor noise and model errors, again at the expense of training speed for large amounts of noise.

Actor-Critic and PPO share the same structure of training two neural networks, trying to choose optimal actions and judge those actions, respectively. However, Actor-Critic's actor and critic shared the bottom layers of their neural network in Figure 2.3, while PPO's critic was a separate network in Figure 2.4, that judged the value of states *and* actions. The idea was that the more information the critic network receives, the more accurate predictions it will be able to make. Once again, this slightly slows training, as there are more parameters that need to be updated during gradient descent. PPO also has a hyperparameter ε that did not seem to significantly affect performance.

The main hyperparameter value we augmented for Chapter 4 was the reward discount factor γ. Intuitively, this controls how much the agent values rewards now over rewards in the future, with $\gamma = 0.9$ giving the rewards a half-life of 0.13 seconds, $\gamma = 0.99$ a half-life of 1.38 seconds, and $\gamma = 0.999$ a half-life of 13.8 seconds. Based on the amount of foresight

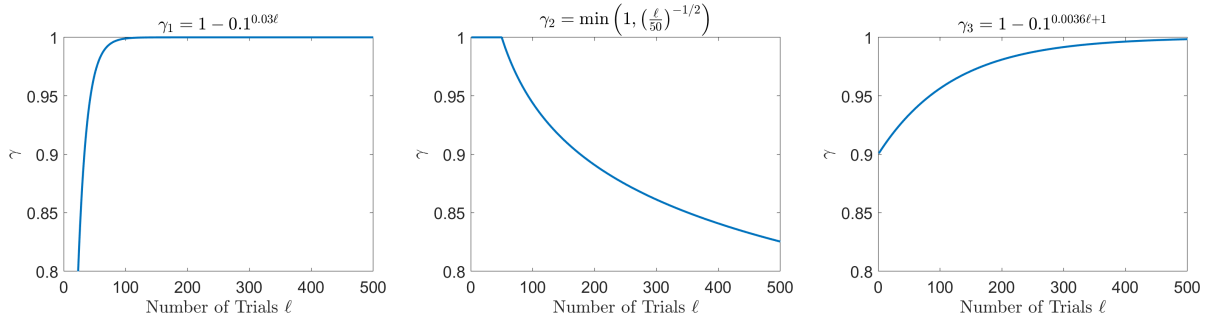


Figure 2.5 Attempted ways of annealing γ over time to improve performance.

each γ implies, we also experimented with increasing or decreasing γ as training progresses. A function like $\gamma = 1 - 0.1^{0.03\ell}$ smoothly increases from $\gamma \approx 0.822$ to $\gamma \approx 1$ as the length of each trial ℓ increases from 25 time steps (0.5 seconds) to 500 time steps (10 seconds). Intuitively, this should work well since attempts that fail early on need to encourage the agent to take more greedy actions to survive immediately, while longer-lasting trials should consider a longer time-horizon, ensuring the cart does not run off the track while trying to balance. However, this function gets very small for $\ell < 10$, corresponding to trials that fail almost immediately. On the other hand, $\gamma = \min\left(1, \left(\frac{\ell}{50}\right)^{-\frac{1}{12}}\right)$ ranges from $\gamma = 1$ for 50 or fewer time steps (≤ 1 second) and decreases to $\gamma \approx 0.825$ for 500 time steps. This function was designed so that after the discounted rewards are normalized, only the last 25 time steps of actions are discouraged. Intuitively, whether a trial lasts 0.5 seconds or 9.5 seconds, only the actions right before it fails directly lead to the crash, so should be the only ones strongly discouraged; 0.5 seconds was chosen arbitrarily. However, this neglects that the actions immediately preceding those lead to the pole being put in a state that it failed to recover from, so should probably be discouraged as well. In practice, neither annealing method improved performance for Policy Gradient, taking more than 45% longer to find a reaonable policy on average, so were not attempted for Actor-Critic or PPO. A third attempt, with $\gamma = 1 - 0.1^{0.0036\ell+1}$, which interpolates between $\gamma = 0.9$ for $\ell = 0$ and $\gamma = 0.9984$ for $\ell = 500$ was also attempted, but still failed to improve performance. All three alternative γ functions can be seen in Figure 2.5.

Each algorithm we tested has many modifications intended to improve performance, to the point where the line between different algorithms significantly blurs, as in Figure 2.1. For example, the REINFORCE algorithm with a baseline is just a lower variance version of Policy Gradient, but this is the vanilla Actor-Critic we used where we choose the baseline to be the state value function $V(s_t)$. There are several changes we can make to Actor-Critic, like using Temporal Difference methods during the update of both networks

[65]. PPO has several modifications to the loss function, like introducing a constraint to minimize the *KL-Divergence* between π_θ and $\pi_{\theta_{\text{old}}}$ to ensure the policy update does not deviate too far from the previous step. The original PPO paper [60] actually introduces several potential surrogate objective functions, including one with an *entropy* parameter.

Entropy is a measure of uncertainty in a probability distribution P, defined as $H(P) = \mathbb{E}[\ln P(x)]$. It is often considered beneficial to maximize entropy, based on the idea that sampled policies that generate similar rewards should be equally probable. By adding an entropy bonus (with yet another hyperparameter to tune) to the loss function of an algorithm, we can encourage a neural network to keep a wide distribution, encouraging exploration of the state space, and preventing premature convergence to suboptimal policies, which effectively halt learning. RL algorithms with entropy are sometimes referred to as *soft* algorithms [2]. Although soft algorithms have seen a lot of success in imporoving the efficiency of the Policy Gradient based algorithms, maximizing entropy also increases the variance in policies (something we spent a lot of time reducing), which can increase training time. Once again, a variety of entropy coefficients failed to improve performance for Policy Gradient, so were ignored for Actor-Critic and PPO.

Another change in performance comes from using an *experience buffer* to store experiences in the form of states, actions, rewards, and transitions. Unlike on-policy algorithms that use these samples once for a policy update then ignore them, off-policy algorithms can replay these experiences, using them for future policy updates as well. This is one of the things PILCO does by default. In fact, as we will see in Chapter 3, one of the reasons PILCO is so data-efficient is that it combines several techniques like adding noise, modeling uncertainty, and replaying experiences in order to outperform other state-of-the-art RL algorithms by an order of magnitude.

Chapter 3

PROBABILISTIC INFERENCE FOR LEARNING CONTROL

In Deisenroth's girthy 2010 PhD thesis [17], he introduced PILCO: Probabilistic Inference for Learning Control, a probabilistic dynamics model for efficient reinforcement learning on continuous action and state spaces. Unlike the model-free approaches in Chapter 2, this algorithm is *model-based*, using Gaussian processes (GPs) along with remarkably little data to learn a probability distribution over plausible dynamics models. By averaging over these models and taking uncertainty into account, PILCO is able to drastically reduce model bias, and achieve state-of-the-art data-efficiency [16], only being surpassed in recent years by modifications of the same algorithm.

In the decade since, a number of extensions to PILCO have been introduced, extending the ideas of PILCO for more computational efficiency, incorporating constraints, and active exploration of the state space. Some of these implementations include Deep PILCO [21], SafePILCO [51], and AEPILCO [69].

For a transition $f : S \times A \to S$ from an unknown dynamics system

$$s_{t+1} = f(s_t, a_t), \tag{3.1}$$

PILCO uses a squared exponential kernel to define a Gaussian posterior predictive distribution. By mapping uncertain test inputs through the Gaussian process dynamics model, it produces a Gaussian approximation of the desired posterior distribution, which can be used to analytically compute the gradients to minimize the expected return

$$J(\pi_\theta) = \sum_{t=0}^{T} \mathop{\mathbb{E}}_{s_t} \left[c(s_t) \right], \quad s_0 \sim \mathcal{N}(\mu_0, \Sigma_0), \tag{3.2}$$

where the initial state s_0 is sampled from a normal distribution. Instead of using gradient descent (although it can), vanilla PILCO minimizes this cost function (negative rewards) through conjugate gradient or (more commonly) limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm (BFGS).

3.0 Gaussian Processes

Given inputs $x_i \in \mathbb{R}^D$ and noisy observations $y_i = h(x_i) + \varepsilon_i \in \mathbb{R}$, where ε_i is (assumed Gaussian) noise ($\varepsilon_i \sim \mathcal{N}(0, \sigma_\varepsilon^2)$), the act of estimating h is known as *regression*. If h is a deterministic function defined by a finite number of parameters θ, regression aims to find the parameter set θ^* such that $h(\theta^*, x_i)$ best explains the corresponding observations y_i, $i = 1, \ldots, N$. By using a Bayesian freamework, we can determine a probability distribution over θ that expresses uncertainty in our estimations. By using non-parametric models instead, we can learn the shape of h directly from data, along with a few assumptions about smoothness, which is usually more reasonable than the assumptions in a parametric model.

Definition 3.1. A *Gaussian process* $\{X_t\}_{t\in T}$ is a collection of random variables X_t, such that every finite subset $F \subset T$ of those random variables has a multivariate Gaussian (normal) distribution.

That is, $\{X_t\}_{t\in T}$ is a *stochastic process* where every finite linear combination $\sum_{t\in F} a_t X_t$ has a Guassian distribution (or is identically 0).

Given data $\mathcal{D}$ composed of inputs $x_i \in X$ and outputs y_i assumed to come from a function $y_i = h(x_i) + \varepsilon_i$ as above, a Gaussian process infers a posterior distribution $p(h|\mathcal{D})$ over h from the Gaussian process prior $p(h)$, the data $\mathcal{D}$, and assumptions on the smoothness of h [17]. This posterior distribution can be used to estimate function values $h(x_*)$ given new inputs $x_* \in \mathbb{R}^D$.

A multivariate Gaussian distribution can be fully specified by a mean vector μ and covariance matrix Σ, such that $X \sim \mathcal{N}(\mu, \Sigma)$ with

$$\Sigma = \operatorname{cov}[x_i, x_j] = \mathbb{E}\left[(x_i - \mu_i)(x_j - \mu_j)^\top\right], \tag{3.3}$$

where the diagonal elements represent the variance σ_i^2 of the i-th random variable, while the off-diagonal elements σ_{ij} describe the correlation between the i-th and j-th random variables. Similarly, a Gaussian process can be fully defined by a mean function $m_h(\cdot)$

and covariance function

$$\begin{aligned} k_h(x, x') &= \mathop{\mathbb{E}}_{h} \left[(h(x) - m_h(x)) \left(h(x) - m_h(x') \right) \right] \\ &= \mathop{\mathrm{cov}}_{h} \left[h(x), h(x') \right], \end{aligned} \tag{3.4}$$

with x, $x' \in \mathbb{R}^D$. This is also known as the *kernel* of h.

Definition 3.2. A function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is called a *kernel* on $\mathcal{X}$ if there exists a Hilbert space H and a map $\phi : \mathcal{X} \to H$ with

$$k(x, y) = \langle \phi(x), \phi(y) \rangle \tag{3.5}$$

for all $x, y \in \mathcal{X}$, where ϕ is called a *feature map* and H is a *feature space* of k.

If we assume that $m_h \equiv 0$ (we can always add it back later), the Gaussian process can be completely defined by the kernel. A number of functions can be chosen for the kernel, including linear, periodic, and squared exponential functions, which can be composed or linearly combined in order to select which type of functions from the space of all possible functions are more probable [24]. This choice is arbitrary and can be made based on *a priori* domain knowledge, increasing their ability to capture relevant trends in the data.

Definition 3.1 tells us that any finite set of function values $h(x_i)$, $i = 1, \ldots, N$ is jointly Gaussian distributed. Therefore, the distribution of $h(X)$ can be defined as

$$p(h(X)) = \mathcal{N} \left(m_h(X), k_h(X, X) \right), \tag{3.6}$$

where $k_h(X, X)$ is the full $N \times N$ covariance matrix of all function values. Importantly, this has grows as N^2, making it very memory intensive for a large number of samples N.

The process of finding a posterior distribution on h with Bayesian inference is known as *kriging*. The general steps are as follows:

1. Specify a *prior* on the unknown quantity h
2. Observe transition data $\mathcal{D}$
3. Compute a posterior distribution over h using evidence from the observations.

This lets us predict the value of h at a new point by combining the Gaussian prior with a Gaussian likelihood function for each of the observed values to compute a weighted average of the known values of the function in the neighborhood of that point.

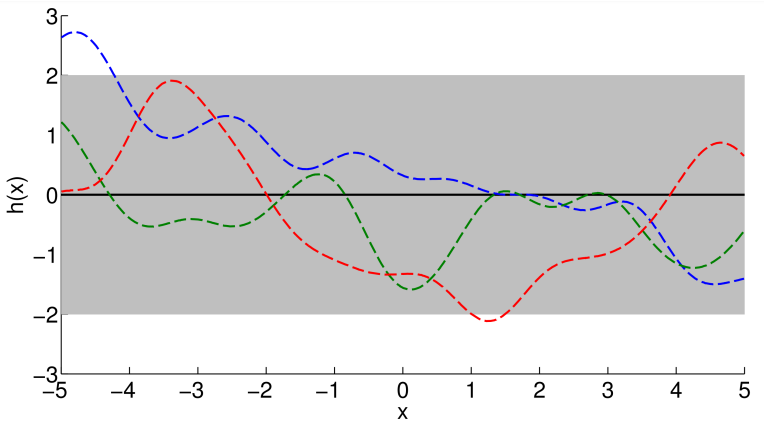


Figure 3.1 Samples from the GP prior along with the 95% confidence interval of the marginal distribution. Without any observations, the prior uncertainty about the underlying function is constant everywhere.

3.0.1 Priors

A Gaussian process defines a probability distribution over possible functions, with each sample from a multivariate Gaussian distribution representing one possible realization of function values. If we have not yet observed any training data, this is our prior distribution P_X, defined around $m_j \equiv 0$, as seen in Figure 3.1. This distribution has the same dimensionality as the number of test points $N = |X|$, with associated $N \times N$ covariance matrix.

Definition 3.3. A *radial basis function* (RBF) is a function $\psi : [0, \infty) \to \mathbb{R}$ with the property that

$$\psi(x) = \psi\left(\|x\|\right) \text{ or } \psi_c(x) = \psi\left(\|x - c\|\right). \tag{3.7}$$

Instead of using neural networks to approximate a function, vanilla PILCO uses an RBF called a squared exponential (SE) kernel

$$\begin{aligned} k_{SE}(x_p, x_q) &= \kappa^2 \exp\left(-\frac{1}{2}(x_p - x_q)^\top \Lambda^{-1}(x_p - x_q)\right) \\ &= \kappa^2 \exp\left(\sum_{i=1}^{D} -\frac{\|x_{p_i} - x_{q_i}\|^2}{2\ell_i^2}\right), \end{aligned} \tag{3.8}$$

with $x_p,\ x_q \in \mathbb{R}^D$, where κ^2 is the signal variance of h and $\Lambda = \operatorname{diag}\left(\ell_1^2, \ldots, \ell_D^2\right)$ is a diagonal matrix of the characteristic length scales ℓ_i, $i = 1, \ldots, D$, roughly describing the distance at which points become more correlated. Usually we add a noise covariance function $\delta_{pq}\sigma_\varepsilon^2$ such that $k_h = k_{SE} + \delta_{pq}\sigma_\varepsilon^2$, where δ_{pq} is the Kronecker delta and σ_ε^2 is the

noise variance. All of the hyperparameters of h can be optimised by maximum likelihood, when training the model.

3.0.2 Observations

Ultimately, the point of kriging is to find a model that describes the observed states well enough to make predictions on the optimal actions to choose, given new states. To this end, we need to ensure that the Gaussian process corresponding to our SE kernel is expressive enough to represent the function learned by our data with sufficient accuracy. To this end, it would be incredibly helpful if SE kernels were universal function approximators, the same way neural networks are (see Theorem 1.2).

Consider a function

$$h(x) = \sum_{i \in \mathbb{Z}} \lim_{N \to \infty} \sum_{n=1}^{N} \gamma_n \exp\left(-\frac{\left(x - (i + \frac{n}{N})\right)^2}{\lambda^2} \right), \tag{3.9}$$

with $x \in \mathbb{R}$, $\lambda \in \mathbb{R}^+$, variance $\frac{\lambda^2}{2}$, and normal weights $\gamma_n \sim \mathcal{N}(0,1)$, $n = 1, \ldots, N$. As $N \to \infty$, we can replace the sums with an integral over $\mathbb{R}$ as

$$\begin{aligned} h(x) &= \sum_{i \in \mathbb{Z}} \int_i^{i+1} \gamma(s) \exp\left(-\frac{(x-s)^2}{\lambda^2} \right) ds \\ &= \int_{-\infty}^{\infty} \gamma(s) \exp\left(-\frac{(x-s)^2}{\lambda^2} \right) ds, \end{aligned} \tag{3.10}$$

with $\gamma(s) \sim \mathcal{N}(0,1)$. This is Gaussian noise convolved with a Gaussian kernel, meaning h is a Gaussian process according to Definition 3.1.

Notably, the prior mean function, calculated by averaging h over γ is

$$\mathop{\mathbb{E}}_{\gamma}[h(x)] = 0, \tag{3.11}$$

because $\mathbb{E}[\gamma(s)] = 0$. Additionally, because the prior mean is 0, we can calculate the prior covariance function by completing the square, giving

$$\operatorname{cov}[h(x), h(x')] = \kappa^2 \exp\left(-\frac{(x-x')^2}{2\lambda^2} \right), \tag{3.12}$$

for suitable κ^2 [17].

Therefore, the prior mean function and the prior covariance function of $h(x)$ in Equation 3.9 correspond to the assumptions we made in Section 3.0.1: that we have a

prior mean function $m_h \equiv 0$ and prior covariance function given Equation 3.8. That is, the Gaussian prior implicitly assumes functions h that can be described by the function in Equation 3.9. All we have to prove is that h is a universal function approximator.

Definition 3.4. A continuous kernel k on a compact domain $\mathcal{X}$ is *universal* if the space of all functions induced by k is dense in $C(\mathcal{X})$, i.e. for every function $f \in C(\mathcal{X})$ and every $\varepsilon > 0$ there exists a function g induced by k with

$$\|f - g\|_\infty \leq \varepsilon. \tag{3.13}$$

Theorem 3.5 (Universal Power Series Kernels [64])
A kernel $k(x, x') = k(\langle x, x' \rangle)$ defined on $\mathbb{R}^n$ with power series expansion

$$k(r) = \sum_{i=0}^{\infty} a_i r^i \tag{3.14}$$

is a universal kernal if and only if $a_i > 0$ for all i.

The proof can be found in [64], which relies on the span of an injective feature map being an algebra.

Lemma 3.6
Let k be a universal kernel on $\mathcal{X}$. Then the normalization

$$k^*(x, x') = \frac{k(x, x')}{\sqrt{k(x, x)k(x', x')}} \tag{3.15}$$

also defines a universal kernel on $\mathcal{X}$.

Proof. Let $\phi : \mathcal{X} \to H$ be a feature map of k and $\alpha(x) = k(x, x)^{-1/2}$ so that $\alpha\phi : \mathcal{X} \to H$ is a feature map of k and k is a kernel.

Let $f \in C(\mathcal{X})$ and $\varepsilon > 0$. For $a = \|\alpha\|_\infty$ we get an induced function $g = \langle w, \phi(\,\cdot\,)\rangle$ with

$$\left\|\alpha^{-1} f - g\right\|_\infty \leq \frac{\varepsilon}{a}. \tag{3.16}$$

Then

$$\|f - \langle w, \alpha\phi(\,\cdot\,)\rangle\|_\infty \leq \|\alpha\|_\infty \left\|\alpha^{-1} f - g\right\|_\infty \leq \varepsilon. \tag{3.17}$$

By Definition 3.4, $k^*(x, x')$ is universal. □

Theorem 3.7 (Universal RBF Kernel [64])
The radial basis function (RBF) kernel

$$k(x, x') = exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right) \tag{3.18}$$

is universal.

Proof. First note that the kernel $k(x, x') = \exp(\gamma\langle x, x'\rangle)$ is universal by Theorem 3.5.

$$\begin{aligned}
\exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right) &= \exp\left(-\frac{\left(\|x\|^2 - 2\langle x, x'\rangle + \|x'\|^2\right)}{2\sigma^2}\right) \\
&= \frac{\exp\left(\frac{\langle x,x'\rangle}{\sigma^2}\right)}{\exp\left(\frac{\|x\|^2}{2\sigma^2}\right)\exp\left(\frac{\|x'\|^2}{2\sigma^2}\right)} \\
&= \frac{\exp\left(\frac{\langle x,x'\rangle}{\sigma^2}\right)}{\sqrt{\exp\left(\frac{\|x\|^2}{\sigma^2}\right)\exp\left(-\frac{\|x'\|^2}{\sigma^2}\right)}} \\
&= \frac{\exp(\gamma\langle x, x'\rangle)}{\sqrt{\exp\left(\gamma\|x\|^2\right)\exp\left(\gamma\|x'\|^2\right)}},
\end{aligned} \tag{3.19}$$

for $\gamma = \frac{1}{\sigma^2}$, meaning the RBF kernel $k(x, x')$ is also universal, by Lemma 3.6. □

An alternative proof can be found in [40]. Therefore, functions h of the form in Equation 3.9 that are described by an RBF kernel are universal function approximators on compact subsets of $\mathbb{R}^D$, approximating functions arbitrarily closely in the ∞-norm.

3.0.3 Posterior

Given a set of input vectors X and observed function values y with $y_i = h(x_i) + \varepsilon_i$, $i = 1, \ldots, N$, the posterior distrubition over h is given by

$$p(h|X, y, \theta) = \frac{p(y|h, X, \theta)p(h|\theta)}{p(y|X, \theta)}, \tag{3.20}$$

by Bayes' theorem.

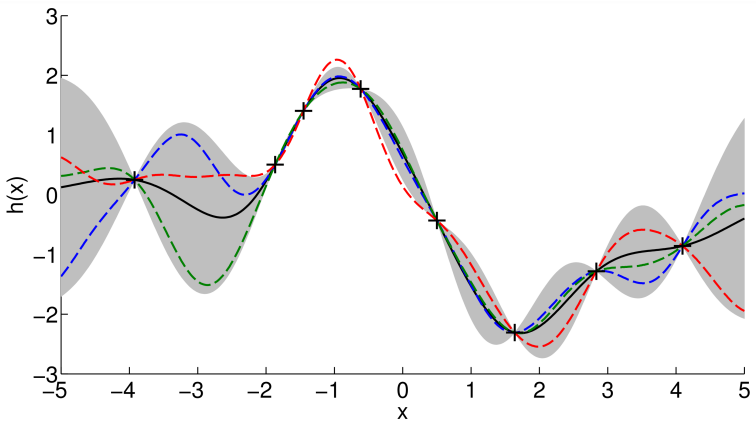


Figure 3.2 Samples from the GP posterior after having observed 8 function values (+'s). The posterior uncertainty depends on the location of the training inputs.

Then the likelihood of h is a finite probability distribution given by

$$\begin{aligned} p(y|h, X, \theta) &= \prod_{i=1}^{N} p(y_i|h(x_i), \theta) \\ &= \prod_{i=1}^{N} \mathcal{N}\left(y_i|h(x_i), \sigma_\varepsilon^2\right) \\ &= \mathcal{N}\left(y|h(X), \sigma_\varepsilon^2 I\right), \end{aligned} \tag{3.21}$$

where I is the identity matrix. This posterior Gaussian process has mean

$$\mathop{\mathbb{E}}_{h}\left[h(x)|X, y, \theta\right] = k_h(\tilde{x}, X)\left(k_h(X, X) + \sigma_\varepsilon^2 I\right)^{-1} y, \tag{3.22}$$

and covariance

$$\mathop{\mathrm{cov}}_{h}\left[h(x), h(x')|X, y, \theta\right] = k_h(x, x') - k_h(x, X)\left(k_h(X, X) + \sigma_\varepsilon^2 I\right)^{-1} k_h(X, x'), \tag{3.23}$$

where x, $x' \in \mathbb{R}^D$ are arbitrary vectors called *test inputs*. As seen in Figure 3.2, the GP posterior significantly reduces the prior uncertainty about h from the GP prior seen in Figure 3.1 with only a few samples.

Evaluating the marginal likelihood in the denominator of Equation 3.20 gives the normalizing constant

$$\int p(y|X, h, \theta) p(h|\theta)\, dh. \tag{3.24}$$

This integral ends up being analytically intractable, requiring numerical methods to

solve [66]. Instead of integrating out a hyper-prior $p(\theta)$ on the hyperparamters and approximating using Monte Carlo methods, PILCO instead finds a reasonable point-estimate $\hat{\theta}$ by maximizing the marginal likelihood in Equation 3.24 as

$$\hat{\theta} = \arg\max_{\theta} \log p(y|X, \theta). \tag{3.25}$$

By using evidence maximization, a compromise is made between data fit and model complexity that helps avoid overfitting.

3.0.4 Predictions

In order to use Gaussian processes to make predictions and learn, we need to compute the posterior predictive distribution of $h(x_*)$ for any test input x_*.

From Definition 3.1, the function values for training inputs X and test inputs X_* are jointly Gaussian with

$$p(h, h_*|X, X_*) = \mathcal{N}\left(\begin{bmatrix} m_h(X) \\ m_h(X_*) \end{bmatrix}, \begin{bmatrix} K & k_h(X, X_*) \\ k_h(X_*, X) & K_* \end{bmatrix}\right), \tag{3.26}$$

with $h = [h(x_1, \ldots, h(x_n)]^\top$ and K is the kernel matrix with $K_{ij} = k_h(x_i, x_j)$.

For test predictions $x_* \in \mathbb{R}^D$ and observed values $y_* \in \mathbb{R}$, the predictive marginal distribution $p\left(h(x_*)|\mathcal{D}, x_*\right)$ of the function value is Gaussian, from Equation 3.26. This has mean

$$\begin{aligned} m_h(x_*) &= \mathop{\mathbb{E}}_{h}\left[h(x_*)|X, y\right] \\ &= k_h(x_*, X)\left(K + \sigma_\varepsilon^2 I\right)^{-1} y \\ &= k_h(x_*, X)\beta, \end{aligned} \tag{3.27}$$

and covariance

$$\begin{aligned} \sigma_h^2(x_*) &= \mathop{\mathrm{var}}_{h}\left[h(x_*)|X, y\right] \\ &= k_h(x_*, x_*) - k_h(x_*, X)\left(K + \sigma_\varepsilon^2 I\right)^{-1} k_h(X, x_*), \end{aligned} \tag{3.28}$$

with $\beta = \left(K + \sigma_\varepsilon^2 I\right)^{-1} y$.

For multivariate predictions with $y_* \in \mathbb{R}^E$, the same process is used to train E independent Gaussian process models, with the same training inputs X, but separate training targets $y^a = [y_1^a, \ldots, y_N^a]$, $a = 1, \ldots, E$. Then the predictive distribution of $h(x_*)$

is a multivariate Gaussian with mean

$$\mu_* = \begin{bmatrix} m_{h_1}(x_*) & \dots & m_{h_E}(x_*) \end{bmatrix}, \tag{3.29}$$

and covariance

$$\Sigma_* = \operatorname{diag}\left(\begin{bmatrix} \sigma_{h_1}^2 & \dots & \sigma_{h_E}^2 \end{bmatrix} \right), \tag{3.30}$$

from Equation 3.27 and 3.28, respectively.

While testing, instead of sampling from a distribution, we can use a deterministic RBF network policy

$$\begin{aligned} \pi(x, \theta) &= \sum_{i=1}^{N} \beta_i \phi_i(x), \\ \phi_i(x) &= \exp\left(-\frac{1}{2}(x - \mu_i)^T \Lambda^{-1} (x_i - \mu_i) \right), \end{aligned} \tag{3.31}$$

which is equivalent to the mean of a Gaussian process, as $\phi_i(x)$ is just our SE kernel from Equation 3.8. Here, N is the number of basis functions of the RBF network. If the policy receives D dimensional inputs and implements an E-dimensional control signal, we need to optimize $N(D + E) + (D + 2)E$ parameters.

3.0.5 Rewards

Unlike the normalized discounted rewards in Equation 2.6 used for the algorithms in Chapter 2, PILCO uses a bounded exponential cost function (negative rewards) that saturates at $c(x) = 1$ when x is not close to the target:

$$c(x) = 1 - \exp\left(-\frac{\|x - x_{\text{target}}\|^2}{\sigma_c^2} \right) \in [0, 1] \tag{3.32}$$

where σ_c controls the width of c. In [16], learning was robust to the size of σ_c, with the model still able to learn how to balance an inverted pendulum with $\sigma_c = 10^{-6}$, although there is no guarantee this type of cost will result in policy convergence. Unlike traditional quadratic costs this cost function merely encodes that a state is simply "far away" from the target state.

This is used in the value function

$$\begin{aligned} V^{\pi}(s_0) &= \mathop{\mathbb{E}}_{\tau}\left[\sum_{t=0}^{T} c(s_t)\right] \\ &= \sum_{t=0}^{T} \mathop{\mathbb{E}}_{s_t}\left[c(s_t)\right], \end{aligned} \tag{3.33}$$

where τ is the trajectory of states $[s_0, \ldots, s_T]$, with the goal of minimizing expected costs.

3.1 Implementing PILCO

In order to use PILCO to predict a function $h(x)$ or learn a control, we have to specify our training set X, testing set x_*, and targets y. Ultimately, we are trying to predict future states s_{t+1} given current inputs s_t and actions a_t.

We introduce new quantities $\Delta_t = s_{t+1} - s_t + \varepsilon$ as training targets, where $\varepsilon \sim \mathcal{N}(0, \Sigma_\varepsilon)$, $\Sigma_\varepsilon = \operatorname{diag}(\sigma_{\varepsilon_1}, \ldots, \sigma_{\varepsilon_D})$. This is the function h that we will be estimating through the Gaussian process regression outlined in Section 3.0. We consider an unknown dynamics system

$$s_{t+1} = f(s_t, a_t), \tag{3.34}$$

with continuous states $s \in \mathbb{R}^D$, actions $a \in \mathbb{R}$, and tuples $(s_t, a_t) \in R^{D+1}$ as training inputs. To obtain the state distributions $p(s_1)$ through $p(s_T)$, we make a series of predictions one time step at a time, mapping uncertain test inputs through the GP dynamics model.

3.1.1 Dynamics Model Learning

To predict a future state s_{t+1} from $p(s_t)$, we require a joint distribution $p(s_t, a_t)$. We do this by first analytically computing the mean μ_a and covariance Σ_a of the predictive control distribution $p(a_t)$ by integrating out the state. Next, compute the cross-covariance $\operatorname{cov}[s_t, a_t]$ and approximate the joint state-control distribution $p(x_t) := p(s_t, a_t) = \mathcal{N}(x_t | \mu_t, \Sigma_t)$ by a Gaussian with matching moments. Although $p(a_t)$ and $p(s_t, a_t)$ are not necessarily Gaussian or even unimodal, we assume a joint Gaussian distribution as in the bottom right of Figure 3.3. As bimodal distributions can lead to the policy prematurely converging to an unfavourable minimum, approximating the control with a Gaussian is more robust than a better fitting variational distribution that minimizes the KL-divergence, for example. Such a distribution is likely to produce overconfident predictions that are not well suited for optimal control [7].

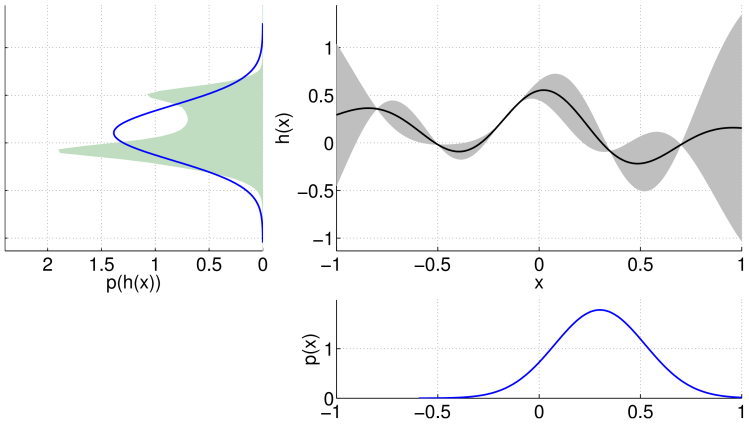


Figure 3.3 GP prediction at an uncertain input. The input distribution $p(s_t, a_t)$ is assumed Gaussian (lower right panel). When propagating it through the GP model (upper right panel), we obtain the shaded distribution $p(\Delta_t)$, upper left panel. We approximate $p(\Delta_t)$ by a Gaussian with the exact mean and variance (upper left panel).

The Gaussian process yields predictions one time step in the future

$$\begin{aligned} p(s_{t+1}|s_t, a_t) &= \mathcal{N}\left(s_{t+1}|\mu_{t+1}, \Sigma_{t+1}\right), \\ \mu_{t+1} &= s_t + \mathop{\mathbb{E}}_{f}\left[\Delta_t\right], \\ \Sigma_t &= \mathop{\mathrm{var}}_{f}\left[\Delta_t\right]. \end{aligned} \tag{3.35}$$

To get to s_{t+1} we need to predict the distribution

$$p(\Delta_t) = \int p\left(f(x_t)|x_t\right) p(x_t)\, dx_t. \tag{3.36}$$

We integrate out the random variable $x_t = (s_t, a_t)$, where $p\left(f(x_t)|x_t\right)$ is obtained from the posterior GP distribution. As in Equation 3.24, this integral is analytically intractable, so we approximate $p(\Delta_t)$ using a Gaussian, as seen in the upper left of Figure 3.3. Since we know the mean and variance of the posterior GP, from Equation 3.22 and 3.23, we can ensure the moments match.

At this point, a Gaussian approximation to $p(x_{t+1})$ is just $\mathcal{N}\left(x_{t+1}|\mu_{t+1}, \Sigma_{t+1}\right)$ with

$$\begin{aligned} \mu_{t+1} &= \mu_t + \mu_\Delta \\ \Sigma_{t+1} &= \Sigma_t + \mathrm{cov}\left[s_t, \Delta_t\right] + \mathrm{cov}\left[\Delta_t, s_t\right], \end{aligned} \tag{3.37}$$

where $\mathrm{cov}\left[\Delta_t, x_t\right] = \mathrm{cov}\left[s_t, a_t\right] \Sigma_a^{-1} \mathrm{cov}\left[a_t, \Delta_t\right]$, which can often be computed analytically, as in Equation 3.27 and 3.28. However, each of these calculations involves inverting an

$N \times N$ matrix, which involves $\mathcal{O}(EN^3)$ computations per gradient step, for E different target dimensions. This is only practical for somewhat short rollouts, being unsuited for tasks requiring a large number of trials. Optimizing the dynamics model for just one trial of an inverted pendulum takes ~ 30 minutes on a CPU.

By recursively feeding the output state distribution with uncertainty as the input state distribution with uncertainty as the next time step, this allows the agent to consider long-term consequences of a particular parametrization, minimizing cumulative expected costs. Predictions at uncertain test inputs scale with $\mathcal{O}(E^2N^2D)$, where N is the number of trials, D is the dimensionality of the input, and E is the dimensionality of the output.

To this end, we calculate the analytic gradients of the value function

$$\frac{dV^{\pi_\theta}(s_0)}{d\theta} = \sum_{t=0}^{T} \frac{d}{d\theta} \mathop{\mathbb{E}}_{s_t} [c(s_t)] \,, \tag{3.38}$$

where

$$\begin{aligned} \frac{d}{d\theta} \mathop{\mathbb{E}}_{s_t} [c(s_t)] &= \left(\frac{d}{dp(s_t)} \mathop{\mathbb{E}}_{s_t} [c(s_t)] \right) \frac{dp(s_t)}{d\theta} \\ &= \left(\frac{d}{d\mu_t} \mathop{\mathbb{E}}_{s_t} [c(s_t)] \right) \frac{d\mu_t}{d\theta} + \left(\frac{d}{d\Sigma_t} \mathop{\mathbb{E}}_{s_t} [c(s_t)] \right) \frac{d\Sigma_t}{d\theta}, \end{aligned} \tag{3.39}$$

and μ_t and Σ_t are the moments of the state distribution $p(s_t) = \mathcal{N}(s_t|\mu_t, \Sigma_t)$. Through repeated application of the chain rule, these gradients can be determined analytically, allowing the value function to be minimized using conjugate gradients, or BFGS.

By default, PILCO uses a limited-memory version of BFGS: a Quasi-Newton method that iteratively computes an estimate of the inverse Hessian in order to minimize the value function $V(\theta)$ [36]. It is $\mathcal{O}(N^2)$, and implemented through SciPy.

Given a positive definite matrix M_0 like the identity I, the search direction is set to

$$\rho_k = M_k \nabla V(\theta_k), \tag{3.40}$$

using the gradients determined from backpropagation. Next, a line search is performed along ρ, giving the update for the parameters at step k

$$\theta_{k+1} = \theta_k - \eta_k \rho_k. \tag{3.41}$$

Finally, the Hessian is updated according to

$$M_{k+1} = M_k \left(1 + \frac{\psi^\top M \psi}{\delta^\top \psi}\right) \frac{\delta\delta^\top}{\delta^\top \psi} - \left(\frac{\delta\psi^\top M + M\psi\delta^T}{\delta^\top \psi}\right), \tag{3.42}$$

where $\psi = \nabla V(\theta_{k+1}) - \nabla V(\theta_k)$ and $\delta = \theta_{k+1} - \theta_k$.

The algorithm implemented in the penultimate step of Algorithm 7, detailing a high-level view of PILCO in full.

Algorithm 7: Training with PILCO. This is generally quite slow.

```
Initialize controller parameters θ ~ N(0, I);
Apply random control signals and record data;
repeat
    Learn probabilistic dynamics model with all data;
    Model-based policy search;
    repeat
        Approximate inference for policy evaluation;
        Gradient based policy improvement; get dJ^π(θ)/dθ;
    until convergence;
    Set π* = π(θ*);
    Apply π* to system (single trial) and record data
until task learned;
```

3.1.2 Testing

Once a Gaussian process dynamics model is successfully learned (e.g. in a simulation), it can be tested on a real plant, like our inverted pendulum. This is a relatively straightforward process compared to learning the GP dynamics model, because all of the optimization is done. The test points x_* are completely deterministic, defined as single point with no variance to worry about. Algorithm 8 describes the process in MATLAB.

Due to physical constraints on the controller, PILCO makes the somewhat baffling decision to *squash* the control to within the amplitude limits $[-a_{\max}, a_{\max}]$ by calculating

$$\pi(x_*) = a_{\max} \sin\left(\tilde{\pi}(x_*)\right), \tag{3.43}$$

with $\tilde{\pi}(x_*) = \beta_\pi^\top k_\pi(X_\pi, x_*)$, $x_* \in \mathbb{R}^D$. While other squashing functions were introduced later [18], they are still periodic, instead of something reasonable like the hyperbolic tangent

Algorithm 8: Testing an optimized dynamics model. This is run in real time.

input : GP means $X = x_i \in \mathbb{R}^D$, $i = (1, \ldots, N)$
input : Targets $y \in \mathbb{R}^N$
input : Lengthscales $\Lambda = \text{diag}(\ell_1^2, \ldots, \ell_D^2)$
input : SE kernel $K \in \mathbb{R}^{N \times N}$ with optional noise σ_ε^2
Calculate $\beta = (K + \sigma_\varepsilon^2 I)^{-1} y$;
repeat
  Collect the next state s_t;
  Calculate $k_h(x_i, s_t) = \exp\left(-\frac{1}{2}(x_i - s_t)^T \Lambda^{-1}(x_i - s_t)\right)$;
  Then $k_h(X, s_t) = [k_h(x_1, s_t), \ldots, k_h(x_n, s_t)]^\top$;
  Set $\pi^* = \beta^\top k_h(X, s_t)$;
  Perform action $\sin(\pi^*)$ to system;
until *task failed*;

function seen in Figure 1.2 or a saturation *clip* function used for PPO in Algorithm 6, both of which locally approximate the identity function at the origin, and are bounded in $[-1, 1]$. By using sine, the output actually *decreases* when the predicted control rises past $a_t = \pi/2$, producing unfavourable actions in the wrong direction.

Because we no longer have to optimize a controller after every trial, this is able to run in real time, without any computational or memory issues. There is no longer a limit on the number of steps N that can be computed, meaning individual tests can run for as long as required, or until the task fails.

3.1.3 Extensions

How does PILCO learn so well? Part of its success stems from the fact that it experiences compounding improvements from all of the tweaks suggested in Section 2.5. While each marginal improvement might only improve training by a comparatively small percentage, using all of them at once results in a drastic reduction of data needed to learn simple models. This can be seen in Table 4.1, as the best trial for the policy-based algorithms was usually considerably faster than the average result. Although this was due to policies being learned from "lucky" initializations, keeping an *experience buffer* of all previous states makes a lucky initialization at some point in training increasingly likely, which is then incorporated into the probabilistic model.

Given the extreme data-efficiency this algorithm is able to obtain, it is almost surprising that it has been improved in recent years. Unsurprisingly, each extension is a modification of the original algorithm designed to tackle a specific problem inherent in the system.

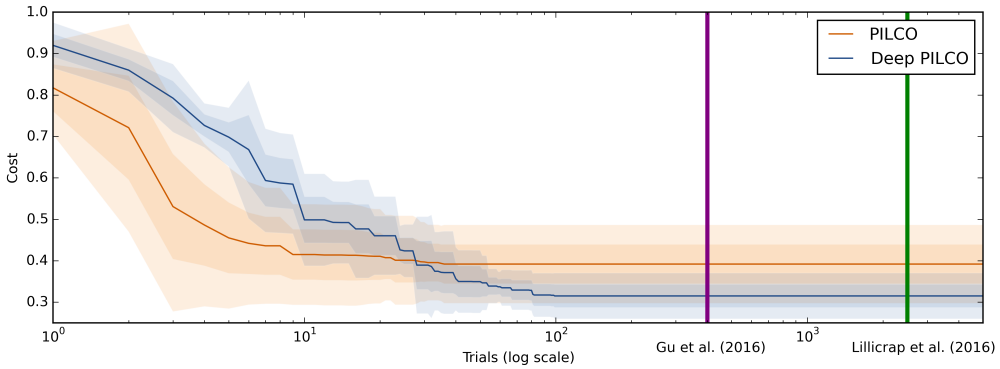


Figure 3.4 Cost per trial for PILCO and Deep PILCO for cartpole *swing-up* task [22].

3.1.3.1 Deep PILCO

The most obvious downside presented by PILCO is the massive computational demands required to optimize the controller, making it impractical for many data points or high-dimensional state spaces [67], due to the fact that its distribution propagation adds a squared term to the state space dimensionlity. As such, we were unable to replicate the same training steps from from Chapter 2, due to out-of-memory errors when training for 500 time steps.

This is addressed by Deep PILCO [22], which uses Bayesian deep dynamics models with approximate variational inference, by using a recurrent neural network (RNN) with *Dropout* as a surrogate for a variational Bayesian approximation. It uses the same Algorithm 7 as PILCO, but uses deep neural networks instead of Gaussian processes, which are capable of scaling to high-dimensional state spaces, scaling with $\mathcal{O}(ED)$. Although it takes approximately twice as many trials to learn a given task, the learned models have reduced cost and variance, when compared with vanilla PILCO, as seen in Figure 3.4.

3.1.3.2 SafePILCO

Further training speedups are found in SafePILCO [51], which extends the original algorithm with safety constraints embedded in the training procedure. By avoiding known dangerous states, this reduces the uncertainty in the distribution propagated through the Gaussian processes, leading to more confident predictions and earlier convergence. SafePILCO was able to balance a virtual inverted pendulum for 2 seconds (100 timesteps) after only 3 trials (with an additional 5 random pre-training rollouts, collecting data without optimizing the controller).

Additionally, SafePILCO was implemented entirely in Python, allowing it to take

advantage of OpenAI Gym environments and TensorFlow GPU access. Helpfully, the authors made their code available[1], included vanilla PILCO implemented in Python as well, which allowed us to take advantage of the GPU speedups and use our custom Gym environment defined in Chapter 1. This step alone resulted in a 2–3× speedup over vanilla PILCO implemented on a CPU in MATLAB for the same number of trials.

3.1.3.3 Active Exploration PILCO

Another thing PILCO fails to consider is the accuracy of the dynamic model it learns, instead only optimizing cumulative rewards. By using information entropy to describe samples, Active Exploration PILCO [69] is able to obtain informative policy parameters in the policy improvement state, increasing data-efficiency even further. It achieves this by incorporating an information entropy criterion $J^E(X) = \ln(|\Sigma(X)|)$ into the long-term sample prediction during the policy evaluation stage. Then the policy evaluation objective function becomes

$$J(\pi_\theta) = \sum_{t=0}^{T} \mathop{\mathbb{E}}_{x_t}[c(x_t)] + \alpha \sum_{t=0}^{T} \ln(|\Sigma(X)|). \tag{3.44}$$

By actively selecting samples in a way that maximizes entropy early on, AEPILCO is able to more closely match the predicted cost function to the real cost function, slightly reducing the number of trials required to swing up an inverted pendulum, with an even greater reduction found in problems with a higher dimensional state space. By using an exponentially decaying exploration parameter, AEPILCO was able to decrease the number of trials by up to 50% across a variety of tasks.

3.1.4 Hyperparameter Optimization

The training steps laid out in Algorithm 7 are incredibly data-efficient by default. However, there is an enormous amount of tweaking that can be performed in order to get the best performance out of the system without actually modifying PILCO like the algorithms in the previous section.

In addition to the number of radial basis functions defining the learned dynamics model (chosen to be 32, corresponding the number of neurons in the hidden layer used by the algorithms in Chapter 2), you can also change the number of steps a trial goes for. Google Colab's GPUs ran out of memory when running for the full 500 time steps. To get around this, instead of shortening the trial to decrease N, you can increase the subsampling parameter, which applies the same action for several time steps in a row,

[1] `https://github.com/nrontsis/PILCO`

without creating a new input point. This effectively reduces the time step size from $\frac{1}{50}$s to $\frac{1}{25}$s or even $\frac{1}{10}$s, with subsampling parameters of 2 and 5, respectively. The same 10 second rollout would only require $N = 100$ time steps, which is far more reasonable.

On the other hand, for tasks that have very short rollouts before failure (like balancing an inverted pendulum), you can run several trials before optimization, collecting data points without bothering to optimize the controller. You can also change the number of iterations the BFGS optimization runs for, and the number of times the optimization restarts, if it gets stuck in a local minimum.

One of the biggest differences between PILCO and the algorithms in Chapter 2 is that PILCO has a target state, which we defined as zero velocity, centred at the unstable equilibrium at the origin. Instead of a maximizing a rewards function encouraging the agent to take actions that lead to longer lifetimes, it minimizes a cost function quantifying the distance from the current state to the target state. Each component of the target state can be individually weighted to provide the best control, much the same way the parameters in LQR need to be tuned for optimal performance.

Importantly, PILCO automatically employs many of the techniques discussed in Section 2.5. In addition to keeping an experience buffer of all previously encountered states, it also has a noise parameter which can be fixed to specify a certain amount of signal noise during training and testing, which can help increase robustness to noisy sensors. However by default, signal variance, signal noise, and lengthscales are all optimised by maximum likelihood when training the model, in addition to the model hyperparameters, as in Equation 3.25.

This comprehensive customization, combined with the probabilistic dynamics modeling uncertainty allows PILCO to push the limits of data-efficiency to unprecedented levels, outperforming state-of-the-art methods by more than an order of magnitude [16]. While it is not appropriate for every learning task, restricted to short rollouts and low-dimensional environments, PILCO and its variants are promising algorithms that could leave slow, high-variance algorithms in the past.

Chapter 4

RESULTS

We trained each of the algorithms in Chapter 2 and Chapter 3 across several different hyperparameters. For the policy-based algorithms in Chapter 2, we changed the learning rate η and discount factor γ. Smaller values of η took longer to train, so we only report training curves for $\eta = 10^{-2}$, with $\gamma = \{0.9, 0.99, 0.999\}$. For PILCO, we changed the number of trials J before optimization on the controller began (*pre-optimization rollouts*) and the subsampling parameter S. Due to the large variance present in several of these configurations, we trained each combination ten times for a maximum of 1000 episodes each (10 for PILCO), or until it learned to balance in the simulation for 500 time steps (10 seconds). Due to memory issues, we only trained PILCO for 100 time steps ($2S$ seconds), but tested it for the full 500. Training was considered successful if the final model was able to keep the virtual pendulum upright ($|\alpha| < 12°$) without running off the track ($|x| < 0.4$m) for all 500 time steps.

4.1 Virtual Training

After training in a simulation, all the tested algorithms were able to balance a virtual inverted pendulum, though they took different amounts of time to learn their policies. Among policy-based algorithms, Actor-Critic performed slightly better than Policy Gradient across most discount factors, with $\gamma = 0.99$ resulting in the fastest learning on average, as well as the most consistent learning for both algorithms. PPO was able to achieve very fast learning times, but only infrequently. While half of the successful simulations learned to balance a pole, more than half of all attempts crashed Google Colab due to numerical instability. Specifically, due to the denominator of the surrogate objective in Equation 2.32, the ratio occasionally explodes past floating point precision, returning `inf` or `nan`. These unrecoverable attempts are not recorded in Table 4.1 or Figure 4.1.

Table 4.1 Training results for all three policy-based algorithms from Chapter 2 across different rewards discount factors γ. **Bold** numbers represent the least number of trials for each column, *italics* are the best for each algorithm.

Algorithm	γ	Success %	Average # Trials	Best # Trials
PG	0.9	60%	339	202
PG	0.99	**100%**	*245*	*80*
PG	0.999	80%	340	118
AC	0.9	50%	300	91
AC	0.99	**100%**	*250*	***19***
AC	0.999	90%	255	46
PPO	0.9	50%	***72***	32
PPO	0.99	50%	258	*23*
PPO	0.999	50%	135	38

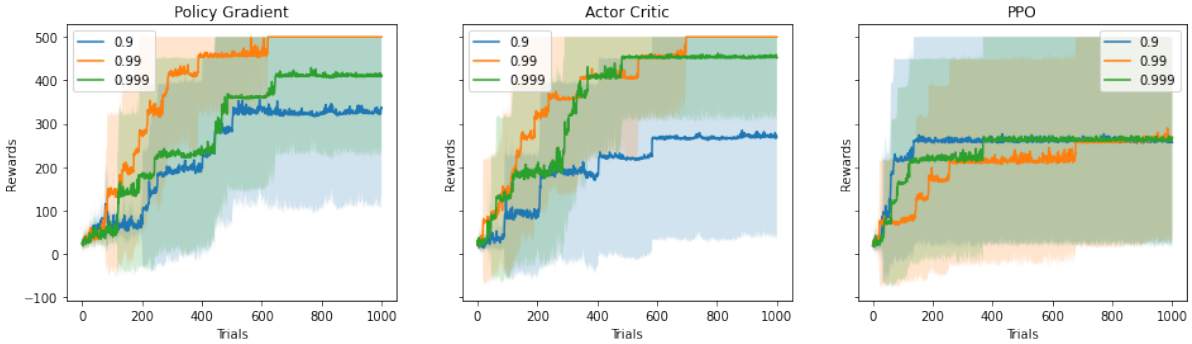


Figure 4.1 Policy Gradient, Actor Critic, and Proximal Policy Optimization for a variety of discount factors γ. The mean trial is graphed, along with ± 1 standard deviation, up to a maximum of 500 time steps.

However, when PPO does not crash, it trains among the quickest of all the algorithms tested, which is supported by results in the literature [60]. Among the trials that completed 1000 iterations without learning a sufficient policy or crashing, most appeared to get stuck in a local optimum where the gradients failed to improve policy performance. For very short rollouts, where the pendulum falls almost immediately, the loss function performs unintuitively, encouraging actions that directly lead to the pendulum falling (Policy Gradient) or discouraging *all* actions resulting in unstable performance (Actor-Critic and PPO). Dynamically changing the hyperparameters η, γ, and ε may help resolve these issues.

Similarly, PILCO also had surprisingly varied results depending on the hyperparameters chosen, as seen in Table 4.2 and Figure 4.2. When subsampling was not used, rolling out a trial for 100 time steps was very memory-intensive, regularly crashing Google Colab with

Table 4.2 Training results for PILCO across different numbers of pre-optimization rollouts J and subsampling S hyperparameters.

Algorithm	J	S	Success %	Average # Trials	Best # Trials
PILCO	0	1	10%	4	4
PILCO	0	2	20%	5	3
PILCO	2	1	70%	2.3	1
PILCO	2	2	80%	2.4	1
PILCO	4	1	**100%**	**1.6**	1
PILCO	4	2	70 %	2.7	1

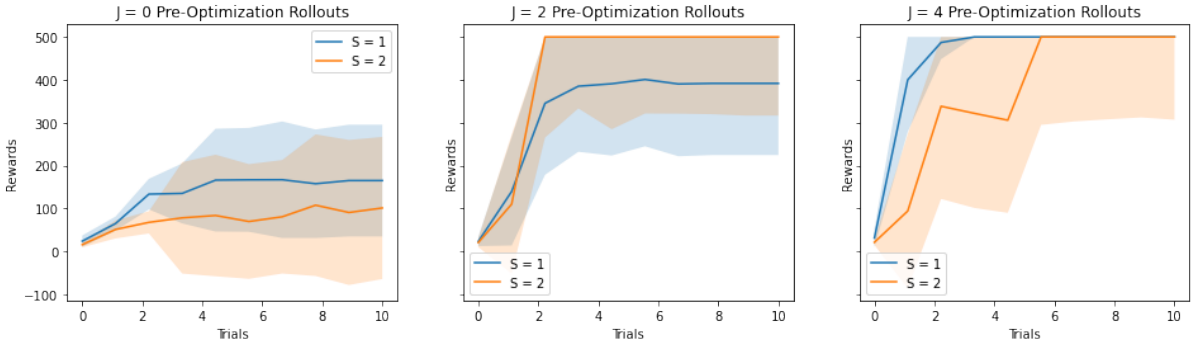


Figure 4.2 PILCO across different hyperparameters. S represents how many time steps a new data point is sampled; J is the number of trials before optimization starts. The mean trial is graphed, along with ± 1 standard deviation, up to a maximum of 500 time steps. Note: the x-axis only goes up to 10—not 1000 as in Figure 4.1.

out-of-memory errors before a policy could be learned. By increasing the subsampling parameter S to 2, the algorithm takes 2 time steps before observing transition dynamics. While this loses some granularity in the control, it doubles the length observations, because 100 samples lasts four seconds instead of two. On average, using subsampling resulted in slightly slower training, but a greater number of successful rollouts.

Because a limit of 100 time steps was chosen, with up to 10 rollouts, it was common for the program to crash between 100 and 500 time steps, before all 10 trials had finished. This could be drastically reduced by collecting transition data, performing J trials before attempting to optimize the dynamics model. These pre-optimization rollouts usually gave the optimizer "free" data that helped it learn the dynamics without wasting time and memory optimizing the model. In the best case scenario, PILCO was able to learn the a sufficient policy after a single optimization of the pre-optimization rollouts. Compared to the other algorithms, this is effectively learning after only a 1 trial (compared to a best-case of 19 and an average case of 72 for the other algorithms, as seen in Table 4.1).

4.1.1 Normalized Rewards

In each of the policy-based networks, we include the normalized discounted rewards-to-go, described in Equation 2.6. The discount factor γ prevents ensures the rewards are finite, and encourages the agent to seek immediate rewards instead of rewards over a longer time-horizon, depending on the value chosen. However, this can still result in some rewards that vary over a large range of magnitudes, especially in the case of large γ or a long trial. Unfortunately, neural networks are not invariant to the scale of their input, and normalizing input values can result in large improvements in performance on their own [36]. As a rule of thumb, if something can be normalized, it probably should be normalized. As a bonus, normalizing rewards causes the agent to encourage and *discourage* roughly half of the actions, instead of encouraging all actions but encouraging bad actions less.

For Actor-Critic, the critic network is trying to estimate the normalized discounted rewards—an estimate of the value function $V^{\pi_\theta}(s)$—given only the state at each time step. Even though it shares many of the parameters with the policy network defining the distribution each action is sampled from, finding a good estimation is essentially an impossible task. Given identical states the same amount of time before the pole falls and the trial ends, the normalized discounted rewards-to-go associated with that time step will differ depending on the length of the trial. The critic has no way of knowing how long any particular trial is, so has no reasonable way to accurately estimate the value function. The critic loss will necessarily be high, meaning less weight will be put on minimizing the actor loss.

By discounting but not normalizing the rewards, this apparently solves that issue entirely. After discounting only, identical states have identical rewards when they occur the same amount of time before a trial ends. Since the critic only has access to the state of the environment, it will predict the same value function given the same state, enabling it to learn the function much better. It turns out, this is a non-issue. The critic network is not trying to estimate the *state-action* value function $Q(s, a)$; that would make the gradients identically 0. Rather, it trying to estimate the *expected* return for a given state, which the normalized discounted reward for that action only provides a sample estimate of. Value estimates for the same states should be the same, regardless of when in a trial they occur. Instead of encouraging actions that lead to longer lifetimes like Policy Gradient, Actor-Critic encourages actions that perform better than expected, regardless of whether the action ends up being particularly good. Similarly, this discourages actions that are worse than expected, even if they do not cause the pole to fall. If the current action is worse than the average action for that state, take another action.

Table 4.3 Training results for unnormalized Actor-Critic with different γ.

Algorithm	γ	Success %	Average # Trials	Best # Trials
AC Unnormalized	0.9	30%	**548**	374
AC Unnormalized	0.99	**50%**	603	**345**
AC Unnormalized	0.999	30%	636	346

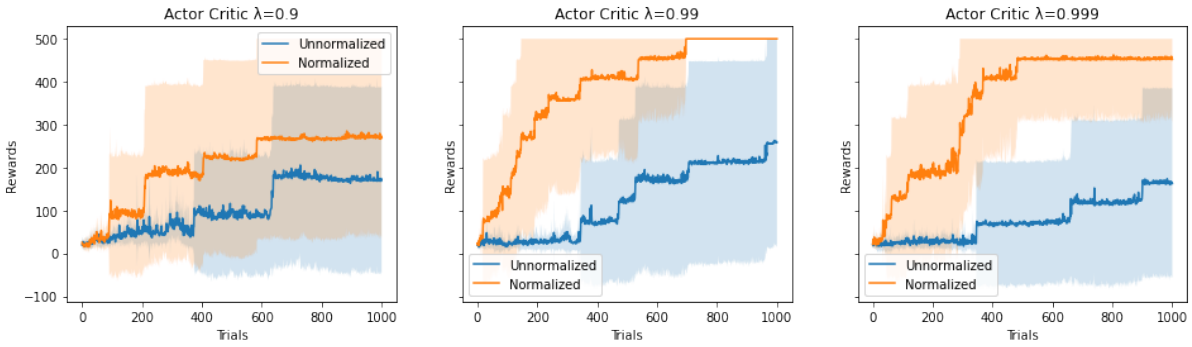


Figure 4.3 Discounted Rewards vs *Normalized* Discounted Rewards for Actor Critic.

As seen in Figure 4.3, the rule of thumb holds true: normalize everything. Despite not being able to exactly learn the value function, performance of the normalized discounted rewards was superior to unnormalized discounted rewards, especially for larger values of γ. Comparing Table 4.3 to Actor-Critic's rows in Table 4.1, we can see that the unnormalized rewards only learned to balance less than half as often, usually taking more than twice as long to get there. Large values of γ result in considerably larger discounted rewards towards the beginning of a long episode, to the point where even a small percentage error will result in a massive critic loss, which will give it far too much weight during optimization. Normalizing the discounted rewards keeps the values in a much smaller range, so the optimizer can focus primarily on updating the policy parameters instead.

4.1.2 Network Size

One thing that remained constant across all four algorithms was the size of the networks used. The policy-based algorithms in Chapter 2 were all represented by neural networks with a single hidden layer containing 32 neurons each. Due to the differences in the top layer and value functions for each network, Policy Gradient had 226 trainable parameters, Actor-Critic had 259 (value function), and PPO had 451 (226 for π_θ and 225 for $\hat{V}_\phi$). Similarly, PILCO's RBF network was composed of 32 squared exponential kernels, consisting of 166 trainable parameters. This network size was chosen completely arbitrarily, as this was enough to learn to balance the inverted pendulum with Policy Gradient early on. In

Table 4.4 Training results for Policy-Gradient with $\gamma = 0.99$ different numbers of neurons in the single hidden layer. *: 0 neurons represents a direct connection from inputs to outputs with no hidden layers.

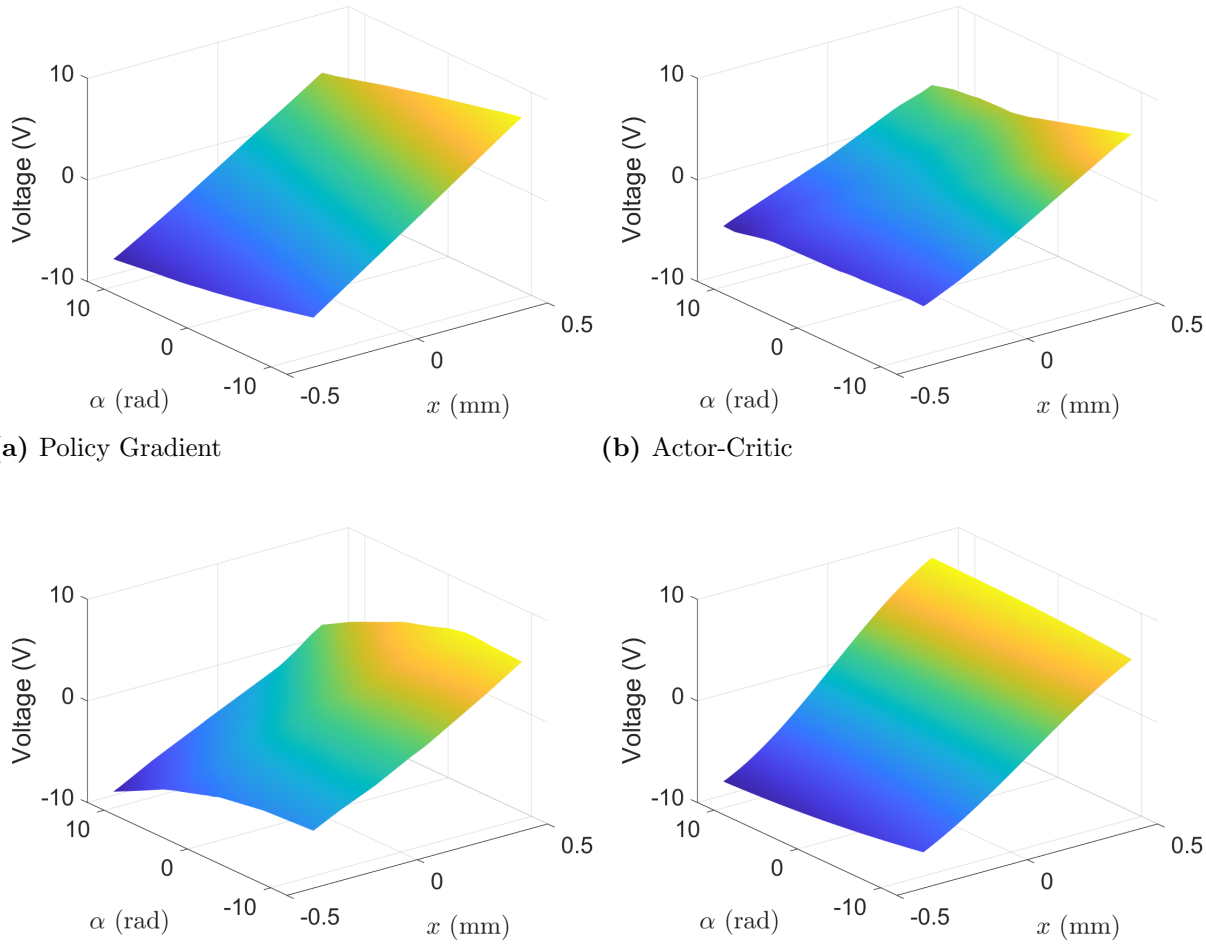
Neurons	Parameters	Success %	Average # Trials	Best # Trials
0*	10	40%	502	295
1	9	60%	556	210
2	16	40%	409	137
4	30	60%	314	128
8	58	60%	384	187
16	114	40%	533	358
32	226	**100%**	245	80
64	450	**100%**	**209**	**32**

Table 4.4, we can see the effect of running Policy Gradient with $\gamma = 0.99$ and the same configuration as the previously reported results, but with a different numbers of neurons in the hidden layer.

Somewhat surprisingly, all of the smaller networks were able to learn an acceptable policy about half of the time. A policy defined by a shallow network with only 1 hidden layer neuron and 9 trainable parameters was still able to balance the pendulum within 210 trials, and a policy with *no* hidden layers at all was still able to learn within 295 trials. For comparison, LQR is a linear controller—effectively a neural network with no hidden layers, no biases, a single output neuron with linear activation, and only 4 parameters—and it is still able to balance an inverted pendulum. That being said, the weights must still be tuned to achieve optimal performance when minimizing the LQR objective. On the other hand, larger networks were also able to consistently learn to balance the inverted pendulum, presumably taking advantage of nonlinearities in the system to learn faster.

If fact, if we plot a few surfaces in Figure 4.4, we can see that the learned policy is basically linear in x and α for each of the four algorithms.

A cross section in x and α can be seen in Figures 4.5 and 4.6, respectively. The control appears to be far more sensitive to the angle than position, especially in the case of PILCO, which explains why it moves around the environment so much, as we will see. For some reason, PPO only cares about positive values of x. For the most part, the sensitivity plots of $\dot{x}$ and $\dot{\alpha}$ look mostly the same, but with higher amplitudes.



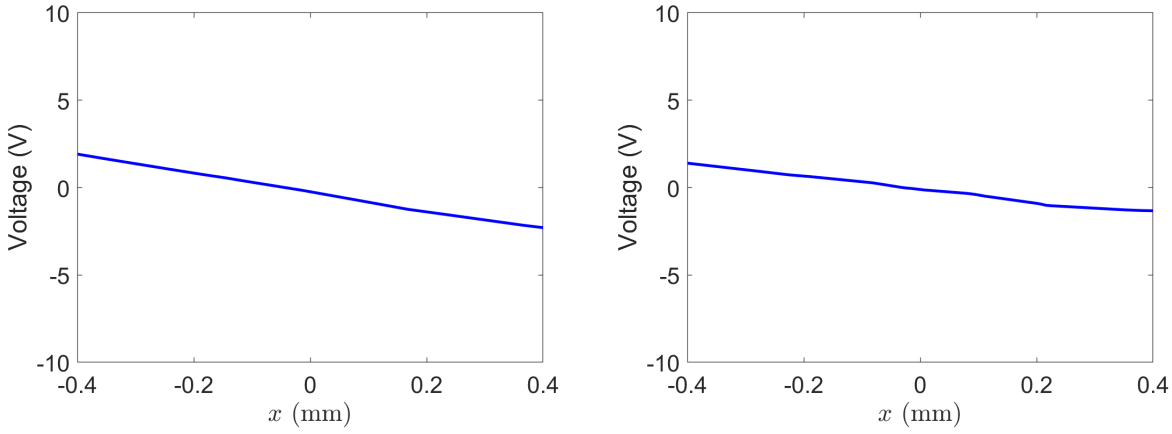
(a) Policy Gradient

(b) Actor-Critic

(c) Proximal Policy Optimization

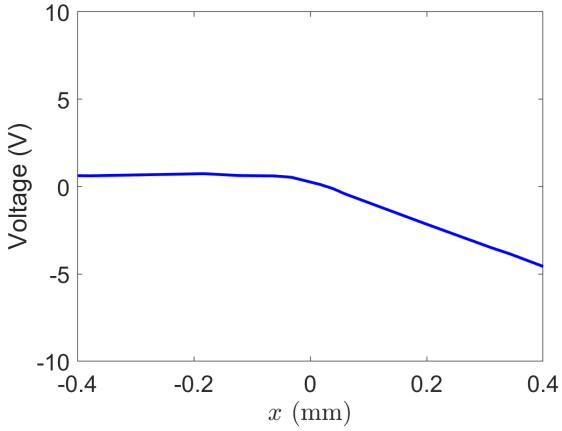
(d) PILCO

Figure 4.4 Surface plots showing the sensitivity of the controller to changes in x and α.

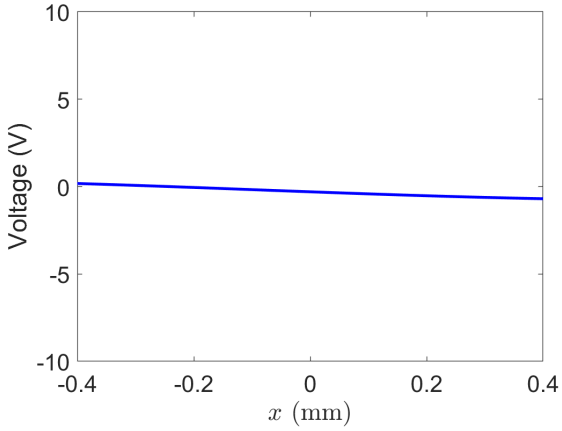


(a) Policy Gradient

(b) Actor-Critic

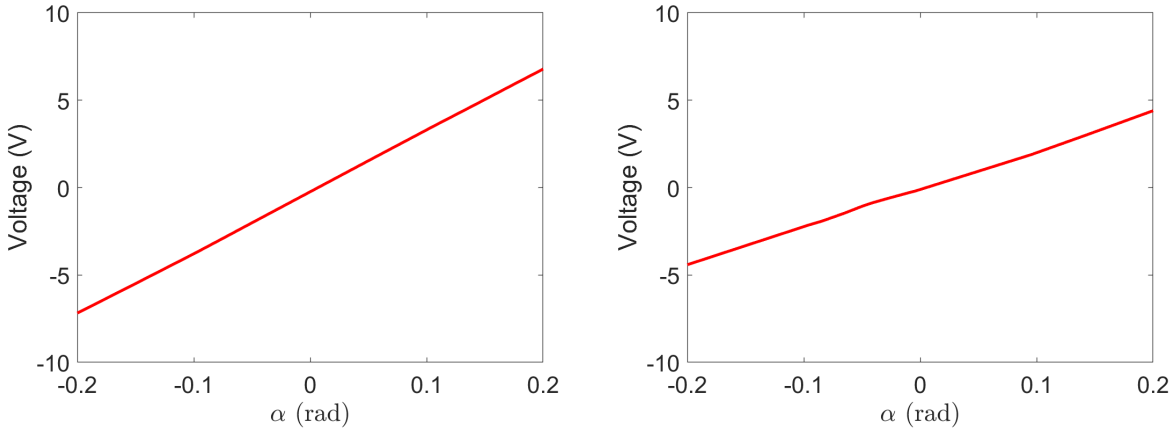


(c) Proximal Policy Optimization



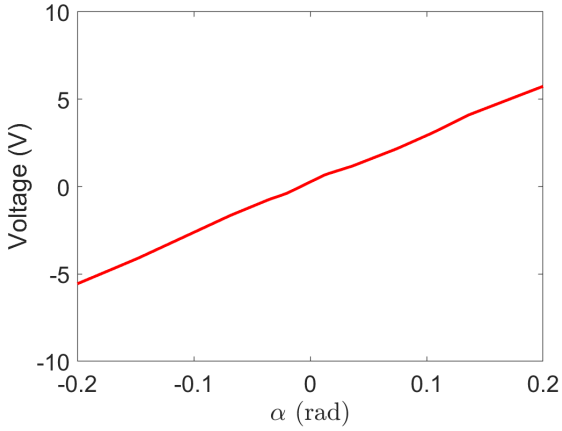
(d) PILCO

Figure 4.5 Cross section of Figure 4.4, showing each algorithm's sensitivity to changes in x.

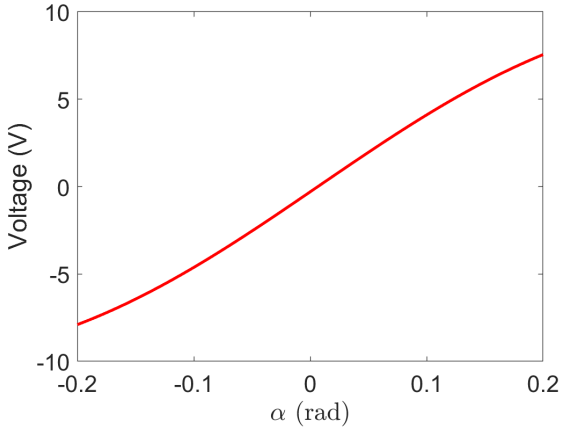


(a) Policy Gradient

(b) Actor-Critic



(c) Proximal Policy Optimization



(d) PILCO

Figure 4.6 Cross section of Figure 4.4, showing each algorithm's sensitivity to changes in α.

4.2 Real World Balancing

Once training is complete and a policy that can balance the pole for 10 seconds has been successfully learned, the neural network or radial basis function network can be downloaded as a series of matrices W_1, b_1, W_2, b_2 corresponding to the weights and biases of each layer of the neural network or X, y, K, Λ corresponding to the Gaussian process means, targets, squared exponential kernel, and lengthscales. These can be loaded into MATLAB, where the network is recreated in a single block in Simulink. While later versions of MATLAB have neural network support, the version we are using does not, meaning each network must be recreated from scratch, using Equation 1.2 or Algorithm 8. During testing, we ignore output neurons corresponding to the value functions and distribution variances, using only the predicted means of each distribution for the controller. This significantly reduces noise that was added to the model during training, which usually makes the real model more robust.

The following pages are going to contain a whirlwind of colours and graphs, which will be labeled and described in detail. In Figure 4.7, we can see the x-coordinate of the state space for each controller as implemented for 20 seconds on a real inverted pendulum described in Section 1.2. As you can see, the controller allows the cart to oscillate back and forth along the track, each algorithm with its own frequency and amplitude. This is unlike traditional controllers like LQR (Figure 4.7 (e)) that minimize the distance from the origin by keeping the cart nearly still at $x = 0$. This is due to the way each controller was trained. For the policy-based algorithms, we run a trial until the pendulum falls or the cart runs off of the track (Algorithm 3). Through hundreds of trials, the agent eventually learns not to let $|x| > 0.4$m; it continues to obtain rewards for any other values. Because it is not explicitly encouraged to keep the cart stationary, it does not. Decreasing the acceptable range of x-coordinates during would help keep the cart closer to the origin, at the expense of increased training time, since the cart would fail and reset more often. Notably, the oscillatory behaviour and biases to one side or another are usually reflected in the simulated trials as well, although the amplitude of the oscillations is usually a little larger in real life. Because the simulation matches the real agent so well, the actions it takes are essentially the same, given the same states as input.

One notable exception to this is PILCO. Unlike the policy-based algorithms, PILCO is given a target state $[0, 0, 0, 0]$, with the goal of minimizing the distance to the target, quantified as the cost function in Equation 3.32. As a result, trained PILCO models are usually very stable in a simulation, centred at the target state. However, this particular trained model lets the cart slowly wander across the track to the very edge; sometimes

it runs off the track completely. This can easily be seen in Figure 4.5, as changes in the x-coordinate have almost no effect on the controller, even at extreme values. A likely explanation for this is that due to the extreme data-efficiency of PILCO, it trained the model over the course of 2 or 3 trials—not two or three hundred. It probably never encountered an exit condition from letting $|x| > 0.4$m, so it never learned to avoid that.

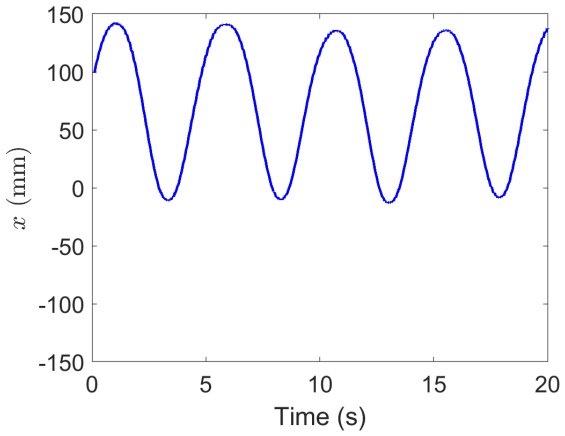
In Figure 4.8, we can see the angle of the pendulum during the same 20 second rollout on the real pendulum. Once again, we observe that controller lets the pendulum oscillate back and forth—sometimes up to 2° in each direction—as it never was explicitly encouraged *not* to do that. Although, PILCO lets the cart move across the track freely, it generally remains very close to upright as it does so, having the smallest average deviation from vertical. Once again, LQR blows every other controller out of the water, with the pendulum angle rarely rising above the quantized sensor noise. This raises an interesting point: while reinforcement learning algorithms are capable of learning from scratch, they generally learn *exactly* what they are rewarded for—in this case remaining relatively upright and on the finite length of track. If the goal is to minimize the cart movement, pendulum angle, or control effort, the desired behaviour has to be explicitly rewarded.

Figures 4.9 and 4.10 show the final two variables in the state space: $\dot{x}$ and $\dot{\alpha}$. These sometimes look like noisy versions of x and α because they were calculated numerically, through a digital filter.

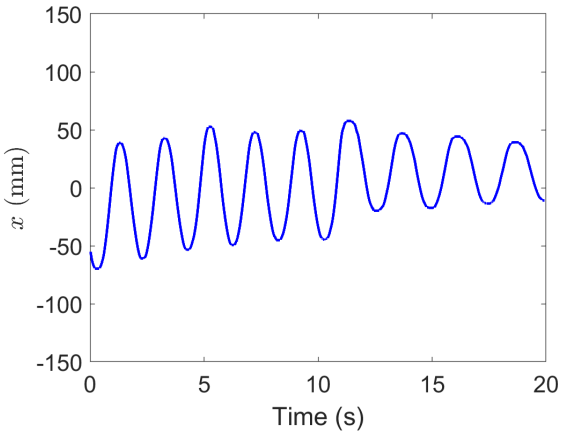
Finally, we have Figure 4.11, showing the voltage applied to the controller; these are the actions a_t that our networks output. Unsurprisingly, most of the controllers also exhibit oscillatory behaviour, as they are calculated in response to oscillatory states. With the notable exception of Policy Gradient, each controller is mostly able to keep the voltage under 2V. What is more interesting is the amount of noise each controller exhibits. Policy Gradient appears noisiest, which results in the highest voltages being sent to the motor. Unlike the other controllers, which appear fairly smooth to the naked eye, Policy Gradient is visibly slightly jerky, as it moves along the track. Perhaps a smoothing filter of some kind could help improve performance, as long as does not cause a delay in the control.

4.3 Forced Disturbances

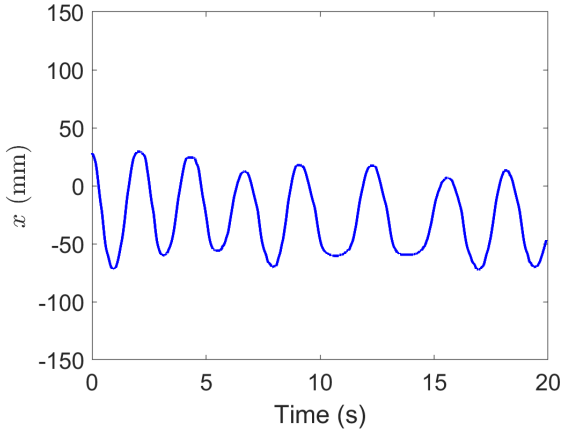
We have shown that each algorithm is able to balance a real inverted pendulum for at least 20 seconds, although they usually continue balancing much longer than that, if unperturbed. So what happens if we perturb the pendulum? In this section, we let the pendulum balance on its own, waiting for it to begin the steady-state oscillations seen in the previous section, then tap the pendulum gently. There was no way to standardize



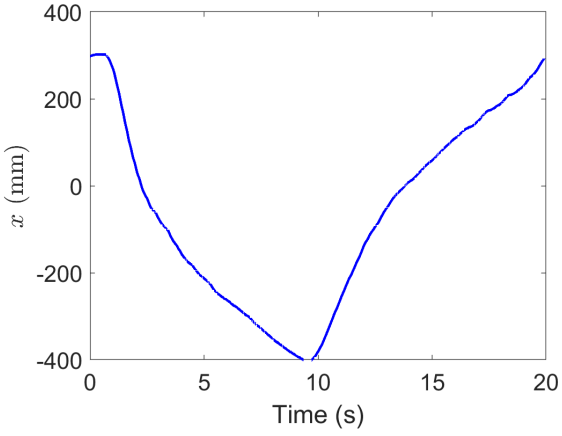
(a) Policy Gradient



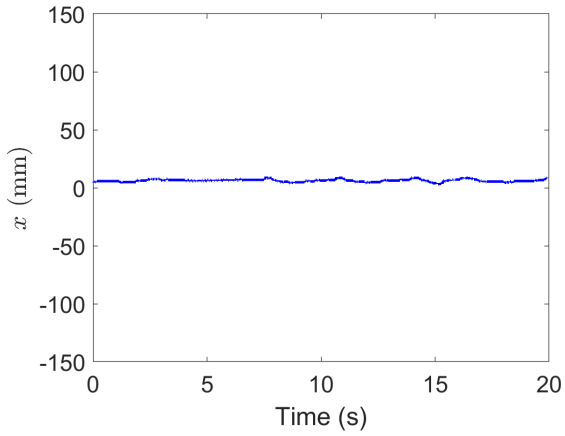
(b) Actor-Critic



(c) Proximal Policy Optimization

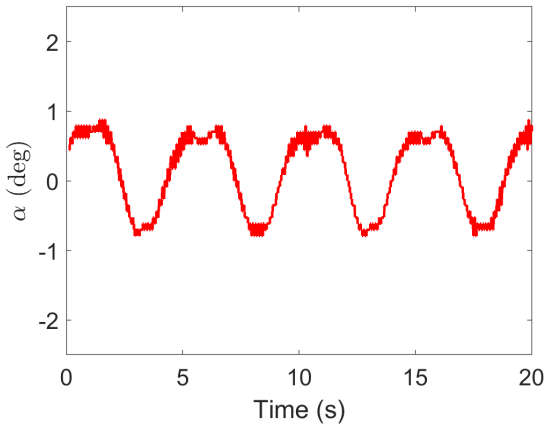


(d) PILCO

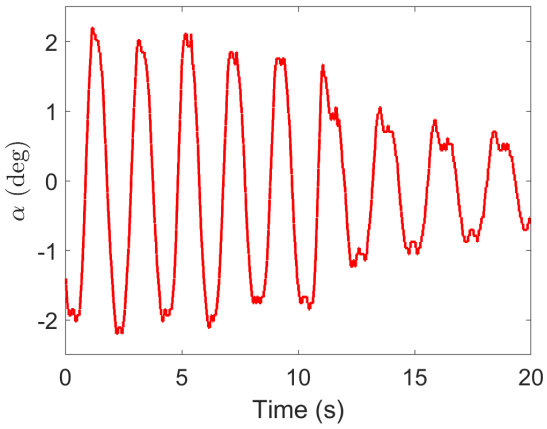


(e) Linear-Quadratic Regulator

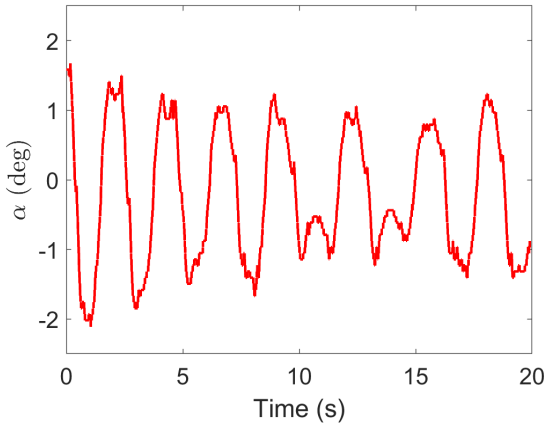
Figure 4.7 State response x-coordinate for each algorithm over 20 seconds, steady.



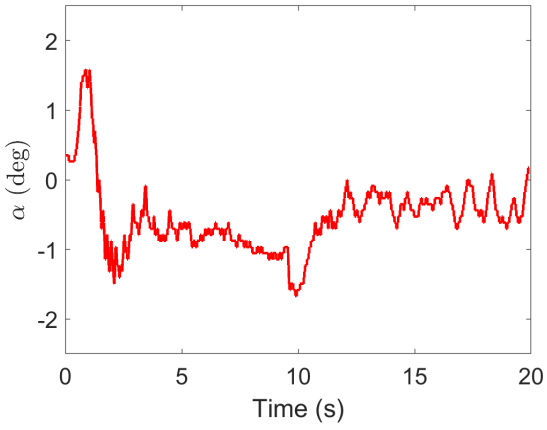
(a) Policy Gradient



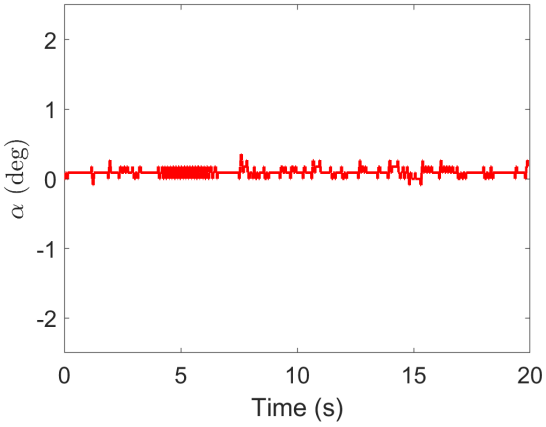
(b) Actor-Critic



(c) Proximal Policy Optimization

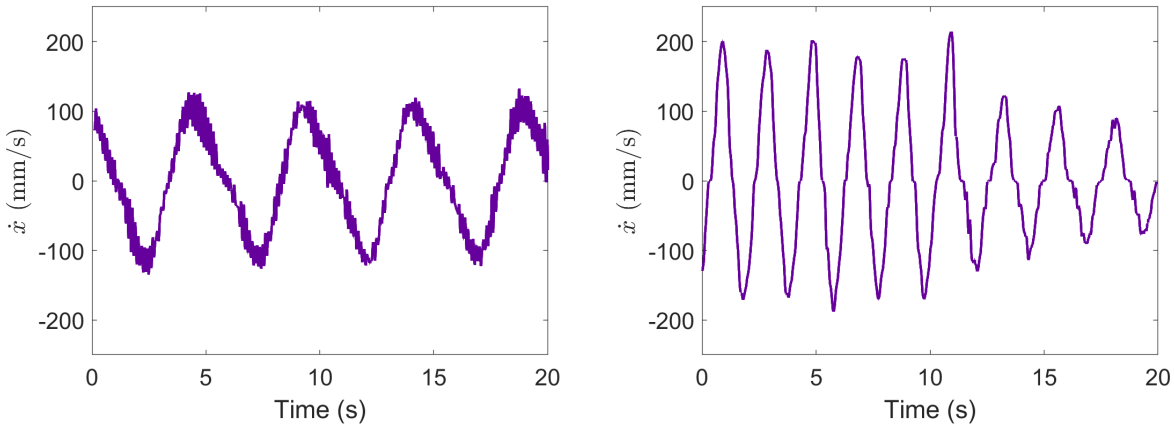


(d) PILCO



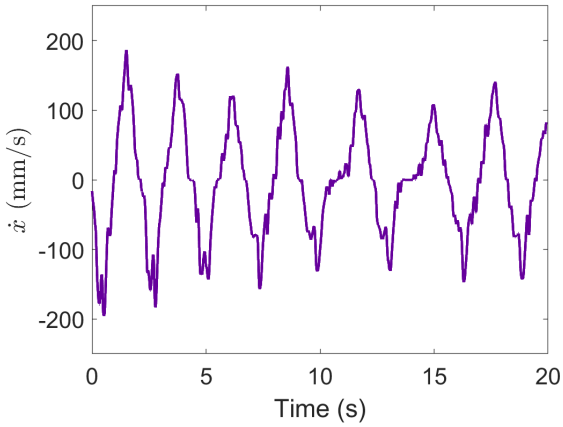
(e) Linear-Quadratic Regulator

Figure 4.8 State response α-coordinate for each algorithm over 20 seconds, steady.

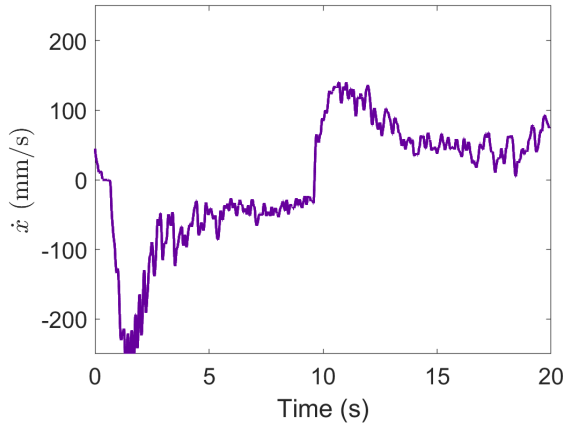


(a) Policy Gradient

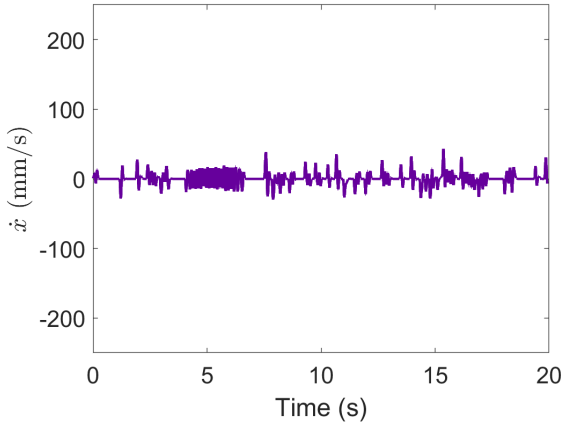
(b) Actor-Critic



(c) Proximal Policy Optimization

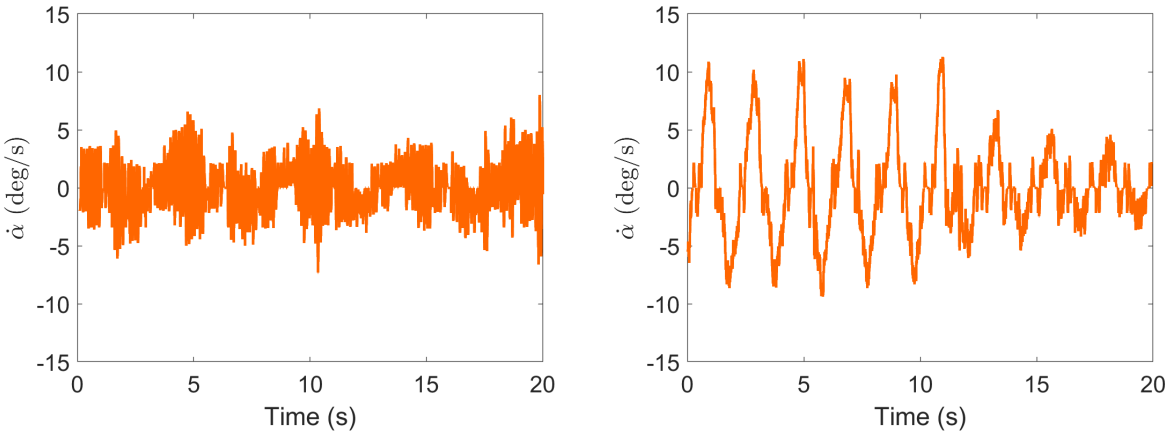


(d) PILCO



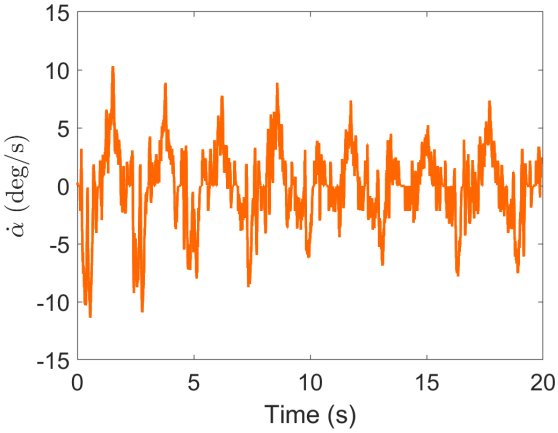
(e) Linear-Quadratic Regulator

Figure 4.9 State response $\dot{x}$-coordinate for each algorithm over 20 seconds, steady.

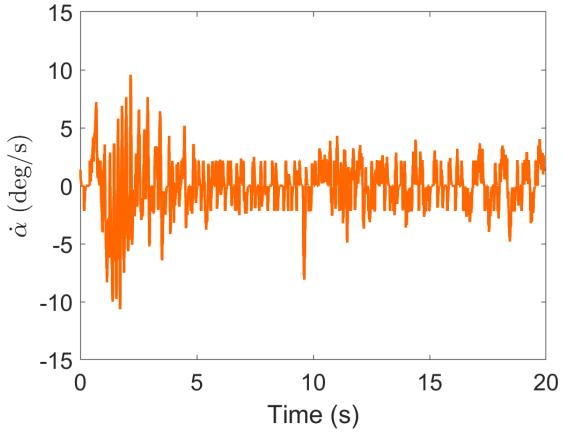


(a) Policy Gradient

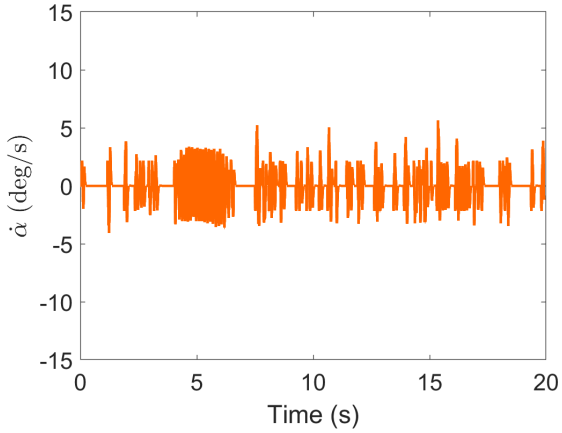
(b) Actor-Critic



(c) Proximal Policy Optimization

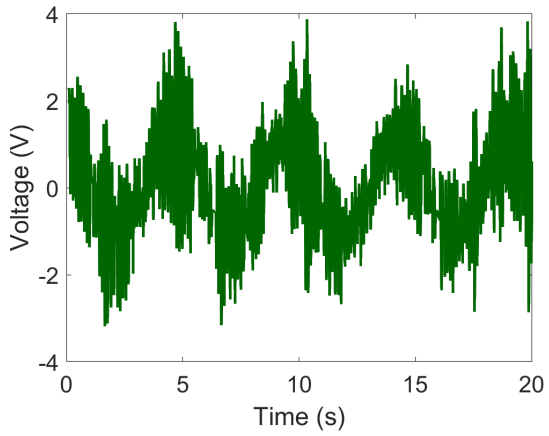


(d) PILCO

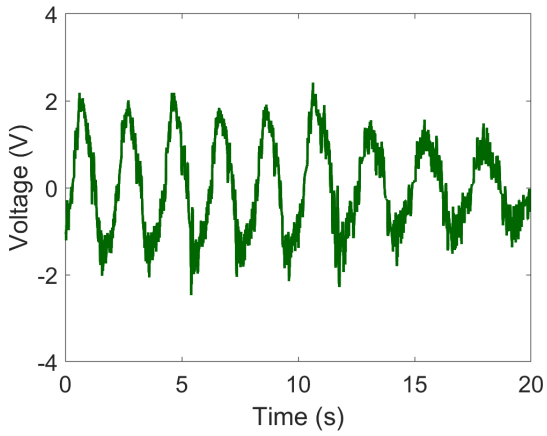


(e) Linear-Quadratic Regulator

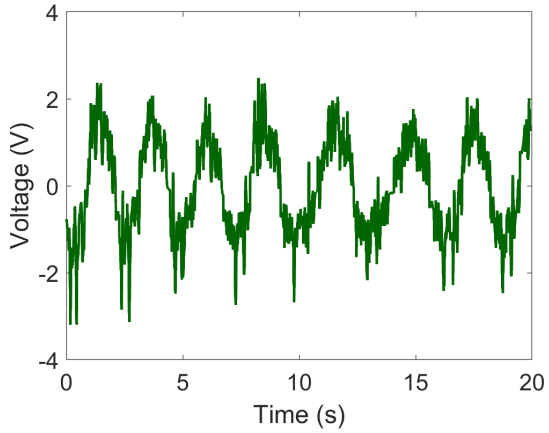
Figure 4.10 State response $\dot{\alpha}$-coordinate for each algorithm over 20 seconds, steady.



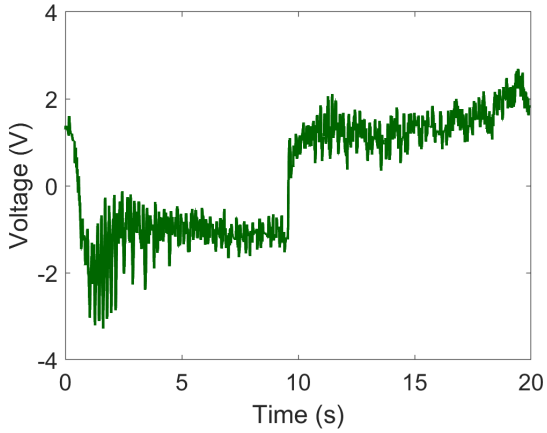
(a) Policy Gradient



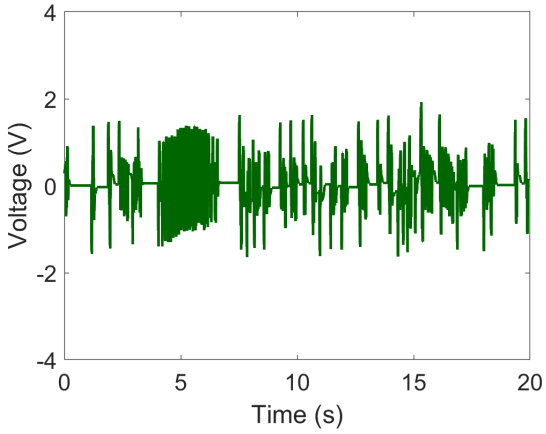
(b) Actor-Critic



(c) Proximal Policy Optimization



(d) PILCO



(e) Linear-Quadratic Regulator

Figure 4.11 Control effort actions chosen by each algorithm for 20 seconds, steady.

the force of impact, so the magnitude of each disturbance varies. Sometimes a tap would cause the pendulum to fall, so we reset and tapped slightly lighter next time. However, in cases where the same pendulum was tapped several times, the magnitudes were usually very similar. As a result, the reported values in Table 4.5 are close to the maximum disturbance that each model can recover from.

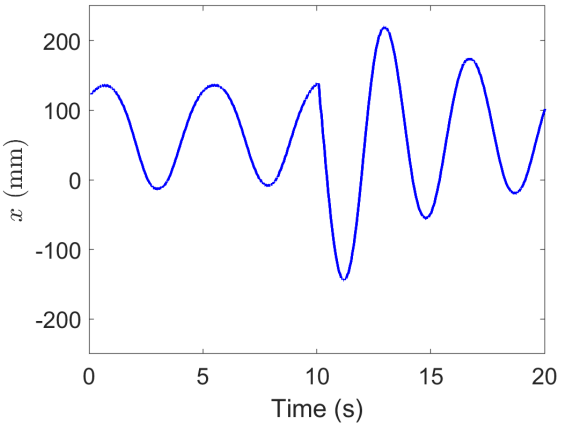
Table 4.5 How each trained model reacted to a disturbance. **Bold** numbers are better.

Algorithm	Max Angle	Max Voltage	Cart Movement	Overcompensation
PG	1.8°	10V	282mm	3.2°
AC	2.5°	**7.3V**	**175mm**	5.5°
PPO	0.9°	9.9V	247mm	8.7°
PILCO	3.2°	8.4V	550mm	3.7°
PILCO	**6.4°**	10V	770mm	7.4°

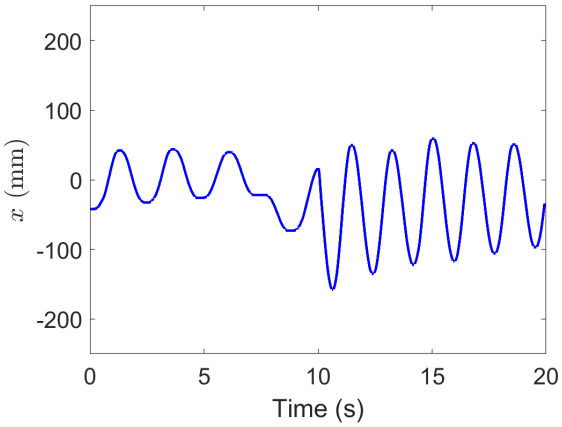
In Figure 4.12, which shows the x values of the cart, we can see that after getting tapped, the models resumed their oscillatory behaviour, albeit with a larger amplitude. Interestingly, Policy Gradient and PPO resumed their normal behaviour relatively quickly, while Actor-Critic took longer to reach steady state. Strangely enough, getting tapped actually *reduced* the amount PILCO travelled along the track, at least temporarily.

The same phenomenon can be seen for the angles α, as seen in Figure 4.13. Among the policy-based algorithms, Actor-Critic performed the best, absorbing a 2.5° disturbance without even saturating the motor. By comparison, Policy Gradient saturated the motor to recover from a 1.8° disturbance, and PPO was unable to recover from any disturbances of 1° or greater. Interestingly, it was able to absorb the initial disturbance but overcompensated, moving so quickly in the other direction that it deflected to 8.7°, as seen in Table 4.5. A possible reason Actor-Critic was able to handle the disturbance so well was the lack of control noise when compared to other algorithms. That does not explain why PPO performed so poorly though—perhaps the asymmetry of the controller.

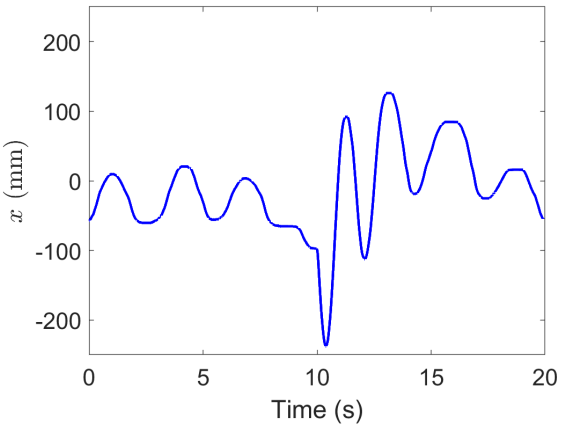
Surprisingly, PILCO was able to recover from the largest disturbance, displacing the pendulum 3.2°, without saturating the motor. In fact, it was able to withstand an impressive 6.4° disturbance, on par with a well-tuned LQR control, but fell immediately after. This was surprising because this was tested on several different models trained with PILCO, and only one of them was able to consistently recover from being tapped. In the others, the pole usually remained upright, but the tap would send the cart careening down the track, where it would inevitably fall off.



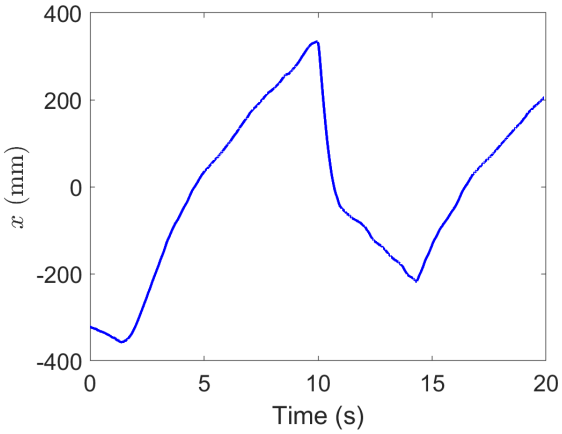
(a) Policy Gradient



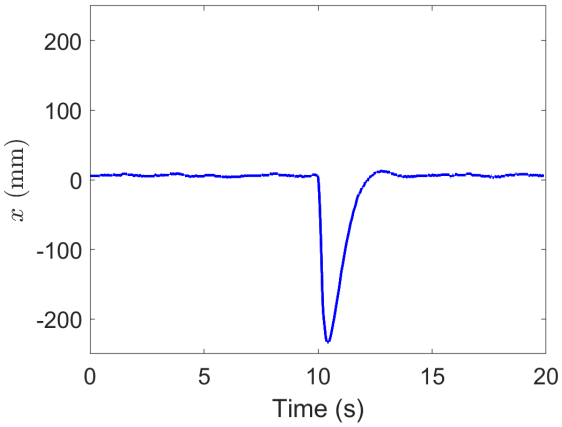
(b) Actor-Critic



(c) Proximal Policy Optimization

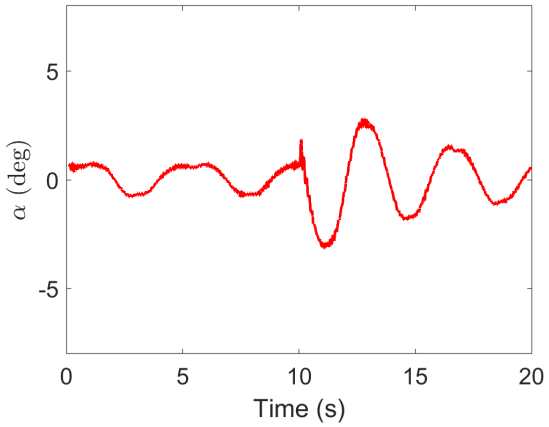


(d) PILCO: note expanded axis

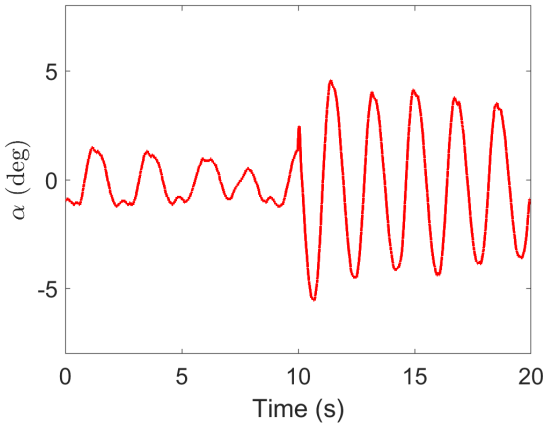


(e) Linear-Quadratic Regulator

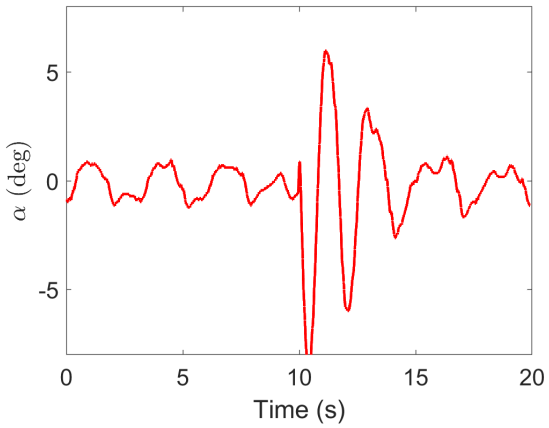
Figure 4.12 State response x-coordinate for each algorithm over 20 seconds, with a forced disturbance (a light tap on the pole) 10 seconds in.



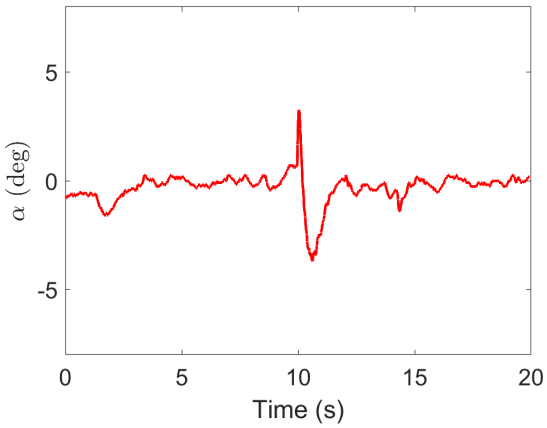
(a) Policy Gradient



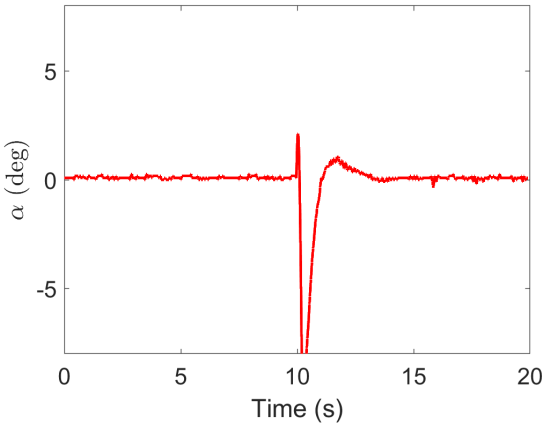
(b) Actor-Critic



(c) Proximal Policy Optimization



(d) PILCO



(e) Linear-Quadratic Regulator

Figure 4.13 State response α-coordinate for each algorithm over 20 seconds, with a forced disturbance (a light tap on the pole) 10 seconds in.

Figures 4.14–4.16 show the remaining variables in the state space $\dot{x}$ and $\dot{\alpha}$, and the voltage corresponding to the output of the controller. In all cases, the motor was briefly saturated, or nearly so in the case of Actor-Critic and PILCO, implying they could take a little more.

4.4 Discussion

As we have seen, all of the tested algorithms are able to train a virtual model well enough that it is able to balance a real inverted pendulum, while being robust to model errors, sensor noise, and disturbances of the system. However, not every hyperparamter combination resulted in a model that learned to balance every time. This optimization process took some trial-and-error in order to find a values that would train most consistently.

Even among fully trained algorithms, the performance in balancing the pole was underwhelming, with the cart oscillating across the track, and some of the controllers only being able to withstand the lightest disturbances. That being said, most of these algorithms were implemented in their vanilla form, with hardcoded hyperparameters specifying the conservative initial and terminal states of each trial. The algorithms did exactly what they were supposed to: keep the pole upright and the cart on the track. With a modified reward and loss function, more favourable results could be obtained.

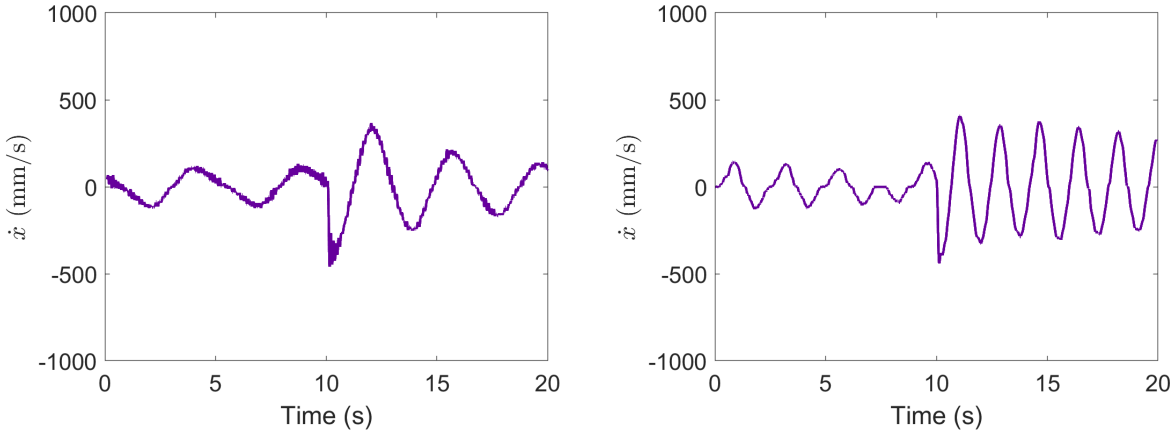
Reinforcement learning is not always the best strategy to approach a task. While it excels at learning a policy from scratch, policy-based algorithms offer no guarantees that the policy will converge to the optimal one. Often, prior information on the environment can be used to find a better traditional solution. Reinforcement learning, imitation learning, or transfer learning can be used to build on that traditional solution, allowing the best of both worlds. When the environment is unknown or simply too large for traditional techniques to be implemented, that is where policy-based algorithms excel.

4.4.1 Improving Results

As discussed in Section 2.5, there are a number of techniques we can use in order to improve on the goals outlined in Section 1.5. As a reminder, we wanted to:

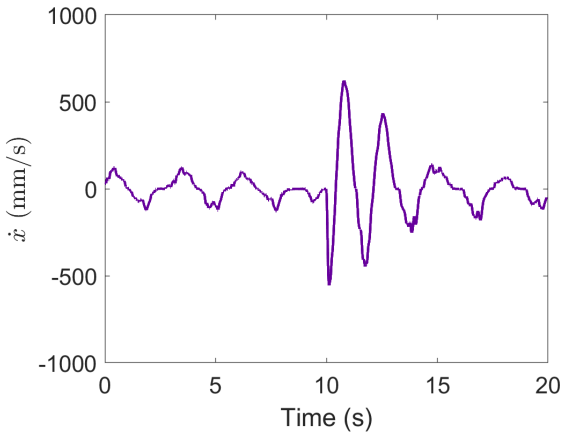
- find an algorithm that will train a virtual pole to balance as quickly as possible
- ensure the algorithm is robust enough to be used on a real inverted pendulum.

In order to meet the first goal, we used a relatively large learning rate. Smaller learning rates helped balance the virtual pendulum about as well as the models with

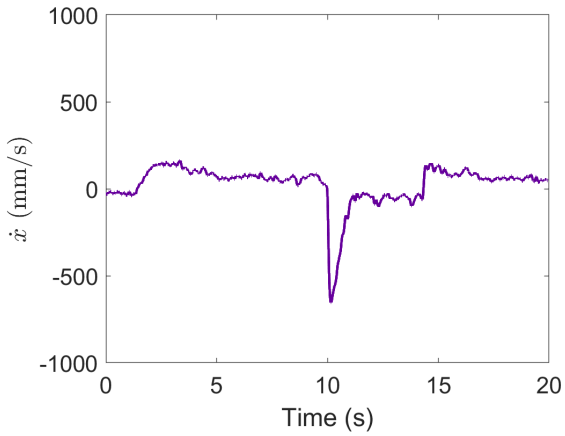


(a) Policy Gradient

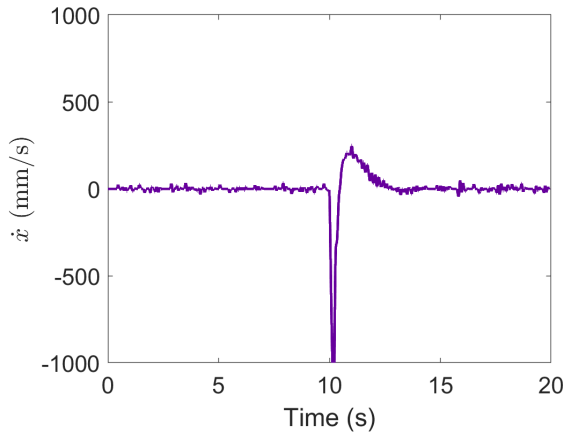
(b) Actor-Critic



(c) Proximal Policy Optimization

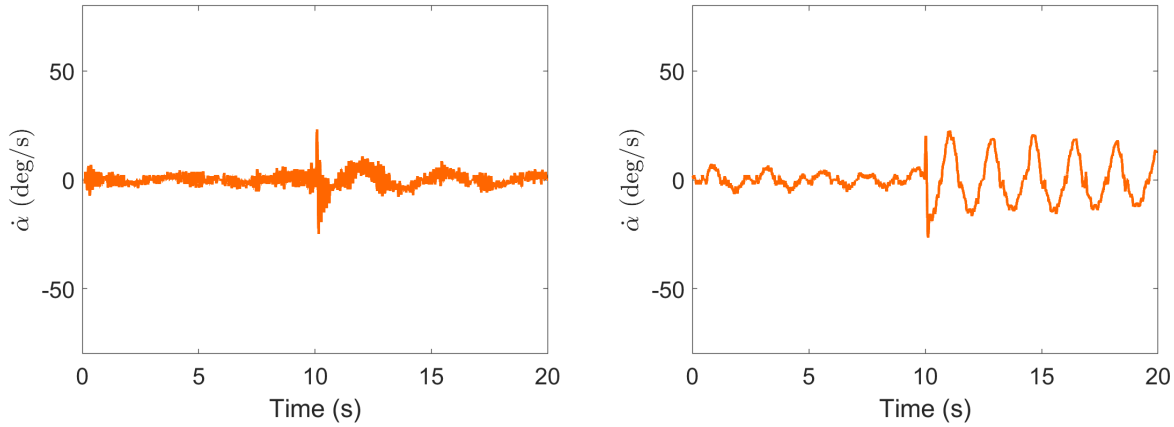


(d) PILCO



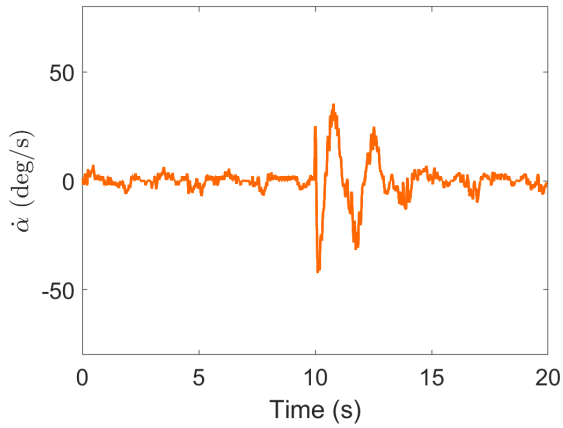
(e) Linear-Quadratic Regulator

Figure 4.14 State response $\dot{x}$-coordinate for each algorithm over 20 seconds, with a forced disturbance (a light tap on the pole) 10 seconds in.

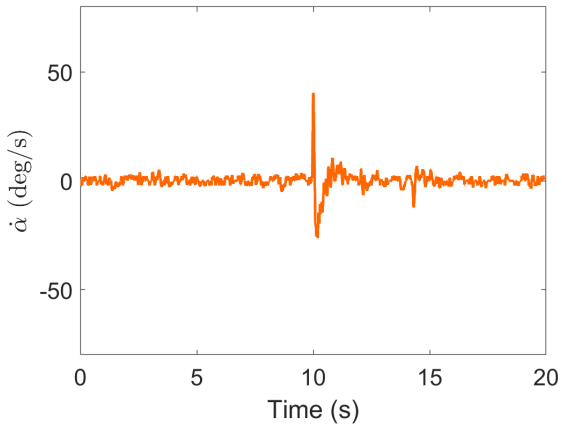


(a) Policy Gradient

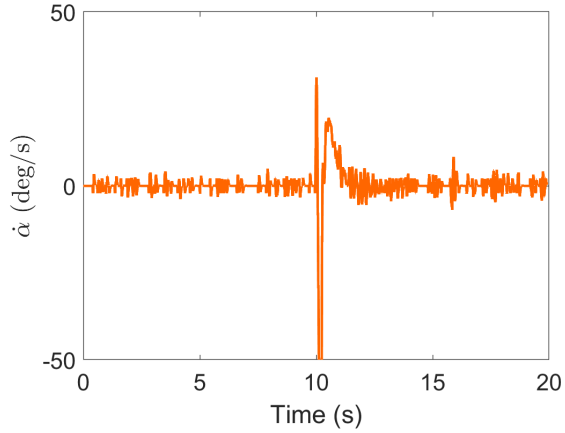
(b) Actor-Critic



(c) Proximal Policy Optimization

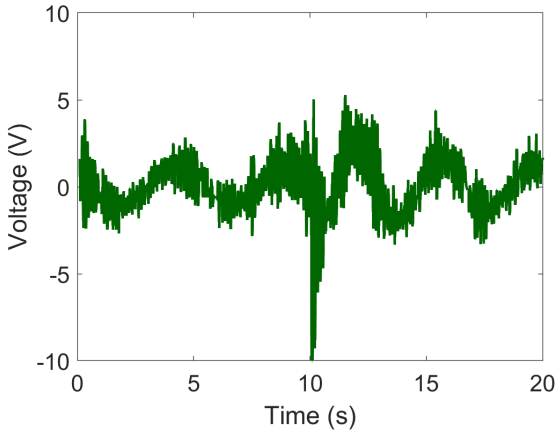


(d) PILCO

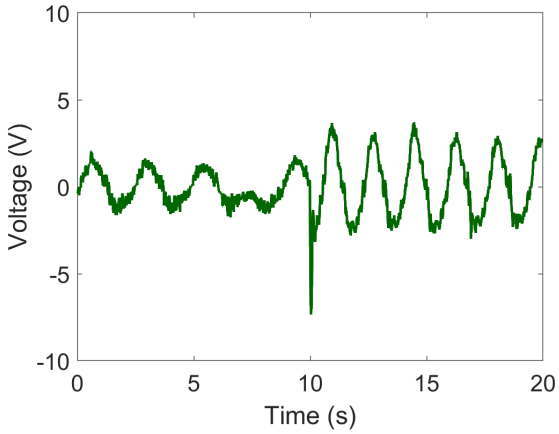


(e) Linear-Quadratic Regulator

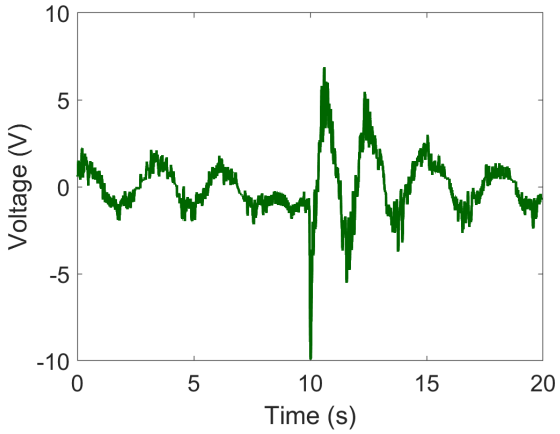
Figure 4.15 State response $\dot{\alpha}$-coordinate for each algorithm over 20 seconds, with a forced disturbance (a light tap on the pole) 10 seconds in.



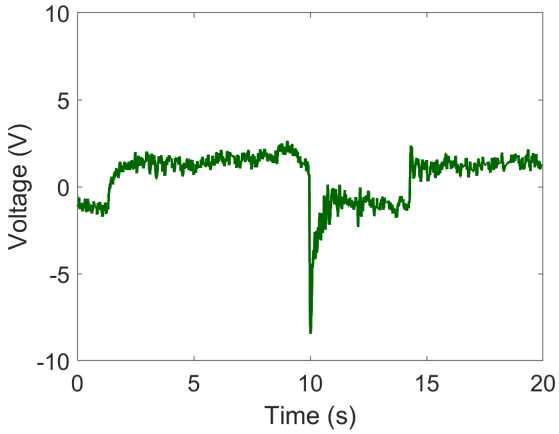
(a) Policy Gradient



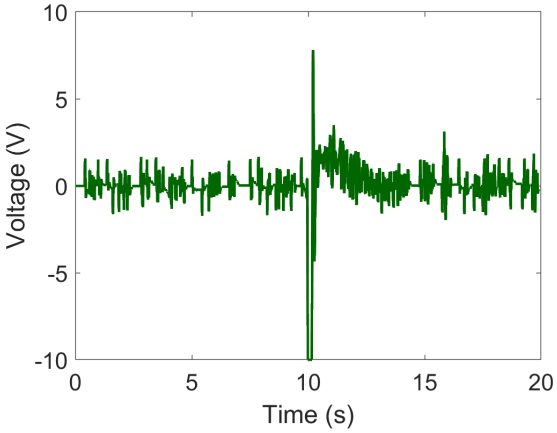
(b) Actor-Critic



(c) Proximal Policy Optimization



(d) PILCO



(e) Linear-Quadratic Regulator

Figure 4.16 Control effort actions chosen by each algorithm for 20 seconds, with a forced disturbance (a light tap on the pole) 10 seconds in.

$\eta = 10^{-2}$ reported in Table 4.1, but regularly took over 2500 trials to get there, which was determined to be too long. However, there is some evidence that smaller learning rates are able to learn more consistently; annealing the value of η over time may be a good compromise. Additionally, the 32 neurons in the hidden layer were also chosen arbitrarily, simply because it worked. As seen in Table 4.4, fewer neurons were still able to balance the pendulum, but having more neurons in the hidden layer tends to increase training speed, even if they are not necessary for learning. As we discussed in Corollary 1.5, increasing approximating power requires an exponential increase in width, but perhaps only a polynomial increase in depth; increasing the number of layers in a neural network may speed up training more than increasing the number of neurons in one layer.

Larger increases in speed came from switching algorithms. Actor-Critic improved on Policy Gradient by subtracting a baseline from the rewards function, reducing variance in the gradients. Additional variance reduction techniques, like collecting several trials of data before updating the parameters, similar to mini-batches in stochastic gradient descent could also be used. PPO provided further increases in training speed on the trials that it managed to learn an acceptable policy. However, its numerical instability and low success rate make the current implementation difficult to recommend for this purpose, especially considering its relative inability to recover from being tapped. Perhaps optimizing the hyperparameters or introducing entropy could help regularize its performance.

Nevertheless, the fastest, most stable, and most fascinating algorithm tested was PILCO. With the ability to learn a reasonable policy from a single rollout, the unprecedented data-efficiency combined with the excellent ability to recover from a disturbance make this algorithm fast and robust: exactly what we want. It is not without its problems though. While a competent high school student could understand the ideas behind most simple policy-based algorithms and Q-learning, PILCO is considerably more complex both to understand and to run. It was the only algorithm that was not coded from scratch for this dissertation. While there are numerous hyperparameters that can be tweaked and optimized, choosing a reasonable set is fairly straightforward: collect enough data to learn without running out of memory. Accordingly, its twin Achilles' heels are the computational complexity and strict memory requirements, making this algorithm impractical for high-dimensional environments or problems with many data points. Although training in a simulation allows it to take advantage of fast GPU compute, being able to solve the one-trial optimizations in under a minute, the longer the trials lasted, the slower the optimization got, occasionally taking more than 30 minutes for a single optimization step before running out of memory. PILCO is efficient because it has to be. If more than a handful of trials were needed, it will run out of memory before finding an acceptable

policy, more often than not. Variations and extensions to PILCO resolve some of these issues, extending its use-case beyond toy problems.

4.5 Conclusion

We have shown that it is possible to quickly and consistently train a simulation of an inverted pendulum that is robust enough to balance in the real world. For small tasks, PILCO provides the best performance, training incredibly quickly in good conditions, while simultaneously being able to absorb disturbances that it did not encounter during training. Traditional policy-based algorithms like Actor-Critic and PPO were also able to obtain some excellent results given the right hyperparameter set, and could likely do even better with some small changes like annealing the learning rate and using an experience buffer or entropy. Using virtual environments to balance an inverted pendulum with reinforcement learning is a proof of concept that could make other reinforcement learning applications more efficient. By letting computers train models in a simulation, they can quickly and efficiently collect large quantities of data, highlighting specific, rare, or dangerous states that would be difficult to obtain in the real world.

While this dissertation has been focused on the task of balancing a specific inverted pendulum, it would be reasonable to extend the applications to other controls where reinforcement learning is not normally used. We have already seen cars, drones, and robots trained in a simulation using Policy Gradient and PPO. It is not a stretch to imagine Actor-Critic being trained for satellite altitude adjustment and landing reusable rockets, or PILCO being used to optimize radiation therapy and control bipedal robots. The biggest challenge reinforcement learning faces is the data-inefficiency of the algorithms. By using techniques like domain randomization and variational autoencoders to bridge the sim-real gap, training in a simulation presents a viable solution to that problem, opening the door for reinforcement learning controls to be used in applications outside of academic research.

BIBLIOGRAPHY

[1] Abeysekera, B. & Wanniarachchi, W. K. "Modelling and Implementation of PID Control for Balancing of an Inverted Pendulum" (2018), pp. 43–53.

[2] Achiam, J. *Spinning Up in Deep Reinforcement Learning*. OpenAI. 2018.

[3] Amini, A. et al. "Learning Robust Control Policies for End-to-End Autonomous Driving From Data-Driven Simulation". *IEEE Robotics and Automation Letters* **5**.2 (2020), pp. 1143–1150.

[4] Amodei, D. et al. *AI and Compute*. 2018. URL: `https://openai.com/blog/ai-and-compute/`.

[5] Barto, A. G. et al. "Neuronlike adaptive elements that can solve difficult learning control problems". *IEEE Transactions on Systems, Man, and Cybernetics* **SMC-13**.5 (1983), pp. 834–846.

[6] Bernstein, A. "Modeling and Control: Applications to a Double Inverted Pendulum and Radio Frequency Interference". PhD thesis. North Carolina State University, 2018.

[7] Bishop, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[8] Bonatti, R. et al. "Learning Visuomotor Policies for Aerial Navigation Using Cross-Modal Representations". *CoRR* (2020). arXiv: `1909.06993`.

[9] Branwen, G. *GPT-3 Creative Fiction*. `https://www.gwern.net/GPT-3`. 2020.

[10] Brown, T. B. et al. *Language Models are Few-Shot Learners*. 2020. arXiv: `2005.14165 [cs.CL]`.

[11] Bryson, A. E. "A gradient method for optimizing multi-stage allocation processes". *Proceedings of the Harvard University Symposium on digital computers and their applications*. 1961.

[12] Campbell, M. et al. "Deep Blue". *Artificial Intelligence* **134** (2002), pp. 57–83.

[13] Cybenko, G. "Approximation by superpositions of a sigmoidal function". *Mathematics of Control, Signals and Systems*. Vol. 2. 1989, pp. 303–314.

[14] Dang, P. & Frank L, L. "Controller for swing-up and balance of single inverted pendulum using SDRE-based solution". *31st Annual Conference of IEEE Industrial Electronics Society, 2005. IECON 2005.* 2005.

[15] Davison, E. & Theory, I. F. of Automatic Control. Technical Committee on. *Benchmark Problems for Control System Design: Report of the IFAC Theory Committee*. International Federation of Automatic Control, 1990.

[16] Deisenroth, M. & Rasmussen, C. "PILCO: A Model-Based and Data-Efficient Approach to Policy Search". *ICML*. 2011, pp. 465–472.

[17] Deisenroth, M. P. "Efficient Reinforcement Learning with Gaussian Processes". PhD thesis. Karlsruhe Institute of Technology, 2010.

[18] Deisenroth, M. P. et al. "Gaussian Processes for Data-Efficient Learning in Robotics and Control". *IEEE Transactions on Pattern Analysis and Machine Intelligence* **37**.2 (2015), pp. 408–423.

[19] Duan, Y. et al. "Benchmarking Deep Reinforcement Learning for Continuous Control". *ICML*. 2016. arXiv: 1604.06778 [cs.LG].

[20] Florian, R. "Correct equations for the dynamics of the cart-pole system" (2005).

[21] Gal, Y. "Uncertainty in Deep Learning". PhD thesis. University of Cambridge, 2016.

[22] Gal, Y. et al. "Improving PILCO with Bayesian Neural Network Dynamics Models". *Data-Efficient Machine Learning workshop, ICML*. Vol. 4. 2016.

[23] Gatys, L. A. et al. *A Neural Algorithm of Artistic Style*. 2015. arXiv: 1508.06576 [cs.CV].

[24] Görtler, J. et al. "A Visual Exploration of Gaussian Processes". *Distill* (2019). https://distill.pub/2019/visual-exploration-gaussian-processes.

[25] Hernandez, D. & Brown, T. *AI and Efficiency*. 2020. URL: https://openai.com/blog/ai-and-efficiency/.

[26] Huang, J. *NVIDIA GeForce RTX 30 Series — Official Launch Event [4K]*. 2020. URL: https://www.youtube.com/watch?v=E98hC9e__Xs#t=2290.

[27] Ivakhnenko, A. G. & Lapa, V. G. *Cybernetic Predicting Devices*. JPRS 37, 803. CCM Information Corporation, 1965.

[28] Kalashnikov, D. et al. "Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation". *Proceedings of The 2nd Conference on Robot Learning*. Ed. by Billard, A. et al. Vol. 87. Proceedings of Machine Learning Research. PMLR, 2018, pp. 651–673.

[29] Karpathy, A. *Deep Reinforcement Learning: Pong from Pixels*. 2016. URL: https://karpathy.github.io/2016/05/31/rl/.

[30] Kelley, H. J. "Gradient Theory of Optimal Flight Paths". *ARS Journal* **30** (1960), pp. 947–954.

[31] Kennedy, E. "Swing-up and Stabilization of a Single Inverted Pendulum: Real-Time Implementation". PhD thesis. North Carolina State University, 2015.

[32] Kingma, D. P. & Ba, J. "Adam: A Method for Stochastic Optimization". *CoRR* (2015).

[33] Klöppelt, C. & Meyer, D. M. "Comparison of different Methods for Encoder Speed Signal Filtering exemplified by an Inverted Pendulum". *2018 19th International Conference on Research and Education in Mechatronics (REM)*. 2018, pp. 1–6.

[34] Koza, J. et al. "Automated Design Of Both The Topology And Sizing Of Analog Electrical Circuits Using Genetic Programming". *Artificial Intelligence in Design '96*. Dordrecht: Springer Netherlands, 1996, pp. 151–170.

[35] Krizhevsky, A. et al. "ImageNet Classification with Deep Convolutional Neural Networks". *Advances in Neural Information Processing Systems*. Ed. by Pereira, F. et al. Vol. 25. Curran Associates, Inc., 2012.

[36] LeCun, Y. et al. "Efficient BackProp". *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*. Berlin, Heidelberg: Springer-Verlag, 1998, pp. 9–50.

[37] Lillicrap, T. P. et al. *Continuous control with deep reinforcement learning*. 2015. arXiv: `1509.02971 [cs.LG]`.

[38] Lu, Z. et al. "The Expressive Power of Neural Networks: A View from the Width". *Advances in Neural Information Processing Systems*. Ed. by Guyon, I. et al. Vol. 30. Supplemental material available at `https://proceedings.neurips.cc/paper/2017/file/32cbf687880eb1674a07bf717761dd3a-Supplemental.zip`. Curran Associates, Inc., 2017.

[39] McKinney, S. M. et al. "International evaluation of an AI system for breast cancer screening". *Nature* **577**.7788 (2020), pp. 89–94.

[40] Micchelli, C. A. et al. "Universal Kernels". *Journal of Machine Learning Research* **7**.95 (2006), pp. 2651–2667.

[41] Michie, D. & Chambers, R. A. "Boxes: An Experiment in Adaptive Control". *Machine Intelligence* **2** (1968), pp. 137–152.

[42] Miranda, J. L. C. "Application of Kalman Filtering and PID Control for Direct Inverted Pendulum Control". MA thesis. California State University, Chico, 2009.

[43] Nielsen, M. A. *Neural Networks and Deep Learning*. Determination Press, 2015.

[44] OpenAI. *OpenAI Five*. https://blog.openai.com/openai-five/. 2018.

[45] OpenAI et al. "Solving Rubik's Cube with a Robot Hand". *CoRR* (2019). arXiv: 1910.07113 [cs.LG].

[46] Osiński, B. et al. "Simulation-Based Reinforcement Learning for Real-World Autonomous Driving". *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2020, pp. 6411–6418.

[47] Ozana, S. et al. "Design and implementation of LQR controller for inverted pendulum by use of REX control system". *2012 12th International Conference on Control, Automation and Systems*. 2012, pp. 343–347.

[48] Park, S. et al. "Minimum Width for Universal Approximation". *International Conference on Learning Representations*. 2021.

[49] Payne, C. *MuseNet*. 2019. URL: https://openai.com/blog/musenet/.

[50] Pinkus, A. "Approximation theory of the MLP model in neural networks". *Acta Numerica* **8** (1999), pp. 143–195.

[51] Polymenakos, K. et al. "SafePILCO: A Software Tool for Safe and Data-Efficient Policy Synthesis". *International Conference on Quantitative Evaluation of Systems*. Springer. 2020, pp. 18–26.

[52] Riedmiller, M. & Braun, H. "A direct adaptive method for faster backpropagation learning: the RPROP algorithm". *IEEE International Conference on Neural Networks*. Vol. 1. 1993, pp. 586–591.

[53] Riedmiller, M. "Neural reinforcement learning to swing-up and balance a real pole". *2005 IEEE International Conference on Systems, Man and Cybernetics*. Vol. 4. 2005, pp. 3191–3196.

[54] Rosenblatt, F. *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory, 1957.

[55] Samuel, A. L. "Some Studies in Machine Learning Using the Game of Checkers". *IBM Journal of Research and Development* **3**.3 (1959), pp. 210–229.

[56] Schaeffer, J. et al. "Checkers Is Solved". *Science* **317**.5844 (2007), pp. 1518–1522.

[57] Schrittwieser, J. et al. "Mastering atari, go, chess and shogi by planning with a learned model". *Nature* **588**.7839 (2020), pp. 604–609.

[58] Schulman, J. et al. "High-Dimensional Continuous Control Using Generalized Advantage Estimation". *CoRR* (2015). arXiv: 1506.02438 [cs.LG].

[59] Schulman, J. et al. “Trust Region Policy Optimization”. *CoRR* (2015). arXiv: 1502.05477.

[60] Schulman, J. et al. “Proximal policy optimization algorithms”. *CoRR* (2017). arXiv: 1707.06347.

[61] Silver, D. et al. “Mastering the game of Go with deep neural networks and tree search”. *Nature* **529** (2016), pp. 484–503.

[62] Silver, D. et al. “Mastering the game of go without human knowledge”. *nature* **550**.7676 (2017), pp. 354–359.

[63] Silver, D. et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. *Science* **362**.6419 (2018), pp. 1140–1144.

[64] Steinwart, I. “On the Influence of the Kernel on the Consistency of Support Vector Machines”. *Journal of Machine Learning Research* **2** (2001), pp. 67–93.

[65] Sutton, R. S. & Barto, A. G. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[66] Tai, M. M. “A Mathematical Model for the Determination of Total Area Under Glucose Tolerance and Other Metabolic Curves”. *Diabetes Care* **17**.2 (1994), pp. 152–154.

[67] Wahlström, N. et al. “From Pixels to Torques: Policy Learning with Deep Dynamical Models”. *arXiv e-prints* (2015). arXiv: 1502.02251 [stat.ML].

[68] Watkins, C. J. C. H. & Dayan, P. “Q-learning”. *Machine Learning*. 1992, pp. 279–292.

[69] Zhao, D. et al. “An Active Exploration Method for Data Efficient Reinforcement Learning”. *International Journal of Applied Mathematics and Computer Science* **29** (2019), pp. 351–362.

APPENDIX

Appendix A

PARAMETER VALUES

Table A.1 contains the model parameter values used for the Equations of Motion in simulation. Some of these values found in the technical specifications of the Quanser User Manual were found to be incorrect, and were determined experimentally through parameter identification in [31].

Table A.1 Model parameter values used in Equation 1.25–1.27.

Parameter	Description	Value
B_p	Viscous Damping Coefficient, as seen at the Pendulum Axis	$0.0024\text{N} \cdot \text{m} \cdot \text{s/rad}$
B_{eq}	Equivalent Viscous Damping Coefficient, as seen at the Motor Pinion	$5.4\text{N} \cdot \text{m} \cdot \text{s/rad}$
g	Gravitational Constant	9.8m/s^2
I_p	Pendulum Moment of Inertia, about its Center of Gravity	$8.539 \times 10^{-3}\text{kg} \cdot \text{m}^2$
J_p	Pendulum's Moment of Inertia at its Hinge	$3.344 \times 10^{-2}\text{kg} \cdot \text{m}^2$
J_m	Rotor Moment of Inertia	$3.90 \times 10^{-7}\text{kg} \cdot \text{m}^2$
K_g	Planetary Gearbox Ratio	3.71
K_t	Motor Torque Constant	$0.00767\text{N} \cdot \text{m/A}$
K_m	Back Electromotive Force (EMF) Constant	$0.00767\text{V} \cdot \text{s/rad}$
ℓ_p	Pendulum Length from Pivot to Center Of Gravity	0.3302m
M	Cart Mass	0.94kg
M_p	Pendulum Mass	0.230kg
R_m	Motor Armature Resistance	2.6Ω
r_{mp}	Motor Pinion Radius	$6.35 \times 10^{-3}\text{m}$