Value-Gradient Learning

Michael Fairbank

A thesis submitted for the degree of

PhilosophiæDoctor (PhD)

School of Informatics
Department of Computer Science
CITY UNIVERSITY
LONDON

February 2014

To Izabela, Helena and Emilia

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

I would like to thank my family for their support throughout the duration of this research, in particular to my wife, parents, brother and the family in Olkusz. I would also like to express my gratitude to my supervisor Eduardo Alonso for his guidance, helpful feedback, and ever-present support throughout my time at the university.

I have had the benefit of helpful discussions and feedback from various researchers in the ADPRL field, in particular from Andy Barto, Peter Dayan, Danil Prokhorov, David Silver, Csaba Szepesvári, Chris Watkins, Paul Werbos and Donald Wunsch, plus many anonymous reviewers.

I am grateful to have received travel bursaries to attend the International Joint Conference on Neural Networks 2012, in Brisbane, Australia, from the generous donors of City University Future fund, and from the IEEE Computational Intelligence Society. Also I am grateful to my employers at Abbey College London for allowing me to take leave during term time for conferences.

Various friends have given patient encouragement over the lifetime of this project, in some cases also including proofreading of early works and theorems. In particular, thank you to Warrick Barton, Jean-Michel Delhôtel, Nick Fairbank, Steph Fairbank, Kim Gilmore, Stephen Jackson, Richard Jacks, Nick McNally, Zahra Merali, Rob Neild, Chris Smith, Ian Tullie and Andrew Tuson.

Declaration

Abstract

This thesis presents an Adaptive Dynamic Programming method, Value-Gradient Learning, for solving a control optimisation problem, using a neural network to represent a critic function in a large continuous-valued state space. The algorithm developed, called VGL(λ), requires a learned differentiable model of the environment. VGL(λ) is an extension of Dual Heuristic Programming (DHP) to include a bootstrapping parameter, λ, analogous to that used in the reinforcement learning algorithm TD(λ). Online and batch-mode implementations of the algorithm are provided, and its theoretical relationships to its precursor algorithms, DHP and TD(λ), are described.

A theoretical result is given which shows that to achieve trajectory optimality in a continuous-valued state space, the critic must learn the value-gradient, and this fact affects any critic-learning algorithm. The connection of this result to Pontryagin's Minimum Principle is made clear. Hence it is proven that learning this value-gradient directly will obviate the need for local exploration of the value function, and this motivates value-gradient learning methods in terms of automatic local value exploration and improved learning speed. Empirical results for the algorithm are given for several benchmark problems, and the improved speed, convergence, and ability to work without local value exploration, is demonstrated in comparison to its precursor algorithms, TD(λ) and DHP.

A convergence proof for one instance of the VGL(λ) algorithm is given, which is valid for control problems with a greedy policy, and a general nonlinear function approximator to represent the critic. This is a non-trivial accomplishment, since most or all other related algorithms can be made to diverge under similar conditions, and new divergence proofs demonstrating this for certain algorithms are given in the thesis.

Several technical problems must be overcome to make a robust VGL(λ) implementation, and these solutions are described. These include implementing an efficient greedy policy, implementing trajectory clipping correctly, and the efficient computation of second-order gradients with a neural network.

Glossary

$U^c(\vec{u})$	An action-cost function (see below).
$\Delta\tau$	The sampling time for the underlying system being simulated/observed.
$\mathrm{diag}(\vec{x})$	The diagonal matrix with ith diagonal element equal to the ith component of the vector argument, $\vec{x}$.
$\bar{f}$	The learned model-function, which if learned correctly, will give the expectation of the true model-function, according to Eq. (1.10). This is a deterministic function.
$f(\vec{x}, \vec{u}, \vec{e})$	The model-function which takes the agent to the next state from the current state via Eq. (1.1).
$\vec{e}$	A random vector from the space E, sampled from a probability distribution $P_e(\vec{e})$. This random vector appears in Eqs. (1.1)-(1.5) and introduces stochastic effects into these equations. See Section 1.2.
$P_e(\vec{e})$	The probability distribution for the noise vector $\vec{e}$. See Section 1.2.
$\widetilde{Q}(\vec{x}, \vec{u}, \vec{w})$	The Model-Based Approximate Q Function, defined by Eq. (3.9).
$Q(\vec{x}, \vec{u}, \vec{w})$	The function approximator used in Q-Learning and ADHDP (Section 2.6).
τ	The real time of the physical system being observed or simulated. Note that τ is not constrained to the integers, unlike the integer step time, t.
$\bar{U}$	The learned instantaneous cost-function, which if learned correctly, will give the expectation of the true instantaneous cost-function, according to Eq. (1.10). This is a deterministic function.
Φ	A scalar function $\Phi(\vec{x}, \vec{e})$, $\vec{x} \in \mathbb{S}$, which gives a final impulse of cost at the instant when the agent reaches a terminal state (Section 1.2).
$\bar{\Phi}$	The learned terminal-cost function, which if learned correctly, will give the expectation of the true terminal-cost function, according to Eq. (1.10). This is a deterministic function.
$U(\vec{x}, \vec{u}, \vec{e})$	The instantaneous-cost function, or utility function, which specifies the cost received on transitioning from the given state $\vec{x}$. See Eq. (1.2).
$\mathrm{sgn}(x)$	The sgn function from computer programming languages, i.e. $\mathrm{sgn}(x) := \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$.

LIST OF ALGORITHMS

c A constant commonly used in an action-cost function to affect the sharpness of the step shape in the sigmoid function used. See Table 5.2 for details.

t The integer time step of the physical system being observed or simulated.

Action Network A neural network $A(\vec{x}, \vec{e}, \vec{z})$ (or similar function approximator), parameterised by a weight vector $\vec{z}$, that chooses actions to take at each state $\vec{x}$. Defined in Section 1.2.

Action-cost function A special function which can be added on to the cost function $U(\vec{x}, \vec{u}, \vec{e})$ to ensure that the greedy policy only chooses actions in a chosen range. See Section 5.2.2.

Actor See action network.

ADHDP A model-free VL algorithm by Paul Werbos that is equivalent to Q-Learning (Section 2.6).

ADP Adaptive Dynamic Programming, also known as Approximate Dynamic Programming.

ADPRL The combined fields of ADP and RL.

Approximate Value Function See critic.

Approximate Value Gradient See critic-gradient function.

Bellman Equation The recurrence equation defining an optimal value function, given by Eq. (2.9).

Bellman's Optimality Condition An optimality condition which states that if there exists a function which satisfies the Bellman Equation all over state space then any greedy policy on that function will be optimal. See Section 2.4, and also Chapter 4, for further details.

Bellman's Optimality Principle See Bellman's optimality condition.

BPTT The backpropagation through time algorithm, as described in Chapter 6, with pseudocode in Alg. 6.1.

Continuing Problem An environment in which trajectories are guaranteed to never meet a terminal state. See Section 1.2.

Cost-to-go Function The function $J(\vec{x}, \vec{\mathbf{e}}, \vec{z})$ defined by Eq. (1.6).

Critic The function $\widetilde{J}(\vec{x}, \vec{w})$, where $\vec{w}$ is the weight vector of a neural network. The critic function returns a neural-network's estimate of the cost-to-go function $J(\vec{x}, \vec{\mathbf{e}}, \vec{z})$.

Critic-Gradient Function The function $\widetilde{G}(\vec{x}, \vec{w})$, as defined in Section 3.4.

DHP The Dual Heuristic Programming algorithm, by Paul Werbos. Defined in Section 3.4 and pseudocode is given in Alg. 3.1.

DHP-Style Critic A synonym for vector critic.

Environment Functions These are the functions $f(\vec{x}, \vec{u}, \vec{e})$, $U(\vec{x}, \vec{u}, \vec{e})$ and $\Phi(\vec{x}, \vec{e})$, or their model-based equivalents.

Episodic Problem An environment in which trajectories are guaranteed to eventually meet a terminal state, no matter what the start point. See Section 1.2.

GDHP The Globalized Dual Heuristic Programming algorithm, by Paul Werbos. Defined in Section 3.4.4.

GDHP-Style Critic A synonym for scalar critic.

Generalised Policy Iteration The process of simultaneously training the action network and critic network, described in Section 2.7.5.

Greedy Policy A policy which chooses actions that minimise the immediate cost calculated by Eq. 1.7.

Greedy-on-$\widetilde{Q}$ Policy The approximate greedy policy defined by Eq. (5.2).

HDP The Heuristic Dynamic Programming algorithm, by Paul Werbos. Defined in Section 2.2. This is equivalent to TD(0) with a neural critic.

LET Locally Extremal Trajectory, defined in Section 7.2.4.

MLP A multilayer-perceptron neural network, for example as described by Bishop (1995).

Model Exploration This is the process of exploring the environment functions.

Model-Based ADPRL In this case, it is assumed that the environment functions will be learned by some machine-learning process and these learned functions will be available to the ADPRL algorithm.

Model-Based Approximate Q Function The function $\widetilde{Q}(\vec{x}, \vec{u}, \vec{w})$ defined by Eq. (3.9).

Model-Free ADPRL In this case, it is assumed that the environment functions are unknown to the ADPRL algorithm. The only way to obtain access to these functions is by the agent making actual interactions with the environment.

Model-Given ADPRL In this case, it is assumed that the environment functions are given and fully known to the ADPRL algorithm.

PGL Policy Gradient Learning. See Chapter 6 for details.

PMP Pontryagin's Minimum (or Maximum) Principle, described in Chapter 4.

Policy A generic term for an action network or greedy policy.

Policy Gradient The gradient $\frac{\partial J(\vec{x}, \vec{\mathbf{e}}, \vec{z})}{\partial \vec{z}}$, i.e. the gradient of total expected trajectory cost, J, with respect to the weight vector of the action network, $\vec{z}$; or some stochastic approximation to this gradient.

Policy Gradient Learning An algorithm which works by explicitly calculating the policy gradient.

Q-Learning A model-free VL algorithm by Chris Watkins that is equivalent to ADHDP (Section 2.6).

RL Reinforcement Learning.

Sampled Cost-to-go Function The function $\widehat{J}(\vec{x}, \vec{\mathbf{e}}, \vec{z})$ defined by Eq. (1.4) or Eq. (1.5), defined in Section 1.2.

Scalar Critic A scalar critic function $\widetilde{J}(\vec{x}, \vec{w}) \in \mathbb{R}$. This can be used to implement a critic-gradient function $\widetilde{G}(\vec{x}, \vec{w})$ by $\widetilde{G}(\vec{x}, \vec{w}) := \frac{\partial \widetilde{J}}{\partial \vec{x}}$, as defined in Section 3.4.1.

Target Value The function J'_t, defined by Eq. (2.3).

Target Value Gradient The function ${G'}_t$, defined by Eq. (3.7).

TD(λ) The “Temporal Differences” learning algorithm, by Richard Sutton, defined in Section 2.3.

Trajectory-Shorthand Notation The use of a subscripted time step to indicate that a variable or parenthesised derivative is to be evaluated at a particular trajectory time step. See Section 2.1.2 for details.

Value Exploration This is the process of exploring the state space to try to increase knowledge of the value-function or of the critic function.

Value Function See cost-to-go function.

Value Gradient The gradient $\frac{\partial \widetilde{J}(\vec{x},\vec{w})}{\partial \vec{x}}$, i.e. the gradient of the critic function $\widetilde{J}(\vec{x},\vec{w})$, with respect to the state vector of the agent in state space, $\vec{x}$. In some contexts, the term “value-gradient” may alternatively be interpreted to mean $\frac{\partial J(\vec{x},\vec{\mathbf{e}},\vec{z})}{\partial \vec{x}}$.

Value Iteration A form of generalized policy iteration where the action network is always kept fully trained in between every critic weight update. This makes the action network behave almost equivalently to a greedy policy (subject to function-approximation limitations of the action network).

Vector Critic A way of implementing the critic-gradient function $\widetilde{G}(\vec{x},\vec{w})$ as the output of a neural network with $\dim(\vec{x})$ output nodes. Defined in Section 3.4.1.

VGL An acronym for Value-Gradient Learning, so this includes the algorithms DHP, GDHP and VGL(λ).

VGLΩ(λ) The combination of the VGL(λ) algorithm with the special Ω_t matrix defined in Eq. (3.8).

VGL(λ) The name of the main new algorithm defined in this thesis: VGL(λ). This is defined by Eqs. (3.6), (3.5) and (3.7). Pseudocode is listed in Alg. 3.2 or Alg. 3.3.

VL An acronym for Value Learning, so this includes the algorithms HDP and TD(λ).

Part I

Introduction, Algorithm Definitions and Motivations

Chapter 1

Introduction

This thesis is a work that contributes to the fields of Adaptive Dynamic Programming (ADP) and Reinforcement Learning (RL). ADP is also known as Approximate Dynamic Programming. ADP and RL are together referred to as ADP/RL, or ADPRL for short.

ADPRL is concerned with optimising the behaviour of an agent, for example a robot or biological organism, which exists and moves around in an environment, and receives rewards or penalties based on the actions that it performs. The agent must learn to behave so as to minimise (/maximise) those penalties (/rewards). ADPRL could be used, for example, to stabilise an industrial plant (here the cost function would be how far the plant's state is from its desired set point), or optimise stock market picks (here the reward function would be financial gain). However ADPRL is potentially a much more important field in artificial intelligence and machine learning, because if the problem as so defined could be solved sufficiently efficiently, then general intelligent behaviour could be programmed to emerge. For example, this is the problem that evolution has solved for successful organisms (with the penalty/reward function here being death/successful procreation, and the actions that lead to these outcomes being the decisions and actions made by the organism during its lifetime); and this process has resulted in intelligent creatures.

One key approach to efficiently finding optimal behaviour is to use variants of Dynamic Programming, as created by Richard Bellman (Bellman, 1957). The key idea there is to assign scalar values to each different state that the agent can be in, such that those values rate how "good" each state is. The values form a scalar field over the state space described by the "value function", also known as the "cost-to-go" function. The value function can be learnt from direct or simulated experience of the agent. Once the value function is learned, the agent can learn to behave "greedily" towards it, which means to choose actions that will lead to states that the value function rates as "best", and hence the agent can (eventually) learn to behave optimally. The Bellman Equation is a condition that the value-function must obey to guarantee optimality.

Paul Werbos initiated the field of ADP. As he wrote about the importance of using Bellman's method, "*In the future, we may recognize that trying to build or understand intelligent systems without exploiting the Bellman Equation is like trying to build hardware without knowing Maxwell's laws. There are times when proper understanding and use of one key equation is the key bridge that makes it possible to connect valid large global goals to the world of concrete mathematical reality, i.e., working designs and valid models.*" (Werbos, 2008, p.898). Similarly, regarding RL, Sutton and Barto wrote "*The central role of value estimation is arguably the most important thing we have learned about reinforcement learning over the last few decades*" (Sutton and Barto, 1998, p.8). While there are other valid research methods which could be used to attack the ADPRL problem, such as genetic algorithms or numerical methods, this thesis concentrates on value-function based methods.

Solving the Bellman Equation requires learning the value function at every point in state space. This process will be referred to as *value exploration*. The original dynamic-programming method to do this was to consider every single point of the state space in turn, in one or more full state-space sweeps. A major difficulty here is that generally there are a huge, or infinite, number of possible states in the state space, an issue Bellman referred to as the *curse of dimensionality*. Therefore each state-space sweep is very computationally expensive. A more efficient method is to concentrate on complete trajectories, one at a time; and this is the preferred method of the ADPRL algorithms presented in this thesis.

A lot of successful work has been done with tabular representations of the value function. However, if the state space is continuous-valued, as it usually is in the real world, then there must be an infinite number of possible states, and no tabular representation will cope with that. Similarly, no agent or biological organism can remember values for all those possible states, or spend the time necessary to learn the correct values for all of them. So some form of generalisation and data compression is required, and both of these problems are solved by using a function approximator, e.g. a neural network, to represent the value function. This function approximator is called the *critic*. Consequently, two other synonyms for the ADP field of research are “Adaptive Critic Designs” and “Neuro-dynamic programming”.

Heuristic Dynamic Programming (HDP) is an early ADP algorithm. This kind of critic-learning algorithm will be referred to as a *value-learning* (VL) algorithm, since it works by learning the “values” of the value-function. In the RL literature, TD(λ) by Richard Sutton (Sutton, 1988) is a value-learning algorithm which extends HDP to include a “bootstrapping” parameter λ. Varying this parameter can affect learning speed and the algorithm’s stability and convergence properties. Apart from some minor historical differences, HDP is equivalent to TD(0).[1]

In an ADPRL critic-based system, the critic enables the agent to behave greedily, i.e. to choose actions rated as best by the critic. It is proven in this thesis (in Section 4.1.1) that if the Bellman Equation is to be satisfied, then the agent must eventually learn to behave greedily towards the critic. However if the agent behaves greedily from the start of learning, then the value-exploration that is necessary for solving the Bellman’s Equation may be excluded, and therefore optimal behaviour would never be achieved. Therefore most ADPRL schemes do not simply use greedy behaviour throughout the whole of learning, even if ideally they would like to do so. This difficulty is part of what is known as the *exploration-versus-exploitation dilemma*.

A way around this problem comes from considering what information greedy behaviour needs. To choose an action greedily, the agent is not concerned with the average magnitude of the values of all the actions available to it, but only the relative values of them, as it only needs to be able to pick the best one. In continuous-valued state spaces, these relative values are encapsulated in the *value gradient*, that is the gradient of the value function with respect to the agent’s state vector. This is what is required for the agent to choose the best action.

The value gradient acts as an arrow pointing in the direction in which states improve the most, and the critic-learning system must make that arrow point in the correct direction, and also the agent must learn to follow it. Since this arrow is so important for all learning systems that work with a Bellman Equation in continuous-valued state

[1]HDP was originally defined with a neural-critic, but TD(0) was originally defined with a tabular critic.

spaces, it would make sense to concentrate on learning this arrow's direction and magnitude directly, instead of learning it indirectly by first learning the values of all of the points in state space immediately surrounding that arrow. Clearly we may expect to attain a massive speed up in learning if we learn the value-gradient (the arrow) directly. And it also might solve the exploration-versus-exploitation problem which occurs with purely greedy behaviour. That is the objective of this thesis: to consider and extend ADPRL methods which learn value gradients directly.

The speed-of-learning advantage for value-gradient learning (VGL) methods led Werbos to invent two VGL algorithms that are pre-existing to the VGL algorithms presented in this thesis. These two prior algorithms are Dual Heuristic Programming (DHP) and Globalized Dual Heuristic Programming (GDHP) (Werbos, 1990b, 1992b). As motivation for creating these algorithms, Werbos (2007) wrote "*I learned very early that the [HDP] method does not scale very well, when applied to systems of even moderate complexity. It learns too slowly. To solve this problem, I developed the core ideas of two new methods— dual heuristic programming (DHP) and Globalized DHP (GDHP).*".

In the same way that TD(λ) extends TD(0)/HDP, this thesis makes an analogous extension to the value-gradient algorithm, DHP, by defining a new algorithm called VGL(λ). VGL(λ) is an extension of DHP which includes a constant parameter λ. As with TD(λ), the λ parameter in VGL(λ) can affect the learning speed and convergence properties of the algorithm. This makes it possible to give a convergence proof for one instance of the new algorithm (other instances of the algorithm can still be made to diverge). This convergence proof is for a full non-linear function approximator representing the value function plus an agent whose behaviour is always greedy towards the changing value function. This type of convergence proof is much sought after in the ADPRL literature, with most (or all) existing proofs being only valid for non-greedy behaviour or linear function approximation.

In the thesis, motivations for using VGL algorithms are discussed, which as mentioned above are principally speed of learning, and also the ability to do local value exploration automatically. This automatic local value exploration comes for free with VGL methods, since the mere act of learning the arrow (i.e. the value gradient) will point the agent in the correct direction. This obviously contrasts to value-learning methods where learning a scalar value does not point the agent in any useful direction. In here, classical value-learning methods must learn all of the values surrounding any point in state space before the gradient that gives the arrow can be deduced, which explains the need for value exploration and slower speed of value-learning systems.

It should be noted, however, that there is a separate form of exploration other than value exploration which also needs to be performed, and which VGL methods do not address. This other form of exploration is the need to know the model functions, and will be referred to as *model exploration*. The model functions describe which state the agent will travel to from any given current state and any chosen action. Model exploration is a major component of exploration that many ADPRL systems need to address, and this thesis does not tackle that problem at all. The VGL methods reduce the need for value exploration only.

The thesis also gives theoretical properties of value-gradient algorithms, including: the convergence proof; an equivalence between two previously unrelated algorithms, backpropagation through time and value-gradient learning; optimality conditions for trajectories when learning by value gradients; and divergence properties of variant algorithms. The thesis also describes solutions to practical issues for making robust implementations of any value-gradient learning algorithm, including clipping, second-order gradient-finding algorithms in neural networks, plus details of how an agent can most efficiently choose actions from the value function.

In the rest of this introductory chapter, the motivation and challenges for the research line of this thesis are described in Section 1.1; a formal statement of the ADPRL optimisation problem is given in Section 1.2; a short literature review is given in Section 1.3 including details of how the new algorithms defined in the thesis relate to pre-existing ADPRL algorithms; Section 1.4 is dedicated to a discussion about model-free versus model-based algorithms; Section 1.5 describes some closely related fields to ADPRL and also how RL and ADP are related to each other within ADPRL; Section 1.6 summarises the original contributions of the thesis; and Section 1.7 gives a brief outline of the structure of the rest of the thesis.

1.1 Motivation and Challenges

Introducing a function approximator for the value function causes theoretical and practical difficulties in ADPRL. For example, the dynamic-programming method is only proven to converge for an exact representation of the value function. Similarly many of the theoretical results from the ADPRL literature are for this exact, or tabular, representation. However when a neural network is used for the value function, learning values in one point of state space can unlearn values previously learned in other points of state space. This is because a function approximator can only have finite flexibility - bending it in one place will cause disturbances in other places. This makes proving convergence of the learning algorithms used by ADPRL difficult.

Another difficulty is a co-dependence in that the "value" of any state depends upon the future actions that an agent subsequently follows from that state; while simultaneously the optimal actions depend on the value function found by Bellman's equation. Hence changing the state values will change the agent's behaviour, and changing the agent's behaviour will change the state values. This co-dependence makes proving convergence of critic-learning algorithms hard. Therefore many critic-convergence proofs are only valid when the agent behaviour is static.

When the agent's behaviour is not fixed, the situation is more challenging. The tabular-critic case was proven to converge by Howard (1960a). However when the critic is a function approximator, there has been only limited progress in the literature. Some successes do exist for the case of changing agent behaviour (e.g. Kakade, 2001; Sutton et al., 2000, 2001). However these successes are not for the greedy policy, and only for the situation where the critic is a "compatible" function approximator (defined in Section 2.7.5). The convergence proof given in this thesis (in Chapter 8) is an improvement upon this, because it is valid for a general non-linear critic, and also where the behaviour of the agent is greedy.

A central motivation for learning value gradients is that value gradients act as "arrows" which point the agent in the correct direction. If the agent is programmed to follow these arrows as closely as possible, then learning the value gradients will cause the trajectories found by the agent to bend themselves into locally optimal shapes. This is proven in the thesis for the VGL(λ) algorithm (in Chapter 7; proven for deterministic environments only). In contrast, simply learning the values along a trajectory will not achieve this, and hence these value-learning algorithms will generally converge to suboptimal trajectories if the ability to explore is taken away from them. Counterexamples demonstrating this suboptimal convergence are given in Chapter 3 (in Fig. 3.1 and in Section 3.7), but these are just manifestations of the exploration-versus-exploitation dilemma (with respect to value-exploration, as opposed to model-exploration).

Doya (2000) extended VL methods to continuous-valued state spaces, and interestingly that research also used a value gradient. The value gradient was not used for learning (so it was not a VGL method), but it was used for determining the agent's greedy actions. This does confirm that value gradients are especially useful in continuous spaces, and are what "drive" the greedy policy. Hence switching from values to value gradients is a very natural way to extend ADPRL efficiently into continuous spaces.

VGL methods are therefore designed for continuous-valued state spaces. However this is also a limitation, in that they are not suited to discrete-valued state spaces. Two more limitations of VGL methods are that they require known and differentiable model functions. A fourth limitation is that in stochastic environments, VGL methods are only derived exactly in the case of additive noise; in more general stochastic environments, VGL methods sometimes rely on approximations (full details are given in Section 3.2.1). In comparison, VL methods do not have any of these limitations. It is therefore hoped that the benefits of VGL methods (i.e. with respect to learning speed and automatic local value exploration) will outweigh these four limitations, in some circumstances.

To make any implementation of a VGL system, including the DHP, GDHP and VGL(λ) algorithms, there are a number of technical hurdles involved, which are described and solved in this thesis. These issues are:

1. How to make an efficient implementation of a greedy policy (Chapter 5).

2. The importance of using clipping correctly in the final time step of a trajectory (Chapter 10).

3. Efficient implementation of the second-order backpropagation that is necessary for certain value-gradient critic architectures (Chapter 11).

1.2 Formal Specification of the ADPRL Optimisation Problem

To avoid having to continually specify the option of either maximising a reward or minimising a cost, the convention of minimising a cost will be adopted. This situation can easily extended to that of maximising a reward simply by including an extra negative sign before all costs considered, and swapping all instances of "minimisation" to "maximisation".

ADPRL seeks to train an agent to choose actions that minimise a total long-term cost. For example a typical scenario is an agent wandering around in a state space $\mathbb{S} \subset \mathbb{R}^n$, such that at integer time t it has state vector $\vec{x}_t \in \mathbb{S}$. At each time t the agent chooses an action $\vec{u}_t$ (from an action space $\vec{u}_t \in \mathbb{A}$) which takes it to the next state according to the environment's model function

$$\vec{x}_{t+1} = f(\vec{x}_t, \vec{u}_t, \vec{e}_t), \tag{1.1}$$

where $\vec{e}_t$ is a noise vector sampled from a space $E \subset \mathbb{R}^n$ with probability distribution function $P_e(\vec{e}_t)$. Each action choice $\vec{u}_t$ results in the agent receiving an immediate scalar cost U_t, given by the function

$$U_t = U(\vec{x}_t, \vec{u}_t, \vec{e}_t). \tag{1.2}$$

The agent keeps moving, forming a trajectory of states $(\vec{x}_0, \vec{x}_1, \ldots)$, which terminates if and when a state from the set of terminal states $\mathbb{T} \subset \mathbb{S}$ is reached. If a terminal state $\vec{x}_t \in \mathbb{T}$ is reached then a final instantaneous cost $\Phi_t = \Phi(\vec{x}_t, \vec{e}_t)$ is given which is independent of any action.

The functions $f(\vec{x}, \vec{u}, \vec{e})$, $U(\vec{x}, \vec{u}, \vec{e})$ and $\Phi(\vec{x}, \vec{e})$ will collectively be referred to as *the environment functions*. Since the environment functions are functions of a random variable, $\vec{e}$, it means they can be used to represent stochastic environments. If the environment functions are independent of $\vec{e}$, then the *environment is deterministic*, and in this case $\vec{e}$ can be omitted as an argument from the environment functions. Otherwise the *environment is stochastic*. In a stochastic physical environment, the noise variable $\vec{e}$ could be an unobservable quantity coming from the environment; or in the case of a simulated environment it could come explicitly from a computer's random number generator. Further details of the randomness introduced by the noise vector $\vec{e}$ are given in the next subsection 1.2.1.

Different ADPRL approaches assume different levels of knowledge of the environment functions. For example, in "model-free RL" it is assumed that these functions are not known or accessible other than by actual interactions with the environment; but in other forms of ADPRL it is assumed the algorithm does have access to the definitions of these three functions. These possibilities are discussed further in Section 1.4.

In this thesis, analysis is restricted to the situation where the state vector $\vec{x}$ is fully observable and known at every time step, although it should be noted that handling "partially observable" state vectors is a major related research topic.

An *action network* is a function approximator (for example a neural network) of the form $A(\vec{x}, \vec{e}, \vec{z})$, where $\vec{z}$ is the parameter vector (weight vector) of the function approximator. The action network's purpose is to calculate actions,

$$\vec{u}_t = A(\vec{x}_t, \vec{e}_t, \vec{z}), \tag{1.3}$$

to take from any given state $\vec{x}_t$. The action network is also known as the *actor*, or *policy function*.

If the function $A(\vec{x}, \vec{e}, \vec{z})$ is independent of $\vec{e}$ then the action network is deterministic, and the argument $\vec{e}$ can be omitted. If the action network is stochastic, then the components of $\vec{e}$ that affect $A(\vec{x}, \vec{e}, \vec{z})$ would usually come from a random-number generator. This is useful because in some cases deliberate randomness is introduced into the action-network, for example to encourage exploration by the agent.

For a trajectory starting from state $\vec{x}_0$, if the trajectory eventually reaches a terminal state at some time step T, then the *sampled cost-to-go function* is defined to be:

$$\widehat{J}(\vec{x}_0, \vec{\mathbf{e}}_0, \vec{z}) := \sum_{t=0}^{T-1} \gamma^t U_t + \gamma^T \Phi(\vec{x}_T, \vec{e}_T), \tag{1.4}$$

subject to Equations (1.1), (1.2) and (1.3), where $\gamma \in [0, 1]$ is a constant *discount factor* that specifies the relative importance of long-term costs over short term ones, and where the bold $\vec{\mathbf{e}}_t$ is shorthand for all the future noise vectors from time step t onwards, i.e. $\vec{\mathbf{e}}_t := (\vec{e}_t, \vec{e}_{t+1}, \ldots, \vec{e}_T)$. Note that T, the time step at which a terminal state is first reached, will in general be dependent on $\vec{x}_0$, $\vec{\mathbf{e}}_0$ and $\vec{z}$. If a terminal state is never met, then the trajectory becomes infinitely long, and similarly $\vec{\mathbf{e}}_t := (\vec{e}_t, \vec{e}_{t+1}, \ldots)$ will be infinitely long, and the sampled cost-to-go function's definition simplifies down to,

$$\widehat{J}(\vec{x}_0, \vec{\mathbf{e}}_0, \vec{z}) := \sum_{t=0}^{\infty} \gamma^t U_t. \tag{1.5}$$

In this case we would need to ensure $\gamma < 1$ to keep the sum of Eq. (1.5) finite.[2]

If the environment is such that a terminal state is guaranteed to be met at some time, then we say the problem is *episodic*, and Eq. (1.4) is appropriate. If the environment is such that a terminal state will never be reached, then the problem is *continuing*, and Eq. (1.5) is appropriate.

Algorithm 1.1 gives example pseudocode illustrating how Equations (1.1)-(1.5) could be used together to calculate a full trajectory and sampled trajectory cost $\widehat{J}_0 := \widehat{J}(\vec{x}_0, \vec{\mathbf{e}}_0, \vec{z})$, from a given start point $\vec{x}_0$.

Algorithm 1.1 Trajectory Unroll and Total-Cost Calculation in the ADPRL problem.

```
1:  $t \leftarrow 0$
2:  $\widehat{J}_0 \leftarrow 0$
3:  while $\vec{x}_t \notin \mathbb{T}$ do
4:      $\vec{u}_t \leftarrow A(\vec{x}_t, \vec{e}_t, \vec{z})$
5:      $\vec{x}_{t+1} \leftarrow f(\vec{x}_t, \vec{u}_t, \vec{e}_t)$
6:      $U_t \leftarrow U(\vec{x}_t, \vec{u}_t, \vec{e}_t)$
7:      $\widehat{J}_0 \leftarrow \widehat{J}_0 + \gamma^t U_t$
8:      $t \leftarrow t + 1$
9:  end while
10: $T \leftarrow t$
11: $\widehat{J}_0 \leftarrow \widehat{J}_0 + \gamma^t \Phi(\vec{x}_t, \vec{e}_t)$
```

This pseudocode omits details of the sampling of the random vector $\vec{e}_t$, which is assumed to come from the environment and/or from a random-number generator. Similarly, this detail is omitted from all of the pseudocode appearing in this thesis.

The expectation of the sampled cost-to-go function is written as

$$J(\vec{x}_0, \vec{\mathbf{e}}_0, \vec{z}) := \mathbb{E}\left(\widehat{J}(\vec{x}_0, \vec{\mathbf{e}}_0, \vec{z})\right), \tag{1.6}$$

where $\mathbb{E}(\cdot)$ denotes expectation and $\widehat{J}$ is defined by Eqs. (1.4) and (1.5). This function, $J(\vec{x}_0, \vec{\mathbf{e}}_0, \vec{z})$, is known as the *value function*, or simply as the *cost-to-go function* (i.e. without the "sampled" prefix). The hat on the variable-name $\widehat{J}$ denotes that the quantity is a random sample of the true expectation given by J.

The ADPRL problem is defined to be the task of choosing a weight vector $\vec{z}$ of the action-network such that $J(\vec{x}_0, \vec{\mathbf{e}}_0, \vec{z})$ is minimized from any start state $\vec{x}_0$, subject to Eqs. (1.1), (1.2) and (1.3).

ADPRL also often uses a second neural network (or function approximator), $\widetilde{J}(\vec{x}, \vec{w}) \in \mathbb{R}$, with weight vector $\vec{w}$, known as the *critic* or *approximate value function*. The tilde

[2]Some ADPRL methods can allow continuing problems with $\gamma = 1$ by considering the average infinite future cost per time step, but these methods are not considered in this thesis.

on the variable-name $\widetilde{J}$ denotes that the quantity is a neural-network approximation to the underling variable, J. This convention is adopted throughout the thesis.

The intermediate objective of ADP is to train the critic to approximate the cost-to-go function, so that $\widetilde{J}(\vec{x},\vec{w}) \approx J(\vec{x},\vec{\mathbf{e}},\vec{z})$ for all $\vec{x} \in \mathbb{S}$. If this is achieved perfectly, and if simultaneously the action network always chooses actions according to:

$$\vec{u} = \arg\min_{\vec{u}\in\mathbb{A}} \mathbb{E}\left(U(\vec{x},\vec{u},\vec{e}) + \gamma\widetilde{J}(f(\vec{x},\vec{u},\vec{e}),\vec{w})\right) \quad \forall\vec{x}, \tag{1.7}$$

then Bellman's optimality condition (Bellman, 1957) shows the trajectories produced will be optimal, and the action network is optimal. Bellman's condition is discussed more in Section 2.4 and Chapter 4.

The method of choosing actions purely by Equation (1.7) is called the *model-based greedy policy on* $\widetilde{J}$ since it chooses actions that the critic rates as best. In this thesis, the model-based greedy-policy function will be abbreviated to *greedy policy* function, and denoted as $\pi(\vec{x},\vec{w})$ to distinguish it from an action network $A(\vec{x},\vec{e},\vec{z})$, and also to emphasise that the greedy-policy function has a dependency on $\vec{w}$, the critic weight vector, and not on $\vec{z}$. Using this notation, the greedy-policy function is defined as:

$$\pi(\vec{x},\vec{w}) := \arg\min_{\vec{u}\in\mathbb{A}} \mathbb{E}\left(U(\vec{x},\vec{u},\vec{e}) + \gamma\widetilde{J}(f(\vec{x},\vec{u},\vec{e}),\vec{w})\right). \tag{1.8}$$

Efficient implementations of the greedy policy are described in Chapter 5, for situations where the environment functions are linear-in-$\vec{u}$. In some stochastic situations these efficient methods are approximations to the greedy policy defined in Eq. (1.8).

1.2.1 Stochasticicty in the Environment Functions

The environment functions $f(\vec{x},\vec{u},\vec{e})$, $U(\vec{x},\vec{u},\vec{e})$ and $\Phi(\vec{x},\vec{e})$ are functions of a random variable, $\vec{e}$, and therefore they can be used to model stochastic environments and cost functions. The introduction of the vector $\vec{e}$ follows the method of handling stochastic environments by Jacobson and Mayne (1970) and Werbos (2012). The noise vector $\vec{e}_t$, at each time step t, can be understood to be a collection of random real numbers, which could be unobservable and come from the environment itself (in the case of a real robot moving in a physical environment), or observable and come from a computer's random-number generator (in the case of a simulated environment). The dimensionality of the vector $\vec{e}$ allows room for enough random numbers so that the noise in the function

$f(\vec{x}, \vec{u}, \vec{e})$ can be specified separately from the noise in the function $U(\vec{x}, \vec{u}, \vec{e})$. Similarly some components of $\vec{e}$ will affect $A(\vec{x}, \vec{e}, \vec{z})$, and these components are likely to be all observable, since they would come from a computer's random number generator to enable deliberate exploration.

Under this scheme, the state transition function given by Eq. (1.1) can be used to model any desired probability distribution $P(\vec{x}_{t+1}|\vec{x}_t, \vec{u}_t)$ for taking us to the next state ($\vec{x}_{t+1}$) given the current state and action ($\vec{x}_t$ and $\vec{u}_t$). This generality follows from the same idea that any uniform random number-generator can be used to generate random numbers to match any desired distribution, following the Fundamental Transformation Law of Probabilities (see Press et al., 1992, Sec.7.2). This means that the method of specifying stochastic state transitions used in Eq. (1.1) is just as general as specifying a full probability distribution $P(\vec{x}_{t+1}|\vec{x}_t, \vec{u}_t)$, which is the method usually used in the RL literature.

1.3 Literature Review

This literature review describes the history and origins of ADPRL (in Section 1.3.1), and important modern ADPRL survey articles and applications, in Section 1.3.2. Section 1.3.3 describes the algorithms which are very closely related to the main algorithm of this thesis, VGL(λ), clearly showing their relation to VGL(λ). Section 1.3.4 describes more distantly related ADPRL algorithms. Finally, Section 1.3.5 summarises some of the convergence proofs that exist for ADPRL algorithms.

1.3.1 Origins of ADPRL

The earliest attempts to create RL systems date back to the 1960's when Minsky (1963) proposed a general purpose reinforcement learning system. However it was soon found that these systems could not handle as much complexity as methods based on Dynamic Programming (Bellman, 1957; Howard, 1960a). Samuel (1959, 1967) made an early machine-learning system to play checkers which used a look-ahead system combined with a state-evaluation function which was self-improving with playing experience. This state-evaluation function was similar, but not identical, to the value function used by ADPRL (see Sutton and Barto, 1998, Sec. 11.2, for a discussion).

A proposal to extend RL to use a function approximator to explicitly represent successive representations of Bellman's value function was made by Werbos (1968). Widrow et al. (1974) made a very early critic-learning system, which was designed for playing black-jack. Heuristic Dynamic Programming (HDP) is one of the main modern ADP algorithms. This was originally defined by Werbos (1977), but described in more detail by Werbos (1992b) and Prokhorov and Wunsch (1997a). The other main algorithm of ADP is Dual Heuristic Programming (DHP), which is a value-gradient algorithm. Historically, DHP was first designed in 1977 (Werbos, 1977), but greater detail came in 1981 (Werbos, 1981). This 1981 algorithm was for the "Galerkinized" version of DHP, but this version "converges to the wrong answer almost always for linear dynamical systems with noise" (Werbos, 1998, 2004). It wasn't until 1992 that a published version of the non-galerkin version came (Werbos, 1992b).

Most ADP algorithms are variants on HDP or DHP. Out of these two algorithms, DHP is a true value-gradient learning algorithm. HDP is a value-learning algorithm. Prokhorov and Wunsch (1997a) give a detailed summary of the "design ladder" of ADP algorithms including HDP and DHP.

Key works founding the RL community were by Barto et al. (1983) and Sutton (1988) with a solution of the cart-pole benchmark problem, and the invention of the Temporal Differences learning algorithm, TD(λ).

1.3.2 Contemporary ADPRL Literature and Applications

The ADP field is nicely summarised in the review article by Wang et al. (2009) which defines the main algorithms HDP and DHP, plus variant algorithms based on these, and also describes industrial applications of ADPRL. Other ADP survey articles are available (Balakrishnan et al., 2008; Lendaris, 2009; Lewis and Vrabie, 2009; Lewis et al., 2012; Powell, 2009).

Surveys and introductions to RL are available by Kaelbling et al. (1996) and Sutton and Barto (1998).

In the ADPRL communities, applications have been reported for missile control (Balakrishnan and Biega, 1995; Han and Balakrishnan, 1999, 2002), automotive control (Prokhorov, 2009), aircraft control over a flight envelope (Ferrari and Stengel, 2002), aircraft landing control (Murray et al., 2002; Prokhorov and Wunsch, 1997a; Prokhorov et al., 1995), helicopter reconfiguration after rotor failure (Enns and Si, 2003), power system control (Li et al., 2012; Lu et al., 2005; Venayagamoorthy and Wunsch, 2003), vehicle steering and speed control (Lendaris et al., 2000), computer go (Silver et al., 2012), computer backgammon (Tesauro, 1994), elevator dispatching (Barto and Crites, 1996) and maintaining the inverted flight of a helicopter (Ng et al., 2004).

1.3.3 The Relationship of VGL(λ) to Existing ADPRL Algorithms

HDP is a value-learning ADP design, i.e. it does not consider value gradients. DHP is a VGL algorithm. The way these two algorithms are related to each other is shown by the two algorithms at the top of Fig. 1.1.

$$\begin{tikzcd}
\begin{array}{c}\textbf{HDP is}\\ \Delta\vec{w}=\alpha\sum_t\left(\frac{\partial\widetilde{J}}{\partial\vec{w}}\right)_t\left(J'_t-\widetilde{J}_t\right)\\ \text{with } J'_t \text{ defined by}\\ J'_t:=U_t+\gamma\widetilde{J}_{t+1}\end{array} \arrow[r, "\frac{\partial}{\partial\vec{x}}"] \arrow[d, "\text{insert } \lambda"'] & \begin{array}{c}\textbf{DHP is}\\ \Delta\vec{w}=\alpha\sum_t\left(\frac{\partial\widetilde{G}}{\partial\vec{w}}\right)_t\Omega_t\left(G'_t-\widetilde{G}_t\right)\\ \text{with } G'_t \text{ defined by}\\ G'_t:=\left(\frac{DU}{D\vec{x}}\right)_t+\gamma\left(\frac{Df}{D\vec{x}}\right)_t\widetilde{G}_{t+1}\\ \text{and } \frac{D}{D\vec{x}}:=\frac{\partial}{\partial\vec{x}}+\frac{\partial A}{\partial\vec{x}}\frac{\partial}{\partial\vec{u}}\end{array} \arrow[d, "\text{insert } \lambda"] \\
\begin{array}{c}\textbf{TD}(\lambda)\textbf{ is}\\ \Delta\vec{w}=\alpha\sum_t\left(\frac{\partial\widetilde{J}}{\partial\vec{w}}\right)_t\left(J'_t-\widetilde{J}_t\right)\\ \text{with } J'_t \text{ defined recursively by}\\ J'_t:=U_t+\gamma\left(\lambda J'_{t+1}+(1-\lambda)\widetilde{J}_{t+1}\right)\end{array} \arrow[r, "\frac{\partial}{\partial\vec{x}}"] & \begin{array}{c}\textbf{VGL}(\lambda)\textbf{ is}\\ \Delta\vec{w}=\alpha\sum_t\left(\frac{\partial\widetilde{G}}{\partial\vec{w}}\right)_t\Omega_t\left(G'_t-\widetilde{G}_t\right)\\ \text{with } G'_t \text{ defined recursively by}\\ G'_t:=\left(\frac{DU}{D\vec{x}}\right)_t+\gamma\left(\frac{Df}{D\vec{x}}\right)_t\left(\lambda G'_{t+1}+(1-\lambda)\widetilde{G}_{t+1}\right)\\ \text{and } \frac{D}{D\vec{x}}:=\frac{\partial}{\partial\vec{x}}+\frac{\partial A}{\partial\vec{x}}\frac{\partial}{\partial\vec{u}}\end{array}
\end{tikzcd}$$

Figure 1.1: Relationship of VGL(λ) to other ADPRL algorithms.

In this figure, $\widetilde{J}$ is the output of the critic network $\widetilde{J}(\vec{x},\vec{w})$, with weight vector $\vec{w}$, and $\widetilde{G}$ is the gradient of $\widetilde{J}$ with respect to the state vector $\vec{x}$ of the agent, i.e. $\widetilde{G}$ is the critic's approximation of the value gradient. The $\Delta\vec{w}$ refers to a weight update for the critic neural network, i.e. this shows the main equation of each learning algorithm, and $\alpha > 0$ is a small learning rate parameter. The subscripted t indicates the time step of a trajectory, U_t is the instantaneous cost received by the agent at that time step, A is the output of the action network $A(\vec{x},\vec{e},\vec{z})$, and Ω_t is an arbitrary positive definite square matrix. Fuller details and full pseudocode for all of the figure's algorithms will be given in Chapters 2-3, but at the moment the figure indicates the key relationships between various ADPRL algorithms.

The TD(λ) algorithm (Barto et al., 1983; Sutton, 1988) is an extension of HDP to include a constant $\lambda \in [0, 1]$ parameter which specifies how the critic neural network's output values are to be updated. λ is a blending parameter that specifies how much the critic values are updated towards other critic values (a process known as "bootstrapping") as opposed to towards the true total trajectory cost. The exact details of this blending and the main TD(λ) weight update is shown in the bottom left corner of Fig. 1.1, but again, fuller details will be given in Chapter 2. TD(λ) is a unifying extension between the earlier critic methods, such that TD(1) is equivalent to Widrow et al. (1974)'s black-jack critic and TD(0) is equivalent to HDP.

From Fig. 1.1, the new algorithm presented by this thesis, VGL(λ), can be shown in context to the other three algorithms. VGL(λ) is most easily understood as an extension of DHP that includes a λ parameter analogous to that used in TD(λ). The parameter λ can affect learning speed of the algorithm and convergence stability, and it was the addition of this parameter that led to the convergence proof in Chapter 8, so it is was an important extension to have made to DHP.

Although best understood as an extension to DHP, the VGL(λ) algorithm was actually derived by changing TD(λ) to learn value gradients, i.e. by following the right-pointing arrow at the bottom of Fig. 1.1. It was the failure of TD(λ) to solve continuous-valued state-space control problems without value exploration, and the slow speed of its ability to solve them with value exploration, that led to the creation of VGL(λ) (Fairbank, 2008). Examples of these two kinds of difficulties, and the benefits of VGL methods in solving them, are given in the experiments in the following two chapters (specifically in Figs. 2.2, 3.7 and 3.10). It was the understanding that TD(λ) could be defined in this concise form using the recursive definition of J'_t from Fig. 1.1 that enabled the algorithm VGL(λ) to be defined easily.

1.3.4 Other ADPRL Algorithms

Other variations on the main algorithms shown in Fig. 1.1 are possible. These include Q-learning and the residual-gradient methods, plus Gradient Temporal Differences methods. These are described in Sections 2.6, 2.7.3 and 2.7.4, respectively. From the ADP community, there is "Action Dependent HDP" (ADHDP), which is roughly the same as Q-learning, plus Globalized DHP. These are described in Sections 2.6 and 3.4.4, respectively. Plus there are other ADP variants including Action Dependent DHP (ADDHP), Action Dependent GDHP (ADGDHP) and Galerkinized DHP, which are not described in this thesis, but are described by Prokhorov and Wunsch (1997a) and Werbos (1998).

Another group of ADPRL algorithms are policy-gradient based methods. These do not use a critic at all, but instead do direct gradient descent on $\widehat{J}(\vec{x}, \vec{\mathbf{e}}, \vec{z})$ with respect to the weight vector of the action network, $\vec{z}$. These include the REINFORCE algorithm by Williams (1992) and Backpropagation Through Time ("BPPT") by Werbos (1990a), and are described in detail in Chapter 6. Despite BPTT being a critic-free method, BPTT is an essential component of this thesis because a unification proof between BPTT and the critic-based VGL(λ) algorithm is given in Chapter 8. Since this forms the basis of a convergence proof for one instance of VGL(λ), this link to BPTT makes the most significant theoretical achievement of this thesis.

1.3.5 Convergence Proofs for ADPRL algorithms

Convergence proofs for various RL algorithms exist, but these are mainly for tabular representations of the critic, or linear function approximation of the critic, and/or fixed action-networks. These include proofs by Baird (1995); Dayan (1992); Sutton (1988); Sutton et al. (2000, 2009); Tsitsiklis and Van Roy (1996a). The work of Maei et al. (2009) is valid for a critic with non-linear function approximation, with a fixed action network. These works are all described in detail in Section 2.7.

Convergence proofs for the ADP algorithms include work by Al-Tamimi et al. (2008); Ferrari and Stengel (2004); Heydari and Balakrishnan (2011); Howard (1960b); Prokhorov and Wunsch (1997b) and these are all described in the introduction of Chapter 8.

The critic-free methods, REINFORCE and BPTT, have very good convergence guarantees with non-linear function approximation and changing action-network behaviour, and these convergence results are described further in Chapter 6.

1.4 Model-Free and Model-Based Algorithms

ADPRL algorithms can be split into two kinds: model-free and model-based algorithms.

Model-free algorithms are ones that don't have an explicit requirement to know the environment functions, $f(\vec{x},\vec{u},\vec{e})$, $U(\vec{x},\vec{u},\vec{e})$ or $\Phi(\vec{x},\vec{e})$. These algorithms often do contain instances of the environment functions, but they are included in such a way that they could represent a robot or agent actually exploring the environment. The agent could still make deductions about cost received (i.e. $U(\vec{x},\vec{u},\vec{e})$) or state transition made (i.e. $f(\vec{x},\vec{u},\vec{e})$) by pure observation, and without knowledge of the functions themselves. For example the, trajectory-unroll algorithm (Alg. 1.1) is model-free according to this definition, and so is the TD(λ) weight update, which is defined in Fig. 1.1 and in Chapter 2.

In contrast, a model-based algorithm is one which must explicitly know the model and cost functions. For example, all VGL based algorithms are model based, because they need to explicitly use the derivatives of the model and cost functions, as can be seen from their definitions in Fig. 1.1. These derivatives cannot be deduced by single observations by the agent.

This completes the definition of what distinguishes a model-free algorithm from a model-based one. The following two subsections describe the extent to which model-based and model-free algorithms are used within ADPRL, and which researchers use which.

1.4.1 Model-Based RL

The sub-field of "model-based RL" is different from the simple classification of an algorithm as model-based by the above definition.

Kaelbling et al. (1996) define model-based RL as "Learn a model, and use it to derive a controller." Model-based RL is also known as "planning in RL" or "indirect RL" (Sutton and Barto, 1998, chap. 9).

When a neural network is trained to learn the model and cost functions using an appropriate supervised learning method, the neural network will converge (as closely as possible) to the expectation of the target functions, which will necessarily be deterministic. In this thesis the targets for the deterministic parts of the learned model and cost functions are denoted with overbar symbols. Using this notation, the model and cost functions can be decomposed into a deterministic learned part plus a pure-noise part, so that:

$$\begin{aligned} f(\vec{x},\vec{u},\vec{e}) &\equiv \bar{f}(\vec{x},\vec{u}) + \xi_f(\vec{x},\vec{u},\vec{e}) \\ U(\vec{x},\vec{u},\vec{e}) &\equiv \bar{U}(\vec{x},\vec{u}) + \xi_U(\vec{x},\vec{u},\vec{e}) \\ \Phi(\vec{x},\vec{e}) &\equiv \bar{\Phi}(\vec{x}) + \xi_\Phi(\vec{x},\vec{e}). \end{aligned} \tag{1.9}$$

It is the objective of the model-learning process to achieve, as closely as possible,

$$\begin{aligned} \bar{f}(\vec{x},\vec{u}) &\equiv \mathbb{E}\left(f(\vec{x},\vec{u},\vec{e})\right) \\ \bar{U}(\vec{x},\vec{u}) &\equiv \mathbb{E}\left(U(\vec{x},\vec{u},\vec{e})\right) \\ \bar{\Phi}(\vec{x}) &\equiv \mathbb{E}\left(\Phi(\vec{x},\vec{e})\right). \end{aligned} \tag{1.10}$$

If this objective is met then the expectations of the three noise terms, $\xi_f(\vec{x},\vec{u},\vec{e})$, $\xi_U(\vec{x},\vec{u},\vec{e})$ and $\xi_\Phi(\vec{x},\vec{e})$ should all be zero.

The three functions $\bar{f}(\vec{x},\vec{u})$, $\bar{U}(\vec{x},\vec{u})$ and $\bar{\Phi}(\vec{x})$ are all deterministic functions, which is reflected in their notation, since they are not dependent on the noise vector $\vec{e}$. The overbar symbol indicates that they are each expectations of their underlying functions, i.e. according to Eq. (1.10).

Sutton and Barto (1998) describe an advantage of model-based RL to be that it makes fuller use of experience, that is it obtains a better policy with fewer environment interactions, and a disadvantage of model-based RL to be that if the model is not perfectly learned then the optimization will optimise the wrong model.

Another benefit of model-based RL can be understood by imagining the case of a robotic flying machine that is to be trained only through the negative reinforcement of crashes. In this situation it would clearly be beneficial for these crashes to happen in model-based simulation rather than in reality. This example, when coupled with the fact that many simulated trajectories can usually be evaluated in the time it takes a real robot to complete one physical trajectory, makes a strong incentive to choose model-based methods over model-free ones.

Concerning the disadvantage of using a possibly incorrect learned model, the model-approximation error is just one more error on top of the approximation error in the critic, and the approximation error in the action network; and as we shall see in Chapter 5, when model-based methods are used it is sometimes possible to remove the action network, thus removing one layer of approximation error, and hence neutralising this model-based disadvantage.

Other potential drawbacks of model-based RL are that time must be spent learning the model functions, and also that it is tricky to decide when it is time to stop learning them. The second of these two issues can be addressed by never stopping the model learning, i.e. to continually learn the model functions concurrently with the critic and action-network training; although with a changing approximate model this could make convergence assurance even more difficult. On the issue of the extra time spent training the model functions, I am in agreement with Lendaris and Neidhoefer, who wrote: "*We mention that some view this model dependence to be an unnecessary 'expense'. The position of the authors, however, is that the expense is in many contexts more than compensated for by the additional information available to the learning/optimization process*" (Lendaris and Neidhoefer, 2004, sec 4.3).

One of the best known and celebrated successes in RL relied upon a model-based method: maintaining the inverted flight of a helicopter (Ng et al., 2004). This model-based RL approach first learned the model functions and then afterwards found a control policy.

1.4.2 Model-Given ADPRL

This thesis goes further than the approach of "learn a model, and use it to derive a controller", and generally completely skips the "model learning" phase, by just assuming the model functions are already known or assumed pre-learned. This will be referred to as "model-given ADPRL" to distinguish it from model-based RL. It is also known as "Planning in Reinforcement Learning" (Sutton and Barto, 1998, chap. 9).

The possibility of learning model functions is discussed in Section 3.2, and only a short mention of actually doing it is given in the experiment of Section 3.7; but otherwise model learning is largely omitted from this thesis. This omission creates a tighter focus on the second stage of model-based of RL, which is a very worthwhile stage to study in its own right.

Potential drawbacks of the "model-given" restriction of this thesis are that we are assuming any model functions can be pre-learned, in principle. This might not be the case in certain examples; maybe some model functions are just too difficult to learn. Possibilities of this kind could be the model functions to describe chaotic turbulent airflow around a hovering helicopter, or learning the model function to describe the changing states of the stock market. But in both cases, there are large unknown elements which could be considered as genuinely random, and therefore could be modelled by attaching stochastic components to the learned model functions. Also, in the stock-market situation, it is an open question whether any model-free algorithm could achieve any statistically-significant advantage over a model-based algorithm where the system-identification process had completely failed.

Another drawback of the model-given restriction of this thesis is that the experiments mostly do not account for the extra learning time it would have taken to learn the model functions. The only partial exception is the experiment in Section 3.7 which does mention this issue briefly.

1.5 Related Fields

There is quite a large overlap between the research disciplines of RL, ADP, Control Theory, Neurocontrol and Differential Dynamic Programming. This section aims to describe how they all relate to each other, and therefore provide a better context for where ADPRL and this thesis fit in.

1.5.1 How ADP Relates to RL

The ADP and RL subcommunities work on the same problem, but possibly with slightly different areas of focus. They both have the same problem formulation as defined in Section 1.2, i.e. both are trying to find an action network and critic function which will optimise the cost-to-go function from any point in state space. They both tackle the same difficulties in proving convergence of the critic function under a concurrently-updating action network, while also dealing with the complication that bending the critic in one place will inadvertently cause it to bend in other places. Hence the two fields together are referred to as ADPRL.

The only differences between ADP and RL seem to be that the RL literature often provides a greater focus on model-free approaches and discrete-valued state spaces, but ADP commonly includes model-based (e.g. value-gradient) algorithms and focuses on continuous-valued state spaces. Also, Table 1.1 shows some notational differences that commonly occur between the RL and ADP communities. None of these differences is strict though. For example, successful model-given applications are published in the

RL literature; one example is computer-go using the "TD-Search" algorithm (Silver et al., 2012).

ADP Concept/Notation	Equivalent RL Concept/Notation
Cost-to-go Function: $J(\vec{x}, \vec{\mathbf{e}}, \vec{z})$	Value Function: $V(\vec{x}, \vec{z})$
Critic Function: $\widetilde{J}(\vec{x}, \vec{w})$	Approximate Value Function $\widetilde{V}(\vec{x}, \vec{w})$
Action Network	Policy or Actor
Utility Function (Cost Function): $U(\vec{x}, \vec{u}, \vec{e})$	Instantaneous Reward Function: $r(\vec{x}, \vec{u})$
Action Vector: $\vec{u}$	$\vec{a}$
HDP Algorithm	TD(0) Algorithm
ADHDP Algorithm	Q-Learning
Discount Factor, γ	Discount Factor, γ

Table 1.1: Common Notational Differences (and Similarities) between ADP and RL

1.5.2 Relationship to Control Theory and Neurocontrol

Control theory (Kirk, 2004; Todorov, 2006) shares a lot of the objectives of ADPRL, and a lot of notation from ADP (e.g. shared function names such as J, $U(\vec{x}, \vec{u}, \vec{e})$, $f(\vec{x}, \vec{u}, \vec{e})$). Pontryagin's Minimum Principle (PMP) and dynamic programming are both fundamental principles of control theory, and both of these methods use value functions and are therefore relevant to ADPRL. These two principles are described further in Chapter 4.

PMP is a work-horse of control theory because it is capable of efficiently finding an optimal trajectory from any given start point, or between any two given points, in a deterministic environment. The calculation can be done for each trajectory using numerical methods. This is how its use in control theory differs from its use in ADPRL: Whereas ADPRL spends a lot of computation effort to find an optimal policy function $A(\vec{x}, \vec{e}, \vec{z})$ which is valid over the whole state space $\mathbb{S}$, the computation of optimal trajectories by PMP is done on individual trajectories. Hence in control theory, when a new optimal-trajectory is required, another numerical computation must usually be done all over again.[3]

[3] In certain cases PMP can be used to find an optimal policy function over the whole of state space, but this has only been in cases where that function can be represented by an exact analytical solution (Pontryagin et al., 1962, chap. 1, sec. 5).

The use of a value function in control theory, whether by PMP or Bellman's approach, would normally be used with numerical methods or exact analytical solutions. If a function approximator is used to represent the value function, then the method becomes equivalent to ADP; and when a neural network is used for the policy function in control theory then the method becomes equivalent to "neurocontrol" (Antsaklis, 1990; Balakrishnan and Weil, 1996; Hagan and Demuth, 1999; Werbos, 1992a).

There are also major methods in control theory which do not rely on a value function, and which therefore are unrelated to PMP or Bellman's methods. These include inverse-kinematics control and proportional-integrator ("PI") controllers.

1.5.3 Relationship to Differential Dynamic Programming

Differential Dynamic Programming (DDP) is a method described by Jacobson and Mayne (1970). DDP makes a local quadratic expansion of the value function about each state along a given trajectory. Using these local quadratic functions it is possible to iteratively improve the trajectory until it bends into a locally optimal shape. This makes a strong similarity to ADPRL, and the equations look very similar to those appearing in VGL. But the key difference between DDP and VGL is that the value function learned by DDP is only accurate locally to the single trajectory under consideration. Hence, unlike ADPRL, the objective of DDP is not to find a policy function $A(\vec{x}, \vec{e}, \vec{z})$ or critic function $\widetilde{J}(\vec{x}, \vec{w})$ all over $\vec{x} \in \mathbb{S}$; but instead to make a local function for each, expanded around the current trajectory.

1.6 Contributions of this Thesis

The major contributions of this thesis are to extend the DHP method into VGL(λ), and provide the theoretical results listed in Part II. In particular these are:

1. The trajectory optimality proof in Chapter 7 shows that, in continuous-valued deterministic environments, learning the value gradients along a trajectory with a greedy policy will ensure the trajectory is locally optimal, for any value of the bootstrapping parameter λ. Also the proof includes an extension of Pontryagin's Minimum Principle to provide a sufficient condition for local optimality (whereas in its traditional form it only supplies a necessary condition for local optimality).

This extension is given in Corollary 7.1, which may well turn out to be an original observation.

2. An equivalence proof between BPTT and one instance of VGL(1) is given in Chapter 8. This equivalence doubles up as a convergence proof for that instance of VGL(1) with a greedy policy and a general non-linear function approximator for the critic function.

3. Concrete divergence examples for other ADPRL algorithms with a greedy policy, including TD(1), TD(0), DHP and other instances of VGL(λ) are given in Chapter 9.

Also the practical considerations listed in Part III (Advanced Implementation Details) are original. These include:

1. The need to handle clipping correctly by VGL algorithms (Chapter 10).

2. The efficient calculation of the second-order gradients in neural networks which are necessary for implementing GDHP efficiently (Chapter 11).

Publications resulting from this thesis are listed in Table 1.2.

Publication	Relates to Chapters
Fairbank (2008)	All chapters
Fairbank et al. (2012b)	Chapters 3, 5, 8
Fairbank et al. (2013)	Chapters 8, 9
Fairbank and Alonso (2011b)	Chapters 3, 7, 8
Fairbank and Alonso (2012b)	Chapter 9
Fairbank and Alonso (2012c)	Chapters 3, 5
Fairbank and Alonso (2012a)	Chapter 4
Fairbank et al. (2012a)	Chapter 11
Fairbank (2013); Fairbank et al. (2014b)	Chapter 10

Table 1.2: Publications Resulting from this Thesis

This thesis is also relevant to, and contributes to, the model-free RL research field. This, and some minor theoretical contributions of the thesis, are listed in the following two subsections:

1.6.1 Contributions to Model-Free RL

There are several aspects of this work which are relevant to model-free algorithms and VL algorithms such as TD(λ). In Chapter 9, a value-gradient analysis is used to derive divergence examples for several major ADPRL algorithms when combined with a greedy policy. One of these divergences is for TD(1). TD(1) is considered to be one of the most reliably convergent algorithms, and it is model-free, but its convergence was still previously only proven for a fixed action network. It is through this value-gradient analysis that the divergence example was found for TD(1), when combined with a greedy policy.

Also, the optimality proof of Chapter 7 states a condition that is necessary (and usually sufficient) for trajectory optimality which affects VL algorithms just as much as it does affect VGL algorithms. Even though VL algorithms do not explicitly learn the value gradient, it still must be learned in these situations if optimality is to be produced.

Finally, the VGL algorithms developed in this thesis may turn out to be relevant to model-free researchers, because it might be possible in future research to develop stochastic model-free approximations to the model-based VGL algorithm instance which is proven to converge.

1.6.2 Minor Theoretical Contributions

The following minor theoretical contributions are made throughout the thesis:

1. The trajectory-shorthand notation described in Section 2.1.2 defines a consistent and clear notation. This makes expressions such as U_t automatically mean $U(\vec{x}_t, \vec{u}_t, \vec{e}_t)$, i.e. the transition cost from leaving state $\vec{x}_t$, as opposed to arriving there. Also, expressions like $\left(\frac{\partial f}{\partial \vec{x}}\right)_t$ are unambiguous compared to expressions from the prior ADP literature, such as $\frac{\partial \vec{x}_{t+1}}{\partial \vec{x}_t}$ or $\frac{\partial \widetilde{J}_{t+1}}{\partial \vec{x}_t}$.

2. The target-value recursions of Eqs. (2.3) and (3.7) are thought to be original to early technical reports that led to this thesis (Fairbank, 2002, 2008). Using these definitions, the TD(λ) and VGL(λ) weight updates can be viewed as single equations. This helps show the relationship between the two algorithms, and also view clearly what the algorithms do, as opposed to how they are programmed.

Eq. (2.3) is now a fairly well-known recursion, e.g. appearing as eq. 8 of reference (Maei and Sutton, 2010).

3. It is deliberate that none of the ADPRL algorithms defined in this thesis are done so by claiming them to be gradient-descent on a given error function. Section 2.7.1 discusses this point further.

4. The emphasis that "local value exploration comes for free" with VGL compared to VL methods is an original explanation and motivation for VGL methods. This is described in Section 3.1, plus other sections and experiments.

5. The splitting of Bellman's principle into two parts (as described in Section 4.1.1) is useful for motivating the importance of considering a greedy policy, even when a greedy policy is never used.

6. The visual explanation of how Bellman's principle relates to Pontryagin's minimum principle (Section 4.4.1).

7. The pseudocode simplifies and clarifies many ADPRL algorithms' specifications. This makes various subtleties explicit; such as how to handle terminal states, or how the multiple applications of backpropagation are to be arranged.

1.6.3 Contributions to Stochastic ADP Algorithms

For stochastic environments, several clarifications have been introduced in this thesis. These clarifications are relevant to BPTT and VGL based methods.

1. An explanation is given, in Section 6.1.2, showing that the optimum obtained by BPTT in stochastic environments can be different from the true ADPRL objective. To counter this deficiency, the thesis also gives useful smoothness conditions which are capable of guaranteeing that these two optima become exactly equal to each other (see Section 6.1.2).

2. An explanation is given of why additive noise is relevant to VGL algorithms when environments are stochastic. When non-additive noise is present, the VGL optimality condition and weight update can become approximations (Sections 3.2.1 and 3.5.4). Also, with non-additive noise, the BPTT weight-update can

become an approximation to true gradient descent (Section 6.1.1). In both of these cases, when additive noise is present, the approximations change into exact equalities.

3. A proof of how VGL methods can lead to a globally optimal control policy is provided in Section 4.5, including an extension to stochastic environments (Section 4.A).

4. For stochastic environments, a discussion is given of how additive noise will ensure that a commonly used action-network weight update will be exactly correct, and also that the efficient "greedy-on-$\widetilde{Q}$" policies will become exactly correct under additive noise (Sections 5-5.1 and 5.A).

1.7 Outline of Thesis

This prelude has introduced the ADP problem formal definition, and described the concept of a critic function and the cost-to-go function. The rest of this thesis develops the value-gradient idea. The thesis is split into three main parts:

- Part I defines the terminology and algorithms, including the main existing algorithms and their motivations, plus the VGL(λ) algorithm. This part also describes Pontryagin's and Bellman's optimality principles, and also efficient greedy-policy implementation.

- Part II gives the main theoretical results of this thesis.

- Part III describes the more specific implementation details that can contribute to making a successful VGL implementation.

All main chapters have their own chapter-conclusions section. Chapter 12 summarises conclusions for the whole thesis.

Chapter 2

Value-Learning Algorithms

The purpose of this chapter is to give background information on the existing value-learning (VL) algorithms, and to provide a context for defining VGL methods in the next chapter. The algorithms considered in this chapter include TD(0) and HDP, both of which are described in Section 2.2, and TD(λ) which is described in Section 2.3. The chapter describes how VL algorithms relate to Bellman's optimality condition (Section 2.4). The difference between off-policy and on-policy algorithms is described in Section 2.5, and the off-policy algorithms Q-learning and Action-Dependent HDP are described in Section 2.6.

This chapter also describes convergence and divergence results for VL algorithms in Section 2.7, and concludes with an empirical case-study in Section 2.8 which demonstrates the difficulties VL methods have in the absence of stochastic value exploration, and this provides the motivation for developing VGL algorithms in the next chapter (Chapter 3).

The algorithms presented in this chapter are valid for discrete and continuous-valued state spaces, discrete and continuous-valued action spaces, non-differentiable or differentiable model functions, and they work without knowledge of the model functions and in general stochastic environments. This is in contrast to the VGL algorithms presented in the rest of this thesis which require known differentiable model functions, continuous-valued state spaces, and for which some of the theoretical results presented are only proven for additive-noise or deterministic environments.

2.1 Preliminary Notation

To define the algorithms in this chapter and thesis, it is convenient to define some fundamental notation.

2.1.1 Column-Vector Notation

A convention is used that all defined vector quantities are columns, whether they are coordinates, or derivatives of a scalar function respect to coordinates. For example, $\vec{w}$, $\vec{x}_t$ and $\frac{\partial \widetilde{J}}{\partial \vec{x}}$ are all column vectors.

2.1.2 Trajectory-Shorthand Notation

Subscripted "t" indices after variable or function names are what we will call *trajectory-shorthand* notation. These refer to the time step of a trajectory and provide corresponding arguments $\vec{x}_t$, $\vec{u}_t$ and $\vec{e}_t$ where appropriate; so that for example $\widetilde{J}_{t+1} := \widetilde{J}(\vec{x}_{t+1}, \vec{w})$, $U_t := U(\vec{x}_t, \vec{u}_t, \vec{e}_t)$, and $\left(\frac{\partial \widetilde{J}}{\partial \vec{w}}\right)_t$ is shorthand for the column-vector function $\frac{\partial \widetilde{J}(\vec{x},\vec{w})}{\partial \vec{w}}$ evaluated at $(\vec{x}_t, \vec{w})$.

Trajectory-shorthand notation is used in almost every chapter of this thesis.

2.2 The TD(0) / HDP Algorithms

The TD(0) algorithm is an RL algorithm defined by Sutton (1988). It is equivalent to the ADP algorithm, Heuristic Dynamic Programming (HDP).

Using the notation of Section 2.1, the TD(0)/HDP algorithms, applied in batch mode to a whole trajectory, can be defined succinctly by the following weight update:

$$\Delta \vec{w} = \alpha \sum_t \left(\frac{\partial \widetilde{J}}{\partial \vec{w}}\right)_t (J'_t - \widetilde{J}_t) \tag{2.1}$$

where $\alpha > 0$ is the learning rate, and J' is the "target value", defined by

$$J'_t := \begin{cases} U_t + \gamma \widetilde{J}_{t+1} & \text{if } \vec{x}_t \notin \mathbb{T} \\ \Phi_t & \text{if } \vec{x}_t \in \mathbb{T}. \end{cases} \tag{2.2}$$

Pseudocode for the TD(0)/HDP algorithms is given in Algorithm 2.1.

Algorithm 2.1 Actor-Critic, On-Line Implementation of TD(0)/HDP Algorithms, including an ADP action-network weight update.

```
1: $t \leftarrow 0$
2: while $\vec{x}_t \notin \mathbb{T}$ do
3:     $\vec{u}_t \leftarrow A(\vec{x}_t, \vec{e}_t, \vec{z})$
4:     $\vec{x}_{t+1} \leftarrow f(\vec{x}_t, \vec{u}_t, \vec{e}_t)$
5:     Calculate ${J'}_t$ by Eq. (2.2).
6:     $\vec{w} \leftarrow \vec{w} + \alpha \left(\frac{\partial \widetilde{J}}{\partial \vec{w}}\right)_t \left({J'}_t - \widetilde{J}_t\right)$
7:     $\vec{z} \leftarrow \vec{z} - \beta \left(\frac{\partial A}{\partial \vec{z}}\right)_t \left(\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \left(\frac{\partial \widetilde{J}}{\partial \vec{x}}\right)_{t+1}\right)$
8:     $t \leftarrow t + 1$
9: end while
10: $\vec{w} \leftarrow \vec{w} + \alpha \left(\frac{\partial \widetilde{J}}{\partial \vec{w}}\right)_t \left(\Phi(\vec{x}_t, \vec{e}_t) - \widetilde{J}_t\right)$
```

The pseudocode includes a critic weight update and an action-network (actor) weight update. Hence this combination is referred to as an *actor-critic* algorithm. Line 7 of the algorithm implements the action-network weight update, with learning rate $\beta > 0$. This is a model-based weight update which is commonly used in the ADP literature (for example, see Prokhorov and Wunsch, 1997a, Eq. 10). There are several other valid choices of action-network weight update that could be used in place of this line, including some model-free ones, described in Chapter 5. There are also different possible arrangements for interleaving the critic and actor weight updates, as discussed in Section 5.1.1. If the learning rate for the action network is set to zero, i.e. if $\beta = 0$, then this pseudocode would reduce down to the pure TD(0)/HDP critic-only weight update.

2.3 The TD(λ) Algorithm

This is an extension of TD(0) made by Sutton (1988), where a constant parameter $\lambda \in [0, 1]$ is introduced. The main idea can be described mathematically by changing the definition of the target value, ${J'}_t$, from Eq. (2.2) into

$$
{J'}_t := \begin{cases} U_t + \gamma \left(\lambda {J'}_{t+1} + (1-\lambda)\widetilde{J}_{t+1}\right) & \text{if } \vec{x}_t \notin \mathbb{T} \\ \Phi_t & \text{if } \vec{x}_t \in \mathbb{T}. \end{cases} \tag{2.3}
$$

Convergence of this recursion is ensured by requiring that either $\gamma < 1$, $\lambda < 1$, or the environment is such that the agent is guaranteed to reach a terminal state at some finite time (i.e. the environment is episodic).

With the above modified definition of J'_t (by Eq. (2.3)), Eq. (2.1) defines the TD(λ) algorithm. This redefined J'_t formula and definition of TD(λ) is represented by the left-hand vertical arrow labelled "insert λ" in Fig. 1.1.

In a stochastic environment, J'_t will depend on the environment noise and random actions taken after time step t, hence the observed value of J'_t is a random sample.

Eq. (2.3) specifies a recursion for J'_t that unrolls forwards in time. Using this forwards-in-time evaluation, it is necessary to wait until the trajectory is completed before any value of J'_t can be known. However it is also possible to accumulate the recursion backwards in time, using an eligibility-trace method derived by Sutton (1988). Pseudocode to do this is given in Alg. 2.2. This method of evaluating the weight update is equivalent to that described by Eqs. (2.3) and (2.1), but the eligibility trace method is more convenient because it means the TD(λ) weight update can be applied continually, i.e. without having to wait for the trajectory to be completed. TD(λ) is therefore an on-line weight update.

Algorithm 2.2 Actor Critic, On-Line Implementation of TD(λ) using Eligibility Traces, including an Action-Network Weight Update from ADP.

```
1: $\vec{e} \leftarrow \vec{0}$ {"Eligibility trace" vector}
2: $t \leftarrow 0$
3: while $\vec{x}_t \notin \mathbb{T}$ do
4:     $\vec{u}_t \leftarrow A(\vec{x}_t, \vec{e}_t, \vec{z})$
5:     $\vec{x}_{t+1} \leftarrow f(\vec{x}_t, \vec{u}_t, \vec{e}_t)$
6:     $\delta \leftarrow U_t + \gamma \widetilde{J}_{t+1} - \widetilde{J}_t$
7:     $\vec{e} \leftarrow \lambda\gamma\vec{e} + \left(\frac{\partial \widetilde{J}}{\partial \vec{w}}\right)_t$
8:     $\vec{w} \leftarrow \vec{w} + \alpha\vec{e}\delta$ {Critic update}
9:     $\vec{z} \leftarrow \vec{z} - \beta\left(\frac{\partial A}{\partial \vec{z}}\right)_t \left(\left(\frac{\partial U}{\partial \vec{u}}\right)_t + \gamma\left(\frac{\partial f}{\partial \vec{u}}\right)_t \left(\frac{\partial \widetilde{J}}{\partial \vec{x}}\right)_{t+1}\right)$ {Action-network update}
10:    $t \leftarrow t + 1$
11: end while
12: $\delta \leftarrow \Phi(\vec{x}_t, \vec{e}_t) - \widetilde{J}_t$
13: $\vec{e} \leftarrow \lambda\gamma\vec{e} + \left(\frac{\partial \widetilde{J}}{\partial \vec{w}}\right)_t$
14: $\vec{w} \leftarrow \vec{w} + \alpha\vec{e}\delta$ {Final critic update}
```

To make the pseudocode applicable to the control problems used later in this chapter, the pseudocode presented here also includes an action-network weight update, i.e. it is presented in actor-critic form. The true TD(λ) critic weight update can be recovered by setting $\beta = 0$.

From the recursion in Eq. (2.3), the effect of λ can be seen to specify a blending of how much the target ${J'}_t$ is to be based upon the critic's estimation of the future state(s), and how much the target ${J'}_t$ is to be based upon the actual future total trajectory cost. When $\lambda = 0$, the above recursion simplifies down to the TD(0) target, i.e. Eq. (2.2). In the other extreme, when $\lambda = 1$, the above recursion simplifies down to ${J'}_t = \widehat{J}_t$, i.e. in this extreme ${J'}_t$ equals the sampled cost-to-go of the trajectory. When λ takes intermediate values between 0 and 1, the target ${J'}_t$ takes an appropriate blending between the above two extremes.

Consequently the parameter λ specifies the degree to which an estimate (the critic) is updated towards itself (also an estimate); a process Sutton and Barto (1998) liken to trying to pull yourself up by your own bootstraps. Hence we will refer to λ as the bootstrapping parameter.

The definition of ${J'}_t$ is equivalent to the "λ-Return" Watkins (1989), as proven in the chapter appendix (Section 2.A.1). The ${J'}_t$ recursive formula of Eq. (2.3) is more succinct than the form by Watkins. The use of J' greatly simplifies the analysis of value functions, and value gradients.

Both TD(λ) and HDP aim to attain, as closely as possible, $\widetilde{J}_t = \mathbb{E}\left({J'}_t\right)$ for all t along a trajectory, and eventually all over $\mathbb{S}$. This is why the values ${J'}_t$ are called "targets". However since ${J'}_t$ is dependent on all future actions, and on $\widetilde{J}(\vec{x}, \vec{w})$ if $\lambda < 1$, then the targets are moving ones. Consequently, it is not a simple matter to attain $\widetilde{J}_t = \mathbb{E}\left({J'}_t\right)$ for all t; it can happen that the $\widetilde{J}_t$ values never quite catch up with the $\mathbb{E}\left({J'}_t\right)$ values. As discussed further in Section 2.7, convergence of TD(λ) is not generally guaranteed, even when an infinitesimally small learning rate is used.

Furthermore, even if the targets were stationary, due to the limitations of function approximation it will be unlikely for the critic network to be ever able to attain $\widetilde{J}_t = \mathbb{E}\left({J'}_t\right)$ for all t, exactly. However due to the universal function-approximation capabilities of neural networks, a multi-layer perceptron with sufficient number of hidden nodes can theoretically get arbitrarily close to this objective.

2.3.1 Motivations for Introducing a λ Parameter

The motivations for including the λ parameter are as follows:

1. Having a high value of λ can make the definition of J'_t do a long look-ahead into the total future true trajectory cost, and hence this could speed up learning.

2. Setting $\lambda = 1$ can turn TD(λ) into true gradient descent (as we will see in Section 2.7.1), which can improve TD(λ) convergence assurance.

3. There is also a benefit to having a low value of λ: In environments which are heavily stochastic, or even an environment which is deterministic but chaotic, the long-term true trajectory cost can have a high variance, which makes the quantity J'_t fluctuate around wildly and therefore is difficult to learn. By reducing λ, we can shorten the look-ahead and hence reduce the variance of J'_t, and this can make learning easier. Consequently, in experiments shown by Sutton and Barto (1998), an intermediate value of $\lambda \in [0, 1]$ often produces the most effective learning.

2.4 The Relationship of the Value-Learning Objective to Bellman's Optimal Principle

The TD(λ) and HDP algorithms aim to satisfy Bellman's optimality principle, which is described here. By satisfying Bellman's optimality conditions, the TD(λ) and HDP algorithms can discover an optimal policy function and thus solve the ADPRL problem.

The TD(λ) weight update has a fixed point when the expectation of the right-hand side of Eq. (2.1) is zero. This would occur if $\widetilde{J}_t = \mathbb{E}(J'_t)$ for all $\vec{x}_t \in \mathbb{S}$. As noted in the previous subsection, in general it will be impossible for a function approximator to attain this goal exactly, but it is possible for a neural network with universal function approximation capabilities to get arbitrarily close to it. Hence this fixed point is referred to as the "target" of neural-network training.

The reason this fixed point is desirable is because it leads to Bellman's Optimality Condition, via the following theorem:

Theorem 2.1. *When actions $\vec{u}_t$ are generated by an action network, $A(\vec{x}_t, \vec{e}_t, \vec{z})$, and $\lambda \in [0, 1]$ is a fixed constant; if $\widetilde{J}_t = \mathbb{E}(J'_t)$ for all $\vec{x}_t \in \mathbb{S}$, then for all $\vec{x}_t \in \mathbb{S}$, we have*

$$\widetilde{J}(\vec{x}_t, \vec{w}) = \begin{cases} \mathbb{E}\left(U(\vec{x}_t, \vec{u}_t, \vec{e}_t) + \gamma \widetilde{J}(f(\vec{x}_t, \vec{u}_t, \vec{e}_t), \vec{w})\right) & \text{if } \vec{x}_t \notin \mathbb{T} \\ \mathbb{E}\left(\Phi(\vec{x}_t, \vec{e}_t)\right) & \text{if } \vec{x}_t \in \mathbb{T}. \end{cases} \tag{2.4}$$

Proof. First note that,

$$\begin{aligned} \widetilde{J}_t &= \mathbb{E}\left(J'_t\right) \\ \Rightarrow \mathbb{E}\left(J'_t - \widetilde{J}_t\right) &= 0 \end{aligned} \tag{2.5}$$

Next, for non-terminal states, we have

$$\begin{aligned} \widetilde{J}_t &= \mathbb{E}\left(J'_t\right) && \text{(by assumption of the theorem)} \\ &= \mathbb{E}\left(U_t + \gamma\left(\lambda J'_{t+1} + (1-\lambda)\widetilde{J}_{t+1}\right)\right) && \text{(by Eq. (2.3); } \vec{x}_t \notin \mathbb{T}\text{)} \\ &= \mathbb{E}\left(U_t + \gamma\widetilde{J}_{t+1} + \gamma\lambda\left(J'_{t+1} - \widetilde{J}_{t+1}\right)\right) \\ &= \mathbb{E}\left(U_t + \gamma\widetilde{J}_{t+1}\right) + \gamma\lambda\mathbb{E}\left(J'_{t+1} - \widetilde{J}_{t+1}\right) \\ &= \mathbb{E}\left(U_t + \gamma\widetilde{J}_{t+1}\right) && \text{(by Eq. (2.5)).} \end{aligned} \tag{2.6}$$

Next considering the terminal states only, we have

$$\begin{aligned} \widetilde{J}_t &= \mathbb{E}\left(J'_t\right) && \text{(by assumption)} \\ &= \mathbb{E}\left(\Phi(\vec{x}_t, \vec{e}_t)\right) && \text{(by Eq. (2.3); } \vec{x}_t \in \mathbb{T}\text{).} \end{aligned} \tag{2.7}$$

Hence combining Equations (2.6) and (2.7) for both terminal and non-terminal states, and removing trajectory-shorthand notation gives

$$\widetilde{J}(\vec{x}_t, \vec{w}) = \begin{cases} \mathbb{E}\left(U(\vec{x}_t, \vec{u}_t, \vec{e}_t) + \gamma\widetilde{J}(f(\vec{x}_t, \vec{u}_t, \vec{e}_t), \vec{w})\right) & \text{if } \vec{x}_t \notin \mathbb{T} \\ \mathbb{E}\left(\Phi(\vec{x}_t, \vec{e}_t)\right) & \text{if } \vec{x}_t \in \mathbb{T}. \end{cases}$$

Since the above equation was derived for an arbitrary state $\vec{x}_t$ in $\mathbb{S}$, the theorem holds for all $\vec{x}_t \in \mathbb{S}$. □

If in addition to the fixed point given above, we also have the action network obeying the greedy policy given by Eq. (1.7), i.e.:

$$\vec{u}_t = \arg\min_{\vec{u}_t \in \mathbb{A}} \mathbb{E}\left(U(\vec{x}_t, \vec{u}_t, \vec{e}_t) + \gamma\widetilde{J}(f(\vec{x}_t, \vec{u}_t, \vec{e}_t), \vec{w})\right) \quad \forall \vec{x}_t \in \mathbb{S}, \tag{2.8}$$

then, combining equations (2.4) and (2.8), and renaming $\vec{x}_t$ to $\vec{x}$, implies that we have achieved, for all $\vec{x} \in \mathbb{S}$:

$$\widetilde{J}(\vec{x}, \vec{w}) = \begin{cases} \min_{\vec{u} \in \mathbb{A}} \mathbb{E}\left(U(\vec{x}, \vec{u}, \vec{e}) + \gamma \widetilde{J}(f(\vec{x}, \vec{u}, \vec{e}), \vec{w})\right) & \text{if } \vec{x} \notin \mathbb{T} \\ \mathbb{E}\left(\Phi(\vec{x}, \vec{e})\right) & \text{if } \vec{x} \in \mathbb{T}. \end{cases}$$

This is equivalent to Bellman's optimal value function $J^*(\vec{x})$, which is defined by the recursion:

$$J^*(\vec{x}) = \begin{cases} \min_{\vec{u} \in \mathbb{A}} \mathbb{E}\left(U(\vec{x}, \vec{u}, \vec{e}) + \gamma J^*(f(\vec{x}, \vec{u}, \vec{e}))\right) & \text{if } \vec{x} \notin \mathbb{T} \\ \mathbb{E}\left(\Phi(\vec{x}, \vec{e})\right) & \text{if } \vec{x} \in \mathbb{T}. \end{cases} \tag{2.9}$$

This equation (Eq. (2.9)) is also known as the Bellman Equation. Bellman (1957) proves that if there exists a function $J^*(\vec{x})$ which satisfies the Bellman Equation for all $\vec{x} \in \mathbb{S}$, then $J^*(\vec{x})$ is the *optimal value function*, and the greedy policy on this value function will be an *optimal policy*. This is called Bellman's optimality principle.

It has been shown that if the TD(λ) fixed point, given by $\widetilde{J}_t = \mathbb{E}\left(J'_t\right)$ for all $\vec{x}_t \in \mathbb{S}$, is attained exactly, and if the action network satisfies the greedy-policy equation exactly (Eq. (1.8)), then Bellman's optimality principle will be satisfied, and the action network will therefore produce optimal actions for all $\vec{x} \in \mathbb{S}$.

This demonstration motivates why both TD(λ) and HDP have the goal of achieving as closely as possible, $\widetilde{J}_t = \mathbb{E}\left(J'_t\right)$ for all $\vec{x}_t \in \mathbb{S}$. If this goal was to be attained exactly, and also if the action-network was trained to satisfy the greedy policy equation exactly (Eq. (1.8)), then this would result in optimal behaviour. In practice, due to the finite capabilities of function approximation, it is unlikely this goal will be met exactly; but nevertheless the goal provides the motivation and target for the TD(λ) and HDP weight updates. The fact that these goals won't be quite met exactly is consistent with one expansion of the acronym ADP to mean *approximate* dynamic programming.

2.5 On-Policy and Off-Policy Weight Updates

Value-learning algorithms are often categorised into either "on-policy" methods, or "off-policy" methods. Any critic-learning algorithm aims to train the critic to represent a cost-to-go function, $J(\vec{x}, \vec{\mathbf{e}}, \vec{z})$. The $\vec{z}$ argument here refers to the fact that the cost-to-go function applies to a given action network, $A(\vec{x}, \vec{e}, \vec{z})$. While doing the learning, the agent is exploring and learning about states in $\mathbb{S}$, so the agent is following a policy function (action network) of its own. If the policy function that the agent is following is equal to the action network used in $J(\vec{x}, \vec{\mathbf{e}}, \vec{z})$, then the learning algorithm is said to be "on-policy". Alternatively, it is possible for the agent to follow one policy (e.g. $A(\vec{x}, \vec{e}, \vec{z}_1)$) while learning the cost-to-go function of another policy (e.g. $J(\vec{x}, \vec{\mathbf{e}}, \vec{z}_2)$), which is known as "off-policy" learning.

Off-policy learning is useful because Bellman's optimality principle requires that the policy becomes greedy eventually (as described already in Section 2.4, and discussed in more detail in Section 4.1.1). So the critic must learn values corresponding to the greedy policy, eventually. However RL algorithms, and particularly value-learning algorithms, need to do value exploration too, in order to satisfy Bellman's Condition over the whole state space. So it is helpful to be able to make exploratory actions while still learning about a greedy policy, and off-policy learning achieves this goal.

As we will see in the next chapter, VGL based methods have much less need to perform value exploration, so off-policy algorithms are much less important in VGL than in VL.

One off-policy learning algorithm is called "off-policy Monte-Carlo learning" (as described by Sutton and Barto, 1998, sec 5.6). Another is Q-learning, which is described in the following section.

2.6 Q-Learning and ADHDP

Q-learning is a value-learning algorithm created by Watkins (1989). It has an equivalent algorithm called "Action Dependent HDP", or ADHDP, independently created by Werbos (1989).

Q-learning has two important features:

1. It is an "off-policy" algorithm. That is, it can evaluate one policy (the greedy policy in this case) while following another policy.

2. It has no requirement for knowing the model and cost functions of the environment at all. It simply learns a function $Q(\vec{x}, \vec{u}, \vec{w})$ at every point of $\mathbb{S} \times \mathbb{A}$.

Because this thesis is concerned with large continuous-valued state spaces, only a version of Q-learning with function-approximation will be described, and hence the function $Q(\vec{x}, \vec{u}, \vec{w})$ will be represented by a neural network with weight vector $\vec{w}$.[4] The simplest form of Q-learning can then be written as the following weight update:

$$\Delta\vec{w} = \alpha \sum_t \left(\frac{\partial Q}{\partial \vec{w}}\right)_t (Q'_t - Q_t)$$

where the "target" Q function, Q' is defined by:

$$Q'_t := \begin{cases} U_t + \gamma \min_{\vec{u}\in\mathbb{A}} \left(Q(\vec{x}_{t+1}, \vec{u}, \vec{w})\right) & \text{if } \vec{x}_t \notin \mathbb{T} \\ \Phi(\vec{x}_t, \vec{e}_t) & \text{if } \vec{x}_t \in \mathbb{T}. \end{cases}$$

This weight update and target is clearly very similar to the TD(0)/HDP weight update (Eqs. (2.1) and (2.2)). The difference is in the min operator embedded into the definition of Q'. This is equivalent to a greedy-policy calculation, so it means that Q-learning learns values of the greedy policy, no matter what policy is actually followed. This is what makes it an "off policy" algorithm, and it means Q-learning can be simultaneously exploring the state space while also learning about the greedy policy.

Furthermore, once a $Q(\vec{x}, \vec{u}, \vec{w})$ function is learned, there is no need to use an action network to make control decisions - it is possible to simply choose actions as the $\vec{u}$ argument which minimises the function $Q(\vec{x}, \vec{u}, \vec{w})$ at the given state $\vec{x}$. And unlike direct use of a greedy policy by Eq. (1.7), this minimisation of the Q function is entirely model-free.

When a bootstrapping parameter, λ, is introduced into Q-learning, to produce the algorithm Q(λ), difficulties occur in that whenever an exploratory action is taken. Learning histories (in the form of eligibility traces) have to be truncated, as discussed by (Sutton and Barto, 1998, sec. 7.6).

Werbos (2004) writes about the pros and cons of having to learn a Q-function: "*With Q learning, you only need to know the Q function. You escape the need for a model. On the other hand, if x and u are actually continuous variables, and if you are using a lookup table approximation to Q, then the curse of dimensionality here is even worse than with conventional DP.*" However it would be hoped that a good function approximator could much-lessen the curse of dimensionality problem cited here.

[4] Technically, this means we are actually describing ADHDP not Q-learning, since the only actual difference between Q-learning and ADHDP is that Q-learning was originally defined for tabular representations of the Q-function, but ADHDP was defined with a neural-network representation.

2.7 Convergence Results for VL Methods

TD(0), and its generalization TD(λ), were originally proven to converge for a tabular representation of the value function by Sutton (1988) and Dayan (1992). However for the situation considered in this thesis, the state space is continuous-valued and therefore a tabular value function is not possible. Consequently, the value function (critic) needs representing by a function approximator. In the most general situation of the function approximator being non-linear (such as with the critic neural networks considered in this thesis), there is no convergence proof for the original TD(λ) algorithm, for any instance where $\lambda < 1$. This is largely because TD(λ) is not generally gradient descent on any analytical function (as described in Section 2.7.1). Certain cases, however, are proven to converge for linear critic functions (Section 2.7.2).

Recently created modifications to TD(λ), for example the algorithm named GTD, do have proven convergence, as described in Section 2.7.4. These newer algorithms solve these previous convergence issues, which have represented a challenge to the ADPRL community for a decade or more, for the case where the policy is fixed.

2.7.1 Gradient Descent Issues for TD(λ)

In the special case of $\lambda = 1$ and a fixed action network, TD(λ) is gradient descent on the following error function:

$$E(\vec{x}_0, \vec{w}) = \frac{1}{2} \sum_t \left({J'}_t - \widetilde{J}_t \right)^2 . \tag{2.10}$$

This fact can be verified by noticing that $-\alpha \frac{\partial E}{\partial \vec{w}}$ for the above error function gives the TD(λ) weight update (Eq. (2.1)), provided $\lambda = 1$. Since it is gradient descent on $E \geq 0$, TD(1) will be guaranteed to converge; assuming the action network is fixed, and assuming smoothness on the error surface described by E in $\vec{w}$-space, and assuming a sufficiently small learning rate.

However, the TD(λ) weight update is only gradient descent on the above error function for TD(1), and not for other values of λ. This is because when $\lambda = 1$, the targets given by Eq. (2.3) simplify into $J' = \sum_t \gamma^t U_t$. These targets are therefore fixed (in their expectations; assuming the action network is fixed). However when $\lambda < 1$, the targets are themselves dependent on $\widetilde{J}(\vec{x}, \vec{w})$ through the recursion of Eq. (2.3), and therefore each ${J'}_t$ is a function of $\vec{w}$. This means for $\lambda < 1$ the targets ${J'}_t$ are no longer fixed and therefore TD(λ) is not gradient descent on the above error function.

Furthermore, it was proven by Barnard (1993) that TD(0) is not gradient descent on any error function, for if Eq. (2.1) was such that $\Delta\vec{w} = -\alpha\frac{\partial E}{\partial \vec{w}}$ for some analytic function $E(\vec{w})$, then the second-derivative matrix $\frac{\partial^2 E}{\partial w^i \partial w^j}$ should be symmetric. Since differentiation of Eq. (2.1) again by $\vec{w}$ does not yield a symmetric matrix, therefore TD(0) cannot be gradient descent on any analytic function.

This lack of gradient-descent property of TD(λ) makes proving convergence very difficult. In the weight update, the ${J'}_t$ values are moving targets, and the $\widetilde{J}_t$ values are the values which are chasing the moving targets. There is no guarantee this chase will ever result in convergence.

When a function approximator is used for the critic, the TD(λ) and HDP weight updates necessarily include a gradient term of the form $\frac{\partial \widetilde{J}}{\partial \vec{w}}$, for example as appearing in Eq. (2.1). This has encouraged HDP and TD(λ), with function approximation, to be referred to incorrectly as "gradient descent" methods. Many ADPRL papers do this, and research into proving convergence of TD-based methods seems to have been held back by this confusing terminology. For example, papers often jump from the proven-convergence in the tabular case, to a supposed justification of a corresponding proven-convergence in the function-approximation case.

2.7.2 On-Policy Linear TD(λ) Convergence

In the case that $\widetilde{J}(\vec{x}, \vec{w})$ is linear in $\vec{w}$, and when the agent is forced to always follow actions that the action network specifies during learning (i.e. the learning is "on policy"), then convergence for TD(0) is proven (Tsitsiklis and Van Roy, 1996a). However, if $\widetilde{J}(\vec{x}, \vec{w})$ is non-linear in $\vec{w}$, then divergence can occur (Tsitsiklis and Van Roy, 1996a). For this reason, many research papers use a linear-in-$\vec{w}$ critic function of the form $\widetilde{J}(\vec{x}, \vec{w}) := \vec{w}^T h(\vec{x})$, where $h(\vec{x}) \in \mathbb{R}^{\dim(\vec{w})}$ is a basis function of $\vec{x}$ (i.e. $h(\vec{x})$ is some fixed non-linear function of the state vector).

2.7.3 Residual-Gradient / Galerkinized Methods

Baird (1995) gives a specific counterexample of TD(0) diverging with a linear-in-$\vec{w}$ critic function. In this divergence example, the TD(0) weight update is applied to all states in $\mathbb{S}$ simultaneously, so this is an off-policy situation and does not contradict the convergence result of the previous subsection. This divergence motivated Baird to consider the weight update formed by doing true gradient descent on Eq. (2.10) when $\lambda = 0$. He refers to this error function as the "Bellman residual error", and his method is called "residual gradients". A similar method was defined by Werbos (1990c), and he describes it as using the "Galerkin method" (Werbos, 1998). Despite this seeming to be the obvious solution to the problem, surprisingly the residual gradients method almost always converges to a suboptimal solution in stochastic environments. This was discovered and explained by Werbos (1990c), and then further explanation was given by Dayan (1992). This suboptimality in stochastic environments is now well-known, and a particularly clear explanation for the reason is given by Sutton et al. (2009).

2.7.4 Non-linear and Off-Policy Convergent Methods

Recent breakthroughs by Sutton et al. (2009) and Maei et al. (2009) have effectively solved both the off-policy TD(λ) convergence problem, and the non-linear function approximation difficulties, both of which were described in Section 2.7.2. As described in Section 2.5, off-policy algorithms are useful in VL because they allow exploration to take place while still learning about the greedy policy. And non-linear function approximation is useful to allow more powerful generalization by the critic. So solving these two problems removes a lot of difficulties that were present in reliably creating successful ADPRL architectures. The solution is only applicable to fixed action networks, i.e. neither is valid for a greedy policy.

Both of these new methods use variants of a new weight update and algorithm called Gradient Temporal Differences (GTD). This is a version of TD(λ) that is modified into being true gradient descent. They are stochastic algorithms where the expectation of the weight update equals true gradient descent on the "mean squared projected Bellman's error" (MSPBE). This is a modification to Eq. (2.10) in that each target ${J'}_t$ is modified to $P{J'}_t$, where P is a projection matrix that maps ${J'}_t$ into the closest point in the range of the function approximator $\widetilde{J}(\vec{x}, \vec{w})$ (see Sutton et al., 2009, for details).

These methods effectively have solved the long-standing convergence-guarantee issue for VL methods, with a fixed action network.

2.7.5 Convergence with an Improving Action Network

When the action network is the only function approximator used in the ADPRL architecture, convergence analysis can be quite straightforward (for example, as described in Chapter 6). However when there is a critic and an action network present, the training of the two neural networks together can interfere with each other, and it makes ensuring convergence more challenging.

All of the critic-convergence results for VL methods described in the preceding subsections, apply only when the action network is fixed. In reality, the action network needs training too, as the ultimate objective is to produce an action network with optimal behaviour. Training the critic is merely an intermediate goal. When the action network is trained simultaneously with the critic, the above convergence guarantees can fail. For example, when a greedy policy is used (which acts like an action network which updates itself to be always fully trained), even the most robust of algorithms such as TD(1) can be made to diverge, as proven in Chapter 9.

To train the action network concurrently with the critic, one or more action-network weight updates can be applied alternately with one or more critic weight updates. This process of applying alternating weight updates between the action network and critic network is known as *generalised policy iteration* (Sutton and Barto, 1998, Sec. 4.6), and various approaches of this kind are described in Sec. 5.1.1.

One of the most important convergence results for generalised policy iteration is by Sutton et al. (2000). This result applies when the function approximator for $\widetilde{J}$ is "compatible" with the function approximator for $A(\vec{x}, \vec{e}, \vec{z})$. Compatibility is defined below. It is a precondition of this convergence theorem that the critic network needs to be fully trained over the whole state space, in preparation for any action-network weight-update.

A concern with this convergence proof is that for the above precondition to hold, the learning rate of the action network must be considered to be infinitesimal. The reason for this limitation can be understood if we imagine the weight update $\Delta\vec{z}$ to be applied gradually along a straight line from $\vec{z}$ to $\vec{z} + \Delta\vec{z}$, then as soon as the first infinitesimal steps are taken along this straight line, the precondition for the theorem is immediately invalidated (since the critic's target is dependent on the current action network, which is dependent on $\vec{z}$). Another concern is that in between every infinitesimal action-network weight update, the critic must be trained to completion over the whole state space for the precondition to hold. Having an inner loop of training the critic to completion can make the combined actor-critic training process very slow. This slowness must be especially true when the environment is stochastic, because then it is very difficult to detect when the critic-training inner-loop stage has converged.

A way of interpreting the above two concerns is that the ratio of the learning rate of the actor (β) over the learning rate of the critic (α) must be infinitesimally small. This infinitesimal learning-rate ratio, and this concern about slowness, must hold for the theorem to apply. In practice it might be possible to violate these conditions and still get successful results, but this will violate the convergence guarantees provided by the proof.

For the critic and action networks to be "compatible", it is required that the critic function (a Q-function in the case of the proof by Sutton et al. (2000)) is a linear function of $\vec{w}$, and is related to the action network's function approximator as follows:

$$Q(\vec{x}, \vec{u}, \vec{w}, \vec{z}) \equiv \vec{w}^T \frac{1}{A(\vec{x}, \vec{e}, \vec{z})} \frac{\partial A(\vec{x}, \vec{e}, \vec{z})}{\partial \vec{z}}. \tag{2.11}$$

This requirement implies that the critic's Q-function has a dependency on $\vec{z}$, and is also linear in $\vec{w}$, and also that the two weight vectors $\vec{w}$ and $\vec{z}$ must have the same dimensionality. The requirement for linearity in $\vec{w}$ at first seems to be a limitation, because it means that even if a nonlinearity convergence proof exists for the critic (such as with the MSPBE methods described in Section 2.7.4), the critic must still be linear in $\vec{w}$ to ensure actor-critic convergence. However this limitation still allows for some nonlinearity in the critic, due to the dependency of Eq. (2.11) on $\vec{z}$. In essence, the full weight vector of the Q-function in Eq. (2.11) is the concatenation $(\vec{w}, \vec{z})$, and this means we can still have nonlinearity via the $\vec{z}$ contribution to the critic's concatenated weight vector.

However the requirement for linearity-in-$\vec{w}$ is still a constraint on the successful convergence proofs for the MSPBE methods described in Section 2.7.4, which were designed for general nonlinearity in the critic's weight vector. For example, the critic weight update only acts on, and must be linear in, the $\vec{w}$ part of the concatenated weight vector $(\vec{w}, \vec{z})$.

Other works by Kakade (2001) and Sutton et al. (2001) also use compatible function approximators. These provide convergence results for the combined actor and critic weight updates, but they have the same precondition for the critic network being fully trained, and therefore have the same consequence described above for β/α being infinitesimal. They also have the same requirement for the two function approximators to be compatible.

Another alternative to keeping the critic trained perfectly as the inner loop of training is to take the opposite extreme, and instead to ensure the action network is fully trained in between every single critic weight update. This particular variant of generalised policy iteration is known as *value iteration*. If the neural network used by the action network was perfectly flexible, and if the action-network's weight update was designed to make actions greedy, then this would make the action network behave exactly like the greedy policy. Therefore using a greedy policy is like using an always perfectly-trained action network, and like using a perfectly flexible function approximator for the action-network.

Learning with a greedy policy does not usually suffer from the high computational cost associated with the method of Sutton et al. (2000). For example, fast methods for evaluating a greedy policy are described in Chapter 5, which are valid under certain linearity or deterministic conditions. Alternatively, if an action network is being trained to completion, it is possible to focus the training of the action network to optimise only those actions along the current trajectory, as opposed to across the whole state space as was required for the inner loop of the method of Sutton et al. (2000).

There are no robust results for convergence for a greedy policy, when used with function approximation for the critic, in the RL literature. One algorithm, the Greedy-GQ algorithm (Maei et al., 2010) does come close to this goal, as this algorithm learns values for the greedy policy, with linear function approximation for the critic, but a limitation of this algorithm is that the agent must be following a fixed policy while learning about the greedy policy. The Natural Policy Gradient algorithm (Kakade, 2001) trains an action network to become equivalent eventually to a greedy policy, but for proven convergence requires an infinitesimal learning rate for the action network, so is the opposite of a greedy policy in this respect.

One of the value-gradient methods presented in this thesis is VGLΩ(1), and this algorithm is proven to converge while following a greedy policy, with general non-linear function approximation, in Chapter 8.

2.8 Case Study: Vertical-Lander Problem

A simple control task is demonstrated to show the ADPRL notation in action, and also to show TD(λ)'s performance on this task. In this task, a spacecraft is constrained to move in a vertical line under gravity, and the neural controller must learn to land the spacecraft gently. This is a deterministic problem, whose state space can be easily plotted in two-dimensions, and it is a good problem for illustrating the ADPRL learning algorithms described in this thesis.

2.8.1 Vertical-Lander Problem Specification

A spacecraft is constrained to move in a vertical line under gravity k_g. The spacecraft has a single thruster which can be used to make upward accelerations. The state vector for the spacecraft is $\vec{x} := (x_h, x_v, x_f)^T$, where these three components refer to height, velocity and fuel remaining, respectively. The actions $\vec{u} := (u_a)$ are one-dimensional, with $u_a \in \mathbb{R}$. u_a represents the spacecraft's thrust, which produces an instantaneous upward acceleration of the spacecraft. This action is constrained to,

$$0 \leq u_a \leq 1, \tag{2.12}$$

or in other words, $\mathbb{A} = [0, 1]$.

Each upward thrust action, u_a, produces an immediate fuel-usage cost given by,

$$U((x_h, x_v, x_f)^T, u_a) := (k_f) u_a \Delta\tau, \tag{2.13}$$

where $k_f > 0$ is a fuel-usage constant, and $\Delta\tau$ is the sampling time used for integration by the Euler method. The deterministic model function, $f(\vec{x}, \vec{u})$, used to calculate the next state is,

$$f((x_h, x_v, x_f)^T, u_a) := (x_h + x_v\Delta\tau, x_v + (u_a - k_g)\Delta\tau, x_f - (k_a)u_a\Delta\tau)^T, \tag{2.14}$$

which is a simple application of the Euler-method to apply the acceleration created by the thrust plus gravity, and to consume fuel. Here, $k_g < 1$ is a constant giving the acceleration due to gravity. Since $k_g < 1$, the spacecraft can produce greater acceleration than that due to gravity. $k_a = 1$ is a unit conversion constant.

Trajectories terminate as soon as the spacecraft hits the ground ($x_h \leq 0$) or runs out of fuel ($x_f \leq 0$). These two conditions define the terminal state set, $\mathbb{T}$. In addition to the cost function $U(\vec{x}, u_a)$ defined above, a final deterministic impulse of cost is given as soon as a terminal state, $\vec{x}_T$, is reached:

$$\Phi(\vec{x}_T) := \frac{1}{2}m(x_v)^2 + m(k_g)x_h, \tag{2.15}$$

where m is the spacecraft mass.

So there are two objectives: To conserve fuel and to land slowly, by Equations (2.13) and (2.15), respectively. Within the expression for the final impulse of cost, $\Phi(\vec{x}_T)$ given by Eq. (2.15), there are two terms, which correspond to kinetic and potential energy, respectively. The kinetic-energy term ($\frac{1}{2}m(x_v)^2$) penalises landing too quickly. The potential-energy term, $m(k_g)x_h$, is zero if the spacecraft crashes into the ground ($x_h = 0$), so this term only contributes to the final cost if the spacecraft runs out of fuel. In this case, when fuel runs out at a height $x_h > 0$, the potential term is equal to the kinetic energy that the spacecraft would acquire by crashing to the ground under free fall from height x_h, by the conservation of energy. Hence if this penalty term is used, it's equivalent to the spacecraft running out of fuel and then free falling to the ground, and finally receiving the usual kinetic-energy cost penalty term on crashing.

Since this is an episodic problem, i.e. where trajectories are guaranteed to terminate at some time, it is convenient to use $\gamma = 1$ for the discount factor. Other constants used were spacecraft mass, $m = 2$; fuel-usage constant, $k_f = 4$; system-dynamics sampling time, $\Delta\tau = 1$; gravity constant, $k_g = 0.2$.

Fig. 2.1 shows a typical trajectory in this problem.

In the state-space diagram, in the right of Fig. 2.1, the axes show velocity and height. The third state dimension, fuel, is omitted from the diagram.[5] The blue curve shows an actual trajectory (starting at the blue square). In this example, the trajectory shows the velocity becoming more and more negative until the height reaches zero. This curve represents the spacecraft crashing badly.

[5]The fuel dimension is not usually important, because provided there is enough fuel to make a gentle landing, then the shape of an optimal trajectory is independent of the amount of fuel left over at the end of the trajectory.

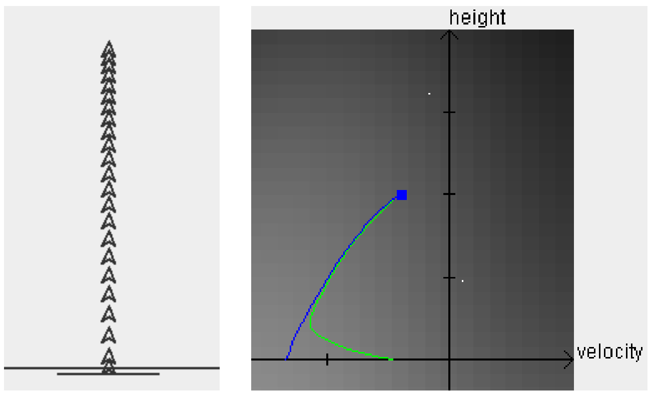


Figure 2.1: State-Space View of Vertical-Lander Trajectories. The left image shows a time-lapse view of a typical spacecraft trajectory, i.e. the spacecraft descending in a vertical line. The blue curve in the right-hand image shows the corresponding trajectory in a state-space view. The green curve shows the theoretical optimal trajectory.

The green curve shows the theoretical optimal trajectory, which was calculated as described by (Fairbank, 2008, Appendix E.2) using Pontryagin's Minimum Principle. The optimal trajectory shows the spacecraft free-falling for an initial time, and then using the thruster to slow the spacecraft down to land gently. This is consistent with the fact that optimal behaviour for a vertical descent is to descend freely for as long as possible and then to brake as hard and late as possible. This can be understood intuitively since the converse, hovering down slowly, would be extremely wasteful of fuel, and therefore the optimal strategy is to behave as differently from hovering as is possible; which means freefalling for as long as possible, and then braking at the last possible moment.[6] Also, contrary to what we might expect, the optimal trajectory does not need to reach the ground with exactly zero velocity. This is because minimising the chosen cost function represents a compromise between landing slowly and saving fuel.

[6]This can alternatively be understood as a time-reversal of a rocket taking off. Rockets are designed with fuel efficiency as a top priority, and they are built to give one massive thrust to achieve escape velocity in as short a time as possible, and then afterwards to coast freely into space.

2.8.2 Use of Value-Learning Methods to Solve the Vertical-Lander Problem

Since the Vertical-Lander is a deterministic problem, its solution by VL methods will require some explicit value exploration over the state space. This is to satisfy the requirement in Bellman's Condition to have a correct value function over the whole state space. This explicit value exploration can be provided either by using a stochastic policy, and/or using a varying choice of the start state ("exploring starts"). In this section the effectiveness of these two methods will be investigated.

First a stochastic policy is considered, of the form,

$$\vec{u}_t = A(\vec{x}_t, \vec{z}) + X_\sigma, \tag{2.16}$$

where $A(\vec{x}_t, \vec{z})$ is a deterministic action network, and X_σ is a normally-distributed random variable, with zero mean and standard deviation σ. Eq. (2.16) is truncated if necessary to satisfy $\vec{u}_t \in \mathbb{A}$. In the case of the Vertical-Lander Problem, this truncation will force the constraint in Eq. (2.12) to be obeyed. The stochastic policy given by Eq. (2.16) would be inserted into line 3 of Alg. 2.1 (or line 4 of Alg. 2.2). The standard deviation, σ, specifies the amount of stochastic exploration that the policy should perform. When $\sigma = 0$, no exploration is done. When $\sigma > 0$ the policy will make exploratory actions, and hence the trajectory will take a zigzag shape.

The action and critic networks were trained using Alg. 2.2 modified into batch-mode. (The weight updates were accumulated but only applied when a trajectory, or set of trajectories, had been completely unrolled.) The network architectures were as specified in Table 2.1. Neural networks work best when their input vectors are rescaled to unit length, hence rescaled coordinates of $\vec{x}' := \text{diag}(0.01, 0.1, 0.035)\vec{x}$ were used throughout the experiments.

Results are shown both with exploration ($\sigma = 0.1$) and without exploration ($\sigma = 0$), for TD(0), TD(0.9) and TD(1), in Fig. 2.2. In all cases, each trajectory was forced to start from the state $(h, v, u)^T = (100, -2, 50)^T$, so there was no form of exploration allowed other than that provided by the parameter σ appearing in Eq. (2.16). The learning rates α and β were tuned through search for each experiment and shown in the graph titles. The results are shown as $J - J^*$, where J^* is the optimal trajectory cost, which from this start state and $\Delta\tau$ value is 23.917.

Critic-Network Details	
Network Dimensions	A $3-6-6-1$ multilayer-perceptron (Bishop, 1995), i.e. $\dim(\vec{x})$ input nodes, two hidden layers of six nodes each, and an output layer with exactly one node.
Bias weights present?	Yes, one for each network node.
Short-cut connections present?	Yes, between all layers (including non-adjacent ones).
Activation Functions on non-final network layers	$f(x) := \tanh(x)$.
Activation Functions on final network layer	Linear, with slope 10, i.e. $f(x) := 10x$.
Weights randomisation method	Uniform probability distribution $[-0.1, 0.1]$.
Action-Network Details	
All details same as critic-network, except for:	
Activation Functions on non-final network layers	Logistic sigmoid, $f(x) := \frac{1}{1+e^{-x}}$.

Table 2.1: Network Architectures used for Vertical-Lander Experiments

The results show that TD(λ) can solve this problem, slowly, when $\lambda < 1$ and when value exploration is present (i.e. when $\sigma > 0$), but it fails without value exploration ($\sigma = 0$) for the reason stated above. When value exploration is removed, the algorithms converge to a large suboptimal J value.

The stochastic policy was necessary for success with TD(λ) with $\lambda < 1$. However, TD(1) finds this problem very difficult even when a stochastic policy is used. This is thought to be because the stochastic policy produces a large variance in the learning signal (i.e. the variance in ${J'}_t$). Reducing this variance was one of the motivations for using $\lambda < 1$, as described in Section 2.3.1, and therefore this explains why the results for $\lambda < 1$ are better than for $\lambda = 1$.

The effectiveness of intermediate λ values is shown in Fig. 2.3. The results indicate that an optimal choice is $\lambda \approx 0.9$ for TD(λ) to solve this problem efficiently. These results were generated after a search for optimal learning rates, which resulted in $\alpha = 10^{-5}$ (for the critic) and $\beta = 10^{-4}$ (for the action network).

A difficulty with this stochastic-policy approach is that the value-learning algorithm is learning values for a suboptimal policy (since the policy is stochastic), and therefore learning cannot converge to truly optimal behaviour. One solution to this problem

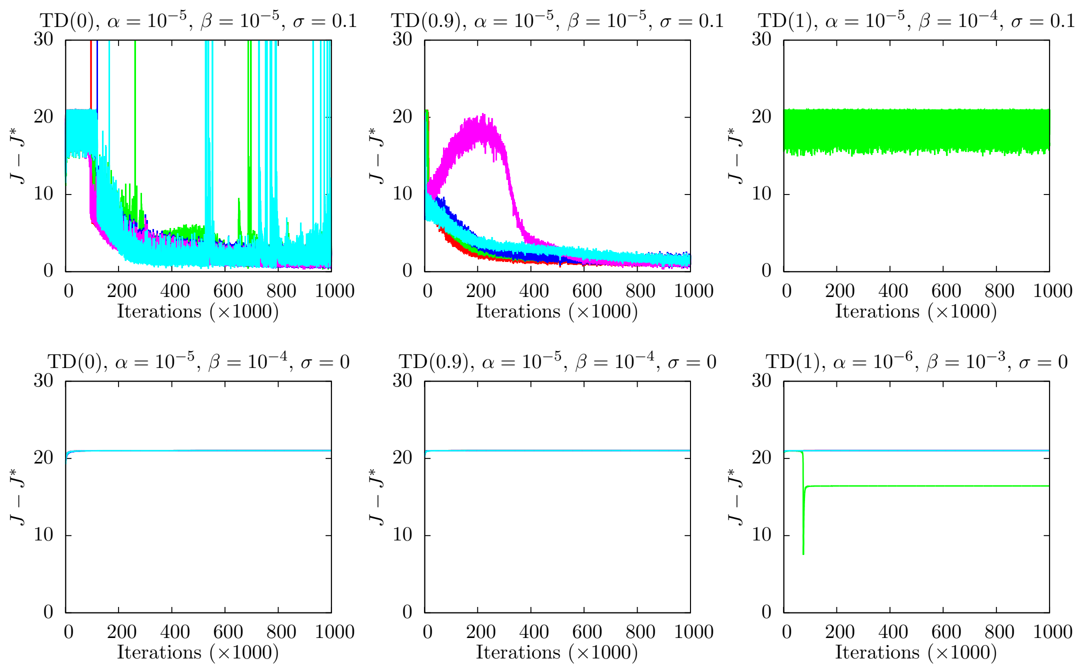


Figure 2.2: Results for Using a Stochastic Policy to Solve the Vertical-Lander Problem by TD(λ)/HDP algorithms. These results correspond to the experiment described in Section 2.8.2. The graphs show total trajectory cost versus training iterations, for five different random weight initialisations each shown in a different colour, both with policy noise ($\sigma = 0.1$) and without policy noise ($\sigma = 0$). Each graph's set of five curves largely overlay each other, but this indicates consistency of results. For successful learning, the graphs should show J decreasing as the number of training iterations increases, which only occurs for two top-left graphs. (For comparison to VGL methods, see also Fig. 3.7.)

would be to let σ slowly tend to zero, but this will slow down learning even more. Off-policy algorithms are also designed to address this specific problem (Section 2.5), but there are other possible solutions, such as the following exploring-starts method.

"Exploring starts" is an alternative form of exploration which addresses the above problem of a stochastic policy, and it also addresses the difficulty encountered by TD(1) with a stochastic policy. Using exploring starts, the experiment was repeated in batch mode on ten trajectories simultaneously. The trajectories now had fixed scattered start points, as specified in Table 2.2, and a deterministic policy function (i.e. $\sigma = 0$). In this case, because the there was no randomness, TD(1) does not suffer from the disruptive variance in the learning signal, J'_t; and also the behaviour is not forced to be suboptimal because of the addition of random actions. Hence the results for TD(1) are much improved, as shown in Fig. 2.4. The critic learning rate used was $\alpha = 10^{-6}$,

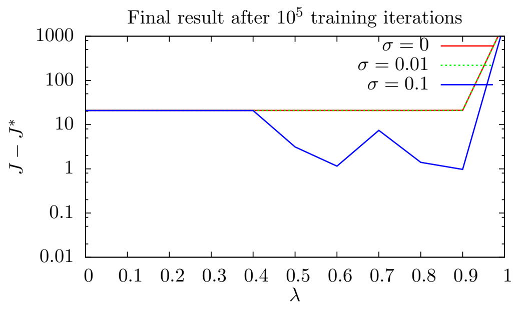


Figure 2.3: Results of TD(λ) under various λ and noise-levels, σ, for solving the Vertical-Lander Problem, from a fixed trajectory start point. All results are generated for critic learning rate $\alpha = 10^{-5}$ and actor learning rate $\beta = 10^{-4}$. For each value of λ, the graph shows the value of $J - J^*$ after 10^5 training iterations. All results are gathered as the median result of 10 trials (i.e. from 10 different initial randomizations of the neural weights).

and the action-network learning rate used was $\beta = 10^{-4}$. The results are shown as $J - J^*$, averaged over all 10 trajectories. Here J^* is the optimal trajectory cost, which from this set of 10 fixed start states, averages to 27.372.

h	100	50	100	50	100	50	100	50	100	50
v	0.0	-1.0	-2.0	-3.0	-4.0	-5.0	-6.0	-7.0	-8.0	-9.0
u	50	50	50	50	50	50	50	50	50	50

Table 2.2: Start-State Coordinates used for Exploring-Starts Vertical-Lander Experiment

The exploring-starts performance seems better than the stochastic-policy results. For example, convergence to a low J value occurs more consistently and within 100,000 iterations, for both TD(1) and TD(0); whereas in the previous experiments this only happened for TD(λ) with $\lambda < 1$ and within 1,000,000 iterations. However, a potential problem still exists with the exploring-starts method, in that the trajectories at the sides of the explored state space are not fully explored on either side of their start points, so these trajectories will often not be optimal (because Bellman's Condition requires the value function to be known over the whole state space to be strictly valid). Hence these side-most trajectories contribute to a poorer average J value than would otherwise be possible.

In the next chapter, the value-gradient methods will be defined and shown to produce better results on the Vertical-Lander than seen so far. In Section 3.6, the VGL

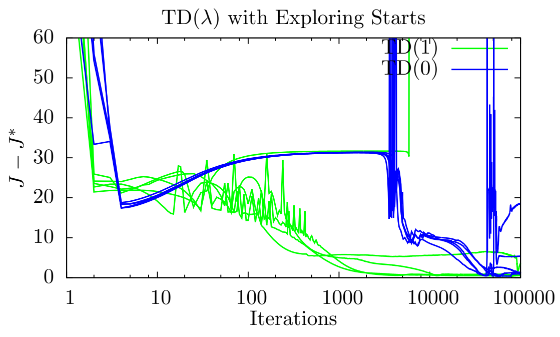


Figure 2.4: Results for Using Exploring Starts to Solve the Vertical-Lander Problem by TD(λ). The graph shows the trajectory cost averaged over 10 trajectories with fixed start points. Each curve shows a different experimental trial (i.e. different weight initialisation); five different trials for TD(1) and five trials for TD(0). In this experiment, both algorithms make significant progress in solving the problem. (For comparison to VGL methods, see also Fig. 3.9.)

methods will be shown to solve the Vertical-Lander problem with more consistent and lower J values, in fewer training iterations, and work well across all values of λ and σ.

2.9 Chapter Conclusions

This chapter has introduced and defined the two main VL algorithms: HDP and TD(λ). The important Bellman's optimality principle has been described, and so have other variant VL algorithms, such as Q-learning, ADHDP and Gradient-Temporal Differences. The convergence and divergence properties of VL algorithms have been discussed.

A simple deterministic problem, the Vertical-Lander, has been defined. This is a good illustrative problem because it is simple to describe, it is episodic, and its state space is suitable for two-dimensional plotting. The results showed that TD(λ) could solve the Vertical-Lander reasonably well, when value exploration was provided through exploring starts or a stochastic policy. In the case of TD(1), only the deterministic exploring-starts method was successful. In the total absence of value exploration, TD(λ) could not solve the problem for any value of λ attempted. This will provide a motivation for developing VGL algorithms in the next chapter.

2.A Chapter Appendix

The introduction of the recursive formula for J'_t in Eq. (2.3) is possibly an original contribution of early technical reports (Fairbank, 2002, 2008) that led to this thesis. This appendix proves that J'_t is equivalent to the "λ-Return", a quantity previously defined by Watkins (1989) which can be used to specify the targets of the TD(λ) weight update, and hence to define the TD(λ) weight update concisely.

2.A.1 Equivalence of the J' Notation to the λ-Return

Although it was first discovered by Watkins (1989), the definition of the λ-Return given by (Sutton and Barto, 1998, sec.7.2) is used:

$$R_t^\lambda := (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}R_t^{(n)}, \tag{2.17}$$

with

$$R_t^{(n)} := \left(\sum_{k=0}^{n-1}\gamma^k U_{t+k}\right) + \gamma^n \widetilde{J}_{t+n}. \tag{2.18}$$

The aim here is to show that J'_t is identical to R_t^λ.

In this section, only continuing trajectories are analysed. Episodic trajectories and terminal states are not explicitly considered here, because they can be seen to be a special case of continuing trajectories if we define the terminal state as a holding state which traps the agent at the point forever afterwards (such that $\vec{x}_t = \vec{x}_T \ \forall t \geq T$) and delivers an infinite number of zero costs afterwards ($U_t = 0 \ \forall t > T$; $U_t = \Phi_t$ for $t = T$).

Expanding the definition of R_t^λ gives

$$\begin{aligned}
R_t^\lambda :=& (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\left(\left(\sum_{k=0}^{n-1}\gamma^k U_{t+k}\right)+\gamma^n\widetilde{J}_{t+n}\right) \qquad \text{(Combining Eqs. (2.17) \& (2.18))}\\
=&(1-\lambda)\left[U_t+\lambda(U_t+\gamma U_{t+1})+\lambda^2(U_t+\gamma U_{t+1}+\gamma^2U_{t+2})+\ldots\right]\\
&+(1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\gamma^n\widetilde{J}_{t+n}\\
=&(1-\lambda)\left[U_t\left(1+\lambda+\lambda^2+\ldots\right)+\gamma U_{t+1}\left(\lambda+\lambda^2+\lambda^3+\ldots\right)\right.\\
&\left.+\gamma^2U_{t+2}\left(\lambda^2+\lambda^3+\lambda^4+\ldots\right)+\ldots\right]+(1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\gamma^n\widetilde{J}_{t+n} \qquad \text{(reordering terms)}
\end{aligned}$$

$$
\begin{aligned}
&=(1-\lambda)\sum_{n=0}^{\infty}\left(\gamma^n U_{t+n}\sum_{k=n}^{\infty}\lambda^k\right)+\gamma(1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\gamma^{n-1}\widetilde{J}_{t+n}\\
&=(1-\lambda)\sum_{n=0}^{\infty}\left(\gamma^n U_{t+n}\left(\frac{\lambda^n}{1-\lambda}\right)\right)+\gamma(1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\gamma^{n-1}\widetilde{J}_{t+n} \qquad \text{(geometric series)}\\
&=\sum_{n=0}^{\infty}\gamma^n\lambda^n U_{t+n}+\gamma(1-\lambda)\sum_{n=0}^{\infty}\lambda^n\gamma^n\widetilde{J}_{t+n+1}\\
&=\sum_{n=0}^{\infty}\gamma^n\lambda^n\left(U_{t+n}+\gamma(1-\lambda)\widetilde{J}_{t+n+1}\right) \qquad (2.19)
\end{aligned}
$$

Expanding the definition of J'_t (Eq. (2.3)) for non-terminal states gives

$$
\begin{aligned}
J'_t &:= U_t+\gamma\left(\lambda J'_{t+1}+(1-\lambda)\widetilde{J}_{t+1}\right)\\
&=\sum_{n=0}^{\infty}\gamma^n\lambda^n\left(U_{t+n}+\gamma(1-\lambda)\widetilde{J}_{t+n+1}\right) \qquad \text{(expanding recursion)}\\
&=R_t^{\lambda} \qquad \text{(by (2.19))}
\end{aligned}
$$

Thus J'_t is identical to R_t^{λ}. The λ-Return provides an equivalent formulation for the algorithm TD(λ) known as the "forwards view of TD(λ)" (Sutton and Barto, 1998, sec.7.2). This proves that Eq. 2.1 is equivalent to the TD(λ) weight update. The concise formulation for J'_t given in Eq. (2.3) is much more convenient than the formula for R_t^{λ}, and it is this concise form that allowed the VGL(λ) algorithm to be created, as described in Section 3.5.4.

Chapter 3

Value-Gradient Learning Algorithms

The VGL algorithms are defined in this chapter. VGL algorithms are algorithms which explicitly aim to learn the critic gradient, $\frac{\partial \widetilde{J}}{\partial \vec{x}}$. These algorithms include the original VGL algorithms by Werbos (1992b), which are Dual Heuristic Programming (DHP) and Globalized Dual Heuristic Programming (GDHP); and also the new algorithm of this thesis, VGL(λ).

The motivations for using VGL methods over VL methods are described in Section 3.1, which are principally that local value-exploration is automatic for VGL methods. However VGL methods are model-based methods that require a learned or known differentiable model of the environment functions, $f(\vec{x}, \vec{u}, \vec{e})$, $U(\vec{x}, \vec{u}, \vec{e})$ and $\Phi(\vec{x}, \vec{e})$. These requirements of VGL methods are limitations compared to VL methods which are model-free methods, and which do not require differentiability of the environment functions. These limitations of VGL methods are discussed further in Section 3.2.

In this chapter, the existing DHP and GDHP methods are first defined. Then the VGL(λ) algorithm is defined as an extension of DHP to include a bootstrapping parameter $\lambda \in [0, 1]$, just as TD(λ) is an extension of TD(0). For $\lambda = 0$, VGL(λ) is equivalent to DHP; but for $\lambda > 0$, VGL(λ) is a new algorithm. The relationship of VGL(λ) to DHP is represented in Fig. 1.1 by the right-hand vertical arrow labelled "insert λ". The motivations for including the λ parameter into the VGL(λ) algorithm are that using $\lambda > 0$ can increase the stability of learning and sometimes increase learning speed, as discussed further in section 3.5.3.

The rest of this chapter is structured as follows: Section 3.1 expands on the motivations for VGL method. Section 3.2 discusses the applicability of VGL methods and the necessity of using learned or known differentiable environment functions. The DHP algorithm is defined in Section 3.4, and the VGL(λ) algorithm is defined in Section 3.5. Motivations for including the λ parameter into DHP to form VGL(λ) are given in Section 3.5.3, and the relationship of VGL(λ) to TD(λ) is described in Section 3.5.4.

In Section 3.6, the Vertical-Lander Problem is revisited, and the efficiency and ability of the VGL method to solve it without any explicit value exploration, is shown. In Section 3.7, further experimental results are given, showing the efficiency and automatic exploration of the methods. Finally, in Section 3.8 chapter conclusions are given.

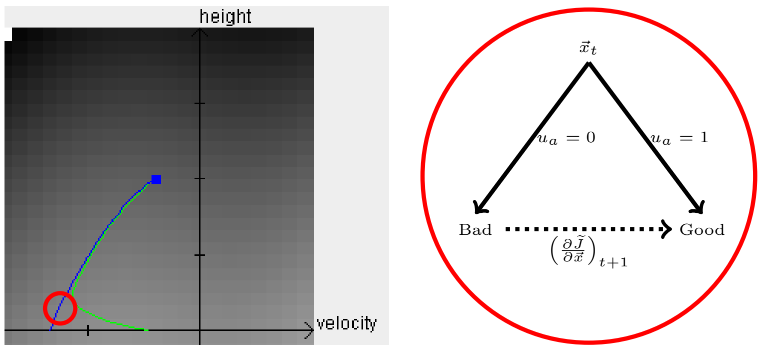
3.1 Motivation for VGL Methods

The main reason to develop VGL methods is to obtain automatic local value exploration, detailed as follows. It was observed in the experiments of Sec. 2.8.2 that when value exploration was removed, VL could not solve the Vertical-Lander Problem. For example, from a fixed trajectory start point and with $\sigma = 0$ for the policy exploration, the graphs in Fig. 2.2 show this failure. This failure was expected because Bellman's Condition requires knowledge of the value function over the whole state space to be applicable; but this knowledge was removed by setting $\sigma = 0$.

As opposed to considering Bellman's requirement to learn the critic over the whole state space, if we consider what critic information needs to be learned locally around a particular trajectory, we can deduce that failure is occurring because the neighbouring preferable trajectories are not recommended as being better ones by the critic. Therefore there needs to be a gradient of critic values (a value gradient) between the current trajectory and the neighbouring ones, to give the information about which neighbouring trajectories are better. To attain optimality, somehow this value gradient needs to be learned. VL methods aim to learn the value gradient indirectly, through value exploration. VGL methods aim to learn it explicitly.

Fig. 3.1 describes this motivation for VGL methods in more detail.

Therefore the intention of using a VGL method is that it will obviate the need to do explicit local value exploration. With VGL, it is hoped that trajectories will bend themselves into locally optimal shapes, i.e. local value exploration will be automatic.


height
velocity
$\vec{x}_t$
$u_a = 0$
$u_a = 1$
Bad
Good
$\left(\frac{\partial \widetilde{J}}{\partial \vec{x}}\right)_{t+1}$

Key: Blue line: Actual trajectory
Green line: Theoretical optimal trajectory
Red circle: The large red circle is a detail of the small red circle

Figure 3.1: The Motivation of Automatic Local Value Exploration for VGL in the Vertical-Lander Problem.
The left image shows the converged results of TD(0)/HDP in the Vertical-Lander Problem, from a fixed trajectory start point and in the absence of value exploration. The HDP error is almost zero ($\sum_t (J'_t - \widetilde{J}_t)^2 = 7.38 \times 10^{-5}$), but the learned trajectory (the blue curve) has not moved at all towards the optimal trajectory (the green curve). The grey-scale background indicates the magnitude of the critic function learned at each point in state space. This left diagram shows that without value exploration, VL methods can converge to severely suboptimal trajectories.
This right-hand diagram shows an enlargement of a particular decision being made during one time step of the crashing trajectory shown in the left-figure. This diagram only shows the two extremes of choice, i.e. $u_a = 0$, or $u_a = 1$, whereas in reality there is also a continuum of choices in between. The optimal decision is to choose the arrow on the right ($u_a = 1$). Since the greedy policy is defined to choose the action which minimises $U_t + \gamma \widetilde{J}_{t+1}$, it is necessary for $\widetilde{J}_{t+1}$ to be significantly lower on the right branch than the left. Therefore we need a significant average gradient in the values of $\widetilde{J}_{t+1}$ across the state space, decreasing in the direction of the dotted arrow (assuming $\widetilde{J}(\vec{x}, \vec{w})$ is a smooth function). Furthermore the decrease in $\widetilde{J}_{t+1}$ along this dotted arrow needs to be monotonic, since every continuous increment in u_a leads to an improvement in the trajectory's cost. This demonstrates that the critic gradient (the dotted arrow) needs to be learned, and this motivates the VGL method. Note that this gradient could also be learned by VL with value exploration. But learning the gradient by VGL will make a greedy policy automatically bend the trajectory into the correct shape, without any need for explicit value exploration.

Experiments at the end of this chapter confirm that this does happen. Of course it might still be necessary to do some global exploration, as local optimality might not be sufficient; but it is hoped that overall using VGL will simplify the exploration process.

It turns out to be only necessary to fully learn the value gradient along a single trajectory, under a greedy policy, for it to be locally extremal, and often locally optimal.

This is proven for deterministic environments in Chapter 7, and is closely related to Pontryagin's Minimum Principle (Chapter 4). Both VL and VGL methods have the same requirement for global optimality, that is if the value function (or its gradient) is exactly learned all over the state space, with a greedy policy, then Bellman's condition assures global optimality. This issue is proven in Section 4.5, for both stochastic and deterministic environments (under certain smoothness assumptions).

By making local value exploration automatic, the VGL method aims to solve some problems that were affecting the experiments given in Section 2.8.2, which were that:

- A stochastic policy ($\sigma > 0$) in the Vertical-Lander Problem is necessarily suboptimal. Since TD(λ) is an on-policy method, TD(λ) will be learning values for the suboptimal stochastic policy. One way to avoid this problem is to slowly let $\sigma \to 0$ over time, but this has performance implications.

- Off-policy solutions to the above problem can have their own problems, for example as noted in Section 2.6, Q-learning experiences eligibility-trace truncation whenever an exploratory action is taken.

- Exploring starts can be used and obtain better results, but these are not perfect either because they can have difficulties learning the edge-most trajectories properly (as noted in Section 2.8.2).

- When a fixed trajectory start point is used, the vertical-lander from cannot easily be solved by VL methods without some form of stochastic assistance.

Another intention is that VGL methods will lead to faster learning. The experiments at the end of this chapter show this speed up in learning.

Fig. 3.2 shows a representation of what VGL algorithms learn in each sampled trajectory, compared to what VL algorithms would have to do to learn the same information by sampling multiple trajectories. This diagram is intended to show how quickly VGL methods can learn the value-function surface compared to VL methods, and also how VGL algorithms go about learning the value gradient in a very direct and straightforward manner. Since learning the value gradient is necessary for optimality and providing automatic local-exploration (as described in Fig. 3.1), this motivates the design of VGL algorithms.

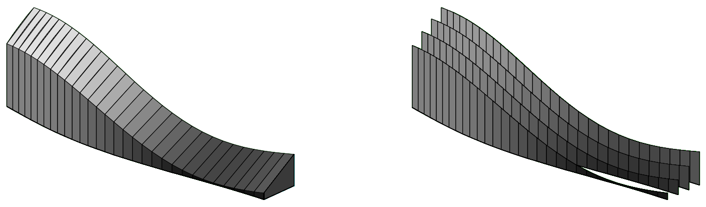


Figure 3.2: The Portion of Value-Function Surface Learned by VGL Compared to that Learned by VL. The left-hand image shows what the VGL learning-targets describe in just one sampled trajectory, and the right-hand image shows what the VL learning-targets describe in four sampled trajectories. The purpose of this image is to show how much more information the VGL weight update contains, compared to the VL weight update. However, this is a simplistic view; in reality VL would be hindered by several further factors: 1. The height-profiles in the right-hand diagram would show distortion due to the necessary random value exploration that VL needs, and thus contain much less useful information. 2. Trajectories often are not parallel like this but may converge, thus learning less useful area of the value-function surface. 3. The VL image would take at least $\dim(\vec{x})$ trajectories to learn as much as the VGL approach can learn in just one trajectory (hence the four plotted trajectories are meant to be representative of higher dimensions).

A final intention of VGL methods is that convergence analyses could become simplified. Since the greedy policy and the value gradient depend upon each other, as described in Fig. 3.1, it is potentially easier to understand what effect a VGL weight update will have upon a greedy policy, and thus more easily predict whether divergence or convergence can occur when the policy and critic interact with each other. This forms the basis of the convergence and divergence proofs in Chapters 8 and 9, respectively.

3.2 Applicability and Limitations of the VGL algorithms

VGL methods are strictly model-based methods. It is assumed that environment functions are known a priori, or at least can be learned by a separate "system identification" learning process, for example as described by Werbos et al. (1992). This system identification process could have taken place prior to the main learning process (e.g. like Ng et al. (2004)'s successful application of a hovering inverted helicopter), or concurrently with it, and results in learned environment functions $\bar{f}(\vec{x}, \vec{u})$, $\bar{U}(\vec{x}, \vec{u})$ and $\bar{\Phi}(\vec{x})$, consistent with Eqs. (1.9)-(1.10). Alternatively, there is an extremely fast online model-learning method by Munos (2006) which could be used, which is capable of learning the necessary derivatives of the model and cost functions at the same time as the trajectory is unrolled. That method is thought to be exact only in deterministic continuous-time environments. Except for Section 3.7.1, where model-learning is briefly described for an example problem, the model-learning process is not discussed further in this thesis.

The VGL methods work naturally with continuous-valued state-space problems, and are not applicable to discrete-valued state spaces. They are also limited to situations to where the learned environment functions and policy are once-differentiable, since VGL algorithms require usage of derivatives of the form $\frac{\partial \bar{f}(\vec{x},\vec{u})}{\partial \vec{x}}$ and $\frac{\partial \bar{f}(\vec{x},\vec{u})}{\partial \vec{u}}$, and similar derivatives of $\bar{U}(\vec{x},\vec{u})$ and $\bar{\Phi}(\vec{x})$. Sometimes, as described more in Section 3.2.1, the derivatives of the true environment functions can be used in place of these learned environment-function derivatives.

VGL methods also require the differentiability of the policy function $A(\vec{x},\vec{e},\vec{z})$. This can be achieved by using a neural network for the action network. However when a greedy policy is used, it can be harder to assure differentiability since bang-bang control often arises then. Using the techniques of Chapter 5, it is possible to define greedy-policy functions that are always smooth and differentiable. In their main form, VGL methods also require a continuous-valued action space. However it is possible to override this limitation using a method described later in Section 5.2.1.

It should be noted that since they are model-based, VGL algorithms optimise performance with respect to the learned environment functions, as opposed to the true environment functions.

VGL methods work well in continuous-valued state spaces. This has led to successes in industrial control (Venayagamoorthy and Wunsch, 2003), autopilot landing (Prokhorov and Wunsch, 1997a), and many others (Wang et al., 2009). However some traditional RL problem domains with discrete spaces would not be readily solvable by VGL methods, such as a “grid world” problem, backgammon, or a k-armed bandit problem. Also, step cost functions would not be applicable, thus excluding problems such balancing a pole where the total cost is a function of the *integer* number of time steps that the pole is balanced for. However the pole-balancing problem can be solved by VGL if a smoothed out cost function is used, as shown in Chapter 5, or if the number of time steps is transformed into a continuous quantity, as demonstrated in Chapter 10. As a rule of thumb, if a problem is suitable for smooth gradient descent on $\widehat{J}$ with respect to $\vec{w}$, then it will be suitable to work on VGL methods.

3.2.1 The Use of Learned Environment Functions in VGL Algorithms

VGL methods make use of the derivatives of the environment functions, $f(\vec{x}, \vec{u}, \vec{e})$, $U(\vec{x}, \vec{u}, \vec{e})$ and $\Phi(\vec{x}, \vec{e})$. That is why VGL methods are classified as model-based (or model-given) methods. However it is usually more practical for the VGL algorithms to work with the derivatives of the *learned* environment functions, $\bar{f}(\vec{x}, \vec{u})$, $\bar{U}(\vec{x}, \vec{u})$ and $\bar{\Phi}(\vec{x})$, as opposed to derivatives of the true environment functions, because the learned functions often are what the system has knowledge of; and also because these learned functions can more easily be assured to be differentiable (for example the learned functions may be represented by a neural network, which will force the necessary differentiability). Another reason to use the learned environment functions is because true partial derivatives of the form $\frac{\partial f(\vec{x}, \vec{u}, \vec{e})}{\partial \vec{x}}$ require knowledge of the $\vec{e}$ vector, which may not be a fully-observable quantity (as noted in Section 1.2). A significant exception where the true environment functions definitely could be used is where the environment is intended to be a virtual world, for example a computer game, and then the function $f(\vec{x}, \vec{u}, \vec{e})$ and the vector $\vec{e}$ could be assumed to be fully known; but still, $f(\vec{x}, \vec{u}, \vec{e})$ would have to be differentiable to be applicable for VGL algorithms.

For the above reasons, the DHP, VGL(λ) and BPTT algorithms presented in this thesis are written explicitly with the use of the derivatives of the learned environment functions, $\frac{\partial \bar{f}}{\partial \vec{x}}$, $\frac{\partial \bar{f}}{\partial \vec{u}}$, $\frac{\partial \bar{U}}{\partial \vec{x}}$, $\frac{\partial \bar{U}}{\partial \vec{u}}$ and $\frac{\partial \bar{\Phi}}{\partial \vec{x}}$. However if the above conditions for an exception exist, then these derivatives could be replaced by the derivatives of the true environment functions.

In general stochastic environments, these learned derivatives will only be approximations to the true derivatives, i.e.:

$$\frac{\partial \bar{f}}{\partial \vec{x}} \approx \frac{\partial f}{\partial \vec{x}}; \quad \frac{\partial \bar{f}}{\partial \vec{u}} \approx \frac{\partial f}{\partial \vec{u}}; \quad \frac{\partial \bar{U}}{\partial \vec{x}} \approx \frac{\partial U}{\partial \vec{x}}; \quad \frac{\partial \bar{U}}{\partial \vec{u}} \approx \frac{\partial U}{\partial \vec{u}}; \quad \frac{\partial \bar{\Phi}}{\partial \vec{x}} \approx \frac{\partial \Phi}{\partial \vec{x}}. \tag{3.1}$$

The circumstances under which the derivatives of the learned functions will equal the derivatives of the true functions, i.e. the circumstances under which the approximation symbols in Eq. (3.1) could be turned into equality symbols, are if the noise terms in Eq. (1.9) were all independent of $\vec{x}$ and $\vec{u}$. This deduction follows by consideration of

Eqs. (1.9)-(1.10), and can be summarised as:

$$\begin{aligned} f(\vec{x},\vec{u},\vec{e}) \equiv \bar{f}(\vec{x},\vec{u}) + \xi_f(\vec{e}) &\Rightarrow \frac{\partial \bar{f}}{\partial \vec{x}} \equiv \frac{\partial f}{\partial \vec{x}}; \frac{\partial \bar{f}}{\partial \vec{u}} \equiv \frac{\partial f}{\partial \vec{u}} \\ U(\vec{x},\vec{u},\vec{e}) \equiv \bar{U}(\vec{x},\vec{u}) + \xi_U(\vec{e}) &\Rightarrow \frac{\partial \bar{U}}{\partial \vec{x}} \equiv \frac{\partial U}{\partial \vec{x}}; \frac{\partial \bar{U}}{\partial \vec{u}} \equiv \frac{\partial U}{\partial \vec{u}} \\ \Phi(\vec{x},\vec{e}) \equiv \bar{\Phi}(\vec{x}) + \xi_\Phi(\vec{e}) &\Rightarrow \frac{\partial \bar{\Phi}}{\partial \vec{x}} \equiv \frac{\partial \Phi}{\partial \vec{x}}. \end{aligned} \tag{3.2}$$

The above set of equations describes what will be referred to as *uniform additive noise*. Another obvious situation where the learned derivatives are equal to the true derivatives is in a deterministic environment, in which case the noise terms in the above equations vanish and the equalities automatically hold. Hence some references define VGL-based methods as being defined for the situation of deterministic environments or stochastic environments with additive noise (Fairbank and Alonso, 2012c), as this will ensure that Eq. (3.2) holds.

To summarise, the derivatives of the environment functions used by the VGL algorithms are usually only approximations to the derivatives of the true environment functions, i.e. Eq. (3.1) generally holds; except in three circumstances:

1. If the true environment functions are the ones that are used in the VGL algorithm (which is possible in a virtual world such as a computer game), and are also differentiable; or,

2. If the environment is deterministic, and the learned environment functions are exact (according to Eq. (1.10)); or,

3. If the noise is uniform and additive, as defined by Eq. (3.2), and the learned environment functions are exact (according to Eq. (1.10)).

In any of these three circumstances, the VGL(λ)/DHP/BPTT algorithms' pseudocode could exchange all the uses of learned derivatives (e.g. $\frac{\partial \bar{f}}{\partial \vec{x}}$) by true derivatives (e.g. $\frac{\partial f}{\partial \vec{x}}$).

3.3 Vector and Jacobian Notation for VGL Algorithms

Before defining the value-gradient algorithms, it is necessary to define some basic notation describing how we differentiate vector functions and functions of vector arguments.

As previously stated in Section 2.1.1, all defined vector quantities are columns, whether they are coordinates, or derivatives with respect to coordinates. For example, $\vec{u}$, $\vec{w}$, $\vec{x}$, $\vec{z}$, $\frac{\partial \widetilde{J}(\vec{x},\vec{w})}{\partial \vec{w}}$ and $\frac{\partial J(\vec{x},\vec{\mathbf{e}},\vec{z})}{\partial \vec{x}}$ are all column vectors.

Vector-by-vector differentiation is defined by example. $\frac{\partial \bar{f}}{\partial \vec{x}}$ is defined to be a matrix with element (i, j) equal to $\frac{\partial \bar{f}(\vec{x},\vec{u})^j}{\partial \vec{x}^i}$. This is the transpose of the usual Jacobian notation. Similarly, for a vector function $\widetilde{G}(\vec{x}, \vec{w})$, we have $\left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)^{ij} = \frac{\partial \widetilde{G}^j}{\partial \vec{w}^i}$. Combining with trajectory-shorthand notation defined in Section 2.1.2, $\left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t$ is the matrix $\frac{\partial \widetilde{G}}{\partial \vec{w}}$ evaluated at $(\vec{x}_t, \vec{w})$.

3.4 The DHP and GDHP Algorithms

The *approximate value gradient*, or *critic gradient*, is defined to be

$$\widetilde{G}(\vec{x}, \vec{w}) := \frac{\partial \widetilde{J}(\vec{x}, \vec{w})}{\partial \vec{x}}, \tag{3.3}$$

where $\widetilde{J}(\vec{x}, \vec{w})$ is the scalar critic function previously defined.

A key concept in the DHP based algorithms is the *target value gradient*. This is the value-gradient analogy to the “target value” used in TD(0) (Equation (2.2)). Using the definitions from the previous section, and some implied matrix-vector products, the target value gradient for DHP is defined to be

$$G'_t := \begin{cases} \left(\frac{D\bar{U}}{D\vec{x}}\right)_t + \gamma \left(\frac{D\bar{f}}{D\vec{x}}\right)_t \widetilde{G}_{t+1}, & \text{for } \vec{x}_t \notin \mathbb{T} \\ \left(\frac{\partial \Phi}{\partial \vec{x}}\right)_t, & \text{for } \vec{x}_t \in \mathbb{T} \end{cases} \tag{3.4}$$

where $\frac{D}{D\vec{x}}$ is shorthand for

$$\frac{D}{D\vec{x}} := \frac{\partial}{\partial \vec{x}} + \frac{\partial A}{\partial \vec{x}} \frac{\partial}{\partial \vec{u}} \ ; \tag{3.5}$$

so that for example $\left(\frac{D\bar{f}}{D\vec{x}}\right)_t$ is the derivative $\frac{D\bar{f}(\vec{x},\vec{u})}{D\vec{x}}$ evaluated with arguments $(\vec{x}_t, \vec{u}_t)$ for the given trajectory, and $\left(\frac{D\bar{f}}{D\vec{x}}\right)_t \widetilde{G}_{t+1}$ is an implied matrix-vector product.

This transformation of the VL target value, J', to the DHP target value, G', is represented by the uppermost horizontal arrow in Fig. 1.1 which is labelled $\frac{\partial}{\partial \vec{x}}$. Whereas the VL algorithms aim to make $\widetilde{J}_t$ equal $\mathbb{E}\left(J'_t\right)$ for all t, the VGL algorithms aim to make $\widetilde{G}_t$ equal $\mathbb{E}\left(G'_t\right)$ for all t, hence the name “value-gradient learning”.

Using these definitions, and the implied matrix-vector products, the DHP algorithm is defined by a weight update of the form:

$$\Delta\vec{w} = \alpha \sum_t \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t \left(G'_t - \widetilde{G}_t\right) \tag{3.6}$$

where $\alpha > 0$ is the learning-rate; G'_t and $\widetilde{G}_t$ are the target and approximate value gradients, respectively; $\Omega_t \in \mathbb{R}^{\dim(\vec{x})\times\dim(\vec{x})}$ is an arbitrary positive-definite matrix explained in Section 3.4.3, which can be chosen freely by the researcher; and where all of these derivatives are assumed to exist. Equations (3.4)-(3.6) define the DHP algorithm.[7]

Pseudocode for the DHP algorithm is given in Algorithm 3.1. An action-network weight update is included in line 7 of the code, which is equivalent to the one used earlier for HDP (in Alg. 2.1). This is the commonly used model-based ADP action-network weight update (e.g. Prokhorov and Wunsch, 1997a; Wang et al., 2009; Werbos, 1992b), but other weight-update schemes are also possible, as described in Chapter 5.

The DHP algorithm has a running time of O(max(dim($\vec{w}$), dim($\vec{z}$), dim($\vec{w}_f$))) operations per trajectory time step, where dim($\vec{w}_f$) is the running time of the function $\bar{f}(\vec{x}, \vec{u})$, as proven in Section 3.4.2. DHP therefore has the same asymptotic running time as the VL algorithms HDP and TD(λ).

Algorithm 3.1 Actor-Critic, On-Line Implementation of DHP algorithm.

```
1:  $t \leftarrow 0$
2:  **while** $\vec{x}_t \notin \mathbb{T}$ **do**
3:      $\vec{u}_t \leftarrow A(\vec{x}_t, \vec{e}_t, \vec{z})$
4:      $\vec{x}_{t+1} \leftarrow f(\vec{x}_t, \vec{u}_t, \vec{e}_t)$
5:      Calculate $G'_t$ by Eq. (3.4).
6:      $\vec{w} \leftarrow \vec{w} + \alpha \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t \left(G'_t - \widetilde{G}_t\right)$
7:      $\vec{z} \leftarrow \vec{z} - \beta \left(\frac{\partial A}{\partial \vec{z}}\right)_t \left(\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \widetilde{G}_{t+1}\right)$ {Action-network weight update}
8:      $t \leftarrow t + 1$
9:  **end while**
10: $\vec{w} \leftarrow \vec{w} + \alpha \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t \left(\left(\frac{\partial \Phi}{\partial \vec{x}}\right)_t - \widetilde{G}_t\right)$
```

This completes the core details of DHP. Subsections 3.4.1 to 3.4.4 describe some

[7] Note that Werbos (1992b) used the variable names $\hat{\lambda}$ and λ^* in place of $\widetilde{G}$ and G', respectively, when defining the DHP algorithm. I used different variable names to avoid the clash with the λ in TD(λ), and also to avoid the clash with the superscripted asterisk in Bellman's optimal value function, J^*.

technical issues and design variants possible for DHP. These include: Two different ways that the function $\widetilde{G}(\vec{x},\vec{w})$ can be implemented (sec. 3.4.1); how the computation complexity of DHP was calculated (sec. 3.4.2); what choices exist for the Ω_t matrix (sec. 3.4.3); and what the variant algorithm GDHP is exactly (sec. 3.4.4). After those short technical sections, the VGL(λ) will be defined in the next main section (Section 3.5).

3.4.1 Scalar Critics and Vector Critics

The theoretical definition of $\widetilde{G}(\vec{x},\vec{w})$ given so far was by Eq. (3.3), i.e. $\widetilde{G} := \frac{\partial \widetilde{J}}{\partial \vec{x}}$. However when it comes to programming a DHP/VGL implementation, the function $\widetilde{G}(\vec{x},\vec{w})$ can be implemented in two different ways.

The first way is to use the theoretical definition as the basis for an actual implementation, i.e. to implement the function $\widetilde{G}(\vec{x},\vec{w}) := \frac{\partial \widetilde{J}(\vec{x},\vec{w})}{\partial \vec{x}}$. This will be referred to as using a “scalar critic”. Since the DHP algorithm uses terms of the form $\frac{\partial \widetilde{G}}{\partial \vec{w}}$, when a scalar critic is used, these terms will require second-order derivatives; since then, $\frac{\partial \widetilde{G}}{\partial \vec{w}} \equiv \frac{\partial^2 \widetilde{J}}{\partial \vec{w} \partial \vec{x}}$. Chapter 11 describes techniques to calculate this second derivative very efficiently.

The second way, and the more common way to implement DHP, is to have $\widetilde{G}(\vec{x},\vec{w})$ directly implemented as the output of a smooth vector function approximator, such as a neural network with output dimension $\dim(\vec{x})$ (for example, see Prokhorov and Wunsch, 1997a; Werbos, 1992b). This method will be referred to as using a “vector critic”. Traditionally, DHP has always been defined this way, contrary to the way DHP was defined in Section 3.4.

Out of the two options, usually a vector critic is easier to implement, since it only requires first-order backpropagation to calculate the $\frac{\partial \widetilde{G}}{\partial \vec{w}}$ term.

Figs. 3.3 and 3.4 show illustrations of a scalar critic compared to a vector critic.

A scalar critic is better for understanding the connection of DHP to value learning: When DHP aims to learn $\widetilde{G}$ all over $\mathbb{S}$, it is really learning $\frac{\partial \widetilde{J}}{\partial \vec{x}}$ all over $\mathbb{S}$, which is the same as learning $\widetilde{J}(\vec{x},\vec{w})$ all over $\mathbb{S}$ (with the addition of an arbitrary constant). This statement is proven in Section 4.5. So this meets the value-learning objective, and Bellman’s Condition too. Another possible advantage of using a scalar critic is that it might be advantageous for $\widetilde{G}$ to be the true gradient of a scalar field $\widetilde{J}(\vec{x},\vec{w})$. This would force it to satisfy the vector calculus identity that its curl must be zero

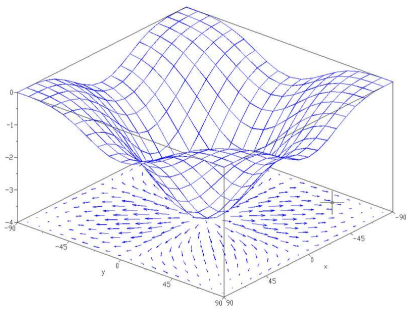


Figure 3.3: Scalar Critic. The vector field (shown below) is derived as the gradient of the value function surface (shown above).

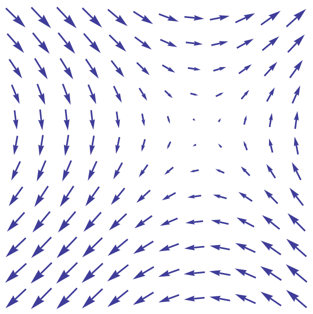


Figure 3.4: Vector Critic. The vector field is arbitrary.

everywhere, unlike the pattern of vectors shown in Fig. 3.5, and this might yield some advantage. But this remains to be seen.

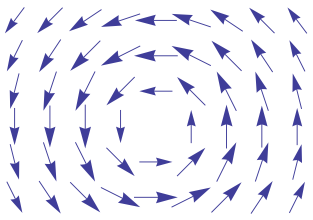


Figure 3.5: A non-zero curl field allowed by a vector critic. This pattern of vectors is allowed by a vector critic, but will never be required by an optimal value function, since the curl is non-zero. A scalar critic has an advantage of not allowing this possibility.

3.4.2 Algorithmic Complexity of DHP

In calculating the complexity of Alg. 3.1 (DHP), the computationally expensive lines to consider are lines 5-7. Whenever a matrix-matrix-vector product appears, it is most efficient to evaluate it from the right first (using the associativity of matrix multiplication).

Line 5 of Alg. 3.1 expands into

$$G'_t = \left(\frac{\partial \bar{U}}{\partial \vec{x}}\right)_t + \left(\frac{\partial A}{\partial \vec{x}}\right)_t \left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \left(\left(\frac{\partial \bar{f}}{\partial \vec{x}}\right)_t + \left(\frac{\partial A}{\partial \vec{x}}\right)_t \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \right) \widetilde{G}_{t+1}$$

using Eqs. (3.4) and (3.5). If the model function $\bar{f}(\vec{x}, \vec{u})$ is given analytically then the derivative matrices $\frac{\partial \bar{f}}{\partial \vec{x}}$ and $\frac{\partial \bar{f}}{\partial \vec{u}}$ will be known exactly, and the products of these with

$\widetilde{G}_{t+1}$ can be done in $\mathrm{O}(\dim(\vec{x})^2)$ operations, assuming $\dim(\vec{u}) < \dim(\vec{x})$. In fact, these derivative matrices are usually sparse, so the products are usually slightly faster than this. By considering the number of weights necessary for a fully connected multilayer perceptron (MLP) neural network $\widetilde{G}(\vec{x}, \vec{w})$ with $\dim(\vec{x})$ inputs and $\dim(\vec{x})$ outputs, we usually have $\dim(\vec{x})^2 \leq \dim(\vec{w})$.

Alternatively if the model function $\bar{f}(\vec{x}, \vec{u})$ was not available analytically and had to be learned, e.g. by a neural network with weight vector $\vec{w}_f$, then the matrix-vector products involving $\frac{\partial \bar{f}}{\partial \vec{x}}$ and $\frac{\partial \bar{f}}{\partial \vec{u}}$ could be calculated by backpropagation in $\mathrm{O}(\dim(\vec{w}_f))$ operations (Rumelhart et al., 1986; Werbos, 1974). If something different and more complicated than a neural network was used to implement $\bar{f}(\vec{x}, \vec{u})$, then assuming the function is smooth enough to be differentiated (which is a necessary condition to be using DHP anyway), it should still be possible to calculate the products involving the derivatives of $\bar{f}(\vec{x}, \vec{u})$ efficiently using generalised backpropagation, which is also known as automatic differentiation (Rall, 1981; Werbos, 1974, 2005). Automatic differentiation is capable of doing static analysis on the computer code for $\bar{f}(\vec{x}, \vec{u})$, and producing from it a "dual" subroutine dedicated to calculating the matrix-vector product that we seek. Furthermore, the dual routine will run asymptotically just as fast as the running time of the original function $\bar{f}(\vec{x}, \vec{u})$. Let the asymptotic running time of the function $\bar{f}(\vec{x}, \vec{u})$ be $\mathrm{O}(\dim(\vec{w}_f))$, as if it were a neural network with weight vector $\vec{w}_f$. So in any case, the multiplications by $\frac{\partial \bar{f}}{\partial \vec{x}}$ and $\frac{\partial \bar{f}}{\partial \vec{u}}$ can be done in $\mathrm{O}(\dim(\vec{w}))$ or $\mathrm{O}(\dim(\vec{w}_f))$.

The analysis for derivatives of $\bar{U}(\vec{x}, \vec{u})$ can be handled in a similar way as for the derivatives of $\bar{f}(\vec{x}, \vec{u})$, and they do not contribute significantly to the calculation time of line 5.

The multiplications by $\frac{\partial A}{\partial \vec{x}}$ in line 5 can be done in $O(\dim(\vec{z}))$ operations if backpropagation is used through the action network $A(\vec{x}, \vec{e}, \vec{z})$. So considering this and the previous three paragraphs, the total operation count for line 5 is $\mathrm{O}(\max(\dim(\vec{w}), \dim(\vec{z}), \dim(\vec{w}_f)))$.

Line 6 contains a matrix-matrix-vector product. Since the matrix Ω_t has dimension $\dim(\vec{x}) \times \dim(\vec{x})$, its product with $({G'}_t - \widetilde{G}_t)$, can be calculated first in $(\dim(\vec{x}))^2$ operations. Afterwards the product with $\frac{\partial \widetilde{G}}{\partial \vec{w}}$ can be calculated by backpropagation in $\mathrm{O}(\dim(\vec{w}))$ operations (using the techniques of Chapter 11 if a scalar critic was chosen). So again assuming $(\dim(\vec{x}))^2 \leq \dim(\vec{w})$, the operation count for this line is $\mathrm{O}(\dim(\vec{w}))$.

Using similar backpropagation methods to those described above, line 7 can be evaluated in $\mathrm{O}(\max(\dim(\vec{z}), \dim(\vec{w}_f)))$.

The maximum asymptotic running time for the three lines of code analysed above is therefore $O(\max(\dim(\vec{w}), \dim(\vec{z}), \dim(\vec{w}_f)))$. Therefore this is the asymptotic running time for the whole DHP algorithm, per trajectory time step. Although this is the same asymptotic running time as for the VL algorithms, the DHP code is likely to be slower by a small constant factor, due to the fact that more passes of backpropagation are required in the DHP code than in the VL algorithms. Experiments in Section 3.6.1 show this constant factor can be quite small. For example in those experiments, value-gradient methods were only 7% slower per iteration than the corresponding VL methods.

3.4.3 Omega Matrix's Purposes and Origins

The Ω_t matrix of the DHP weight update (Eq. (3.6)) only appears in some descriptions of DHP/GDHP in the ADP literature (for example see Werbos, 1998, eq. 32), and it has never had the t subscript prior to its use in value-gradient learning (Fairbank, 2008). In other cases it is omitted, which is equivalent to setting $\Omega_t = I$, the identity matrix. One reason for its usual omission from the ADP literature might be down to the difficulty of deciding what value it should take, since Ω_t is a free parameter for the researcher to choose.

Ω_t is included in the weight update for generality, since the presence of any positive-definite matrix here in Eq. (3.6) will force every component of $\widetilde{G}_t$ to move towards the corresponding component of ${G'}_t$ (in any basis). For example, if Ω_t was chosen to be a diagonal matrix, as it was originally defined by Werbos (1987), then the relative values of the diagonal elements could be chosen to specify how much priority the DHP weight update should assign to learning the different components of the value gradient. This kind of design decision for a diagonal Ω_t matrix can be achieved implicitly in VL by the rescaling the state space axes, but the use of Ω_t allows us to make this choice explicit. There seems to be no VL equivalent method for simulating a non-diagonal Ω_t matrix.

The choice of a non-diagonal and time-step-dependent Ω_t matrix turns out to be key in proving equivalence between VGL(λ) and BPTT in Chapter 8.

3.4.4 The GDHP Algorithm

The algorithm Globalized Dual Heuristic Programming (GDHP) is a variant on DHP created by Werbos (1987), and is another precursor algorithm to VGL(λ). The GDHP algorithm is a critic-learning algorithm which is only defined for a scalar critic. It is equivalent to a linear combination of DHP (Eq. (3.6)) and HDP (Eq. (2.1)). The GDHP weight update can therefore be written as

$$\begin{aligned}\Delta\vec{w} &= \eta_1 \sum_t \left(\frac{\partial \widetilde{J}}{\partial \vec{w}}\right)_t (J'_t - \widetilde{J}_t) + \eta_2 \sum_t \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t \left(G'_t - \widetilde{G}_t\right) \\ &= \eta_1 \sum_t \left(\frac{\partial \widetilde{J}}{\partial \vec{w}}\right)_t (J'_t - \widetilde{J}_t) + \eta_2 \sum_t \left(\frac{\partial^2 \widetilde{J}}{\partial \vec{w}\partial \vec{x}}\right)_t \Omega_t \left(G'_t - \left(\frac{\partial \widetilde{J}}{\partial \vec{x}}\right)_t\right)\end{aligned}$$

where $\eta_1 > 0$ and $\eta_2 > 0$ are the relative learning rates between the HDP learning term and the DHP learning term, respectively.

Two motivations for why GDHP was designed over DHP are given by Werbos (2004). These are that using a scalar critic would enforce "*strict adherence to the requirement that the vectors $\widetilde{G}_t$ are the gradient of a [scalar] function J*". This point has already been discussed in the previous section (Section 3.4.1). Also, Werbos (2004) writes, "*GDHP allows one to train the critic to minimize a weighted sum of the DHP second-order error measure and the usual HDP/TD first-order error measure; if some state variables are continuous while others are discrete, one can use GDHP on the entire J function simply by not using the (undefined) second-order terms for the discrete state variables*". VGL methods were designed for continuous-valued state spaces. This second statement by Werbos indicates that GDHP was designed to be a hybrid between VL and VGL, such that the advantages of VGL could apply to the continuous-valued state dimensions, and the advantages of VL could apply to the discrete dimensions.

3.5 The VGL(λ) Algorithm

In this section the VGL(λ) Algorithm is defined and its relation to its precursor algorithms TD(λ) and Dual Heuristic Programming is described.

The VGL(λ) algorithm is an extension to DHP to include a $\lambda \in [0,1]$ constant analogous to that used in TD(λ). Hence the target value gradient defined for DHP (Eq. (3.4)) is modified into:

$$G'_t := \begin{cases} \left(\frac{D\bar{U}}{D\vec{x}}\right)_t + \gamma \left(\frac{D\bar{f}}{D\vec{x}}\right)_t \left(\lambda G'_{t+1} + (1-\lambda)\widetilde{G}_{t+1}\right), & \text{for } \vec{x}_t \notin \mathbb{T} \\ \left(\frac{\partial \bar{\Phi}}{\partial \vec{x}}\right)_t, & \text{for } \vec{x}_t \in \mathbb{T} \end{cases} \tag{3.7}$$

This is the only change needed to define VGL(λ). The weight update is then the same as the DHP weight update (Eq. (3.6)), i.e.

$$\Delta\vec{w} = \alpha \sum_t \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t ({G'}_t - \widetilde{G}_t)$$

Equations (3.6), (3.7) and (3.5) define the VGL(λ) algorithm. Section 3.5.1 gives further implementation details and pseudocode. This completes the modification of DHP into VGL(λ). As with DHP, there is a free choice as to whether $\widetilde{G}(\vec{x}, \vec{w})$ should be implemented by a vector critic or a scalar critic, as described in Section 3.4.1.

Like with TD(λ), the recursion in Eq. (3.7) is forced to converge by requiring that either $\gamma < 1$, $\lambda < 1$, or the environment is such that the agent is guaranteed to reach a terminal state at some finite time (i.e. the environment is episodic). Since the VGL(λ) target (${G'}_t$ by Eq. (3.7)) is related to the DHP target (${G'}_t$ by Eq. (3.4)) in the same way that the TD(λ) target (${J'}_t$ by Eq. (2.3)) is related to the TD(0) target (${J'}_t$ by Eq. (2.2)), the modification of DHP to VGL(λ) is represented by the right-hand vertical arrow labelled "insert λ" in Fig. 1.1.

The target value-gradients, G', are so called because the VGL objective is to achieve as closely as possible $\widetilde{G}_t = \mathbb{E}({G'}_t)$ for all t along a trajectory. This objective, if attained closely enough, ensures a locally extremal, and often locally optimal, trajectory (as proven for the deterministic case in Chapter 7), when combined with a greedy policy. It should be noted that this objective is not straightforward to achieve since the targets ${G'}_t$ are moving ones and are highly dependent on $\vec{w}$ (especially when a greedy policy is being used, so that then the policy is also indirectly dependent on $\vec{w}$). Hence we must use the weight update to slowly move the approximated gradients towards their targets. Also, due to the limitations of function approximation, it will not ever be possible to attain the targets exactly, in general, but only arbitrarily close to this goal.

The Ω_t matrix is the same as one defined for DHP, and discussed previously in Section 3.4.3. For DHP and GDHP it has never been clear how to choose this matrix, but for the special choice of

$$\Omega_t = \begin{cases} \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)^T_{t-1} \left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{u} \partial \vec{u}}\right)^{-1}_{t-1} \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_{t-1} & \text{for } t > 0 \\ 0 & \text{for } t = 0, \end{cases} \tag{3.8}$$

the algorithm VGL(1) is proven in Chapter 8 to converge, under certain smoothness assumptions, for a sufficiently small learning rate and when used in conjunction with a policy that is greedy on the following model-based approximate Q-Value function:

$$\widetilde{Q}(\vec{x},\vec{u},\vec{w}) := \bar{U}(\vec{x},\vec{u}) + \gamma \widetilde{J}(\bar{f}(\vec{x},\vec{u}),\vec{w}). \tag{3.9}$$

Throughout this thesis, whenever this special Ω_t matrix is used in combination with VGL(λ), the algorithm will be referred to as "VGLΩ(λ)". In absence of the specification of Ω_t, if an algorithm is referred just to as VGL(λ) then it will be assumed that the Ω_t matrix was not used at all, or equivalently, it was set to the identity matrix. Variants of VGL(λ) other than VGLΩ(1) are not proven to converge with a greedy policy, and divergence examples do exist and are presented in Chapter 9.

In the following subsections, further details on aspects of VGL(λ) are given. These cover: Pseudocode for the algorithm in two forms (sec. 3.5.1); details of how the "on-line" version of the pseudocode was derived (sec. 3.5.2); details on the motivations for inserting a λ parameter into DHP to make VGL(λ) (sec. 3.5.3); and details of how VGL(λ) relates directly to TD(λ) (sec. 3.5.4).

3.5.1 Implementation of VGL(λ)

Pseudocode is now given for the VGL(λ) algorithm. It would be possible to use the DHP algorithm (Alg. 3.1) directly, with just a small tweak to use the recursive Eq. (3.7) to define ${G'}_t$. But just like the target value ${J'}_t$ used by TD(λ) in the previous chapter, there are two choices in how to unroll this recursion. It can be done either forwards or backwards in time, therefore there are two possible versions of the pseudocode for VGL(λ); one is for on-line learning which can be continually applied as trajectories are expanded (analogous to Alg. 2.2 for TD(λ)), and one is a batch-mode implementation which is slightly more efficient but is only applicable to completed trajectories.

Algorithm 3.2 makes a direct implementation of VGL(λ) for episodic environments. It makes a forward pass through the trajectory, storing all states and actions, followed by a backward pass through the trajectory accumulating ${G'}_t$ by the recursion in Eq. (3.7).

All steps in this algorithm have similar complexity to their corresponding lines in the DHP algorithm, therefore the running time is the same, i.e. O(dim($\vec{w}$)) operations per

Algorithm 3.2 VGL(λ). Actor-Critic, Batch-Mode Implementation for Episodic Environments.

```
1: Unroll full trajectory using Alg. 1.1, and retain variables $\vec{x}_t$, $\vec{u}_t$ and $T$.
2: $G'_T \leftarrow \left(\frac{\partial \bar{\Phi}}{\partial \vec{x}}\right)_T$, $\Delta\vec{z} \leftarrow \vec{0}$
3: {Backwards pass...}
4: $\Delta\vec{w} \leftarrow \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_T \Omega_T \left(G'_T - \widetilde{G}_T\right)$
5: for $t = T-1$ to 0 step $-1$ do
6:     $\vec{p} \leftarrow \lambda G'_{t+1} + (1-\lambda)\widetilde{G}_{t+1}$
7:     $G'_t \leftarrow \left(\frac{\partial \bar{U}}{\partial \vec{x}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{x}}\right)_t \vec{p} + \left(\frac{\partial A}{\partial \vec{x}}\right)_t \left(\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \vec{p}\right)$
8:     $\Delta\vec{w} \leftarrow \Delta\vec{w} + \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t \left(G'_t - \widetilde{G}_t\right)$
9:     $\Delta\vec{z} \leftarrow \Delta\vec{z} - \left(\frac{\partial A}{\partial \vec{z}}\right)_t \left(\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \widetilde{G}_{t+1}\right)$
10: end for
11: $\vec{w} \leftarrow \vec{w} + \alpha\Delta\vec{w}$
12: $\vec{z} \leftarrow \vec{z} + \beta\Delta\vec{z}$
```

trajectory time step (assuming the complexity of the model, action and critic networks are all approximately equal to each other).

Algorithm 3.3 is the on-line version of VGL(λ) that is suitable for non-episodic environments. It accumulates the aggregate weight update for VGL(λ) in a single forward pass of the trajectory. Compared to the episodic version of the algorithm (Alg. 3.2) which must store a copy of the trajectory so it can later do a backwards pass along it, this version is more memory efficient in that it does not require any storage of the trajectory. On the other hand, the on-line algorithm requires more time to carry out matrix multiplications. The derivation of this algorithm is given in Section 3.5.2.

This on-line version of the VGL(λ) algorithm requires the full $\frac{\partial \widetilde{G}}{\partial \vec{w}}$ matrix which, for a neural network, would take $O(\dim(\vec{x})\dim(\vec{w}))$ operations to evaluate. Hence the slowest steps in the algorithm would be the matrix-matrix multiplications of lines 7 and 9, each taking $O((\dim(\vec{x}))^2\dim(\vec{w}))$ operations. Hence the total time for the algorithm to run is $O((\dim(\vec{x}))^2\dim(\vec{w}))$ operations per trajectory time step (i.e. slower by a factor of $(\dim(\vec{x}))^2$ than the episodic version of VGL(λ), and the TD(λ) algorithm, and DHP).

In the case of $\lambda = 0$, the algorithm can be optimised to remove the variable E, and then it becomes equivalent to the algorithm for DHP previously stated.

Both algorithms incorporate the action-network weight update used by DHP, with

Algorithm 3.3 VGL(λ). Actor-Critic, On-Line Implementation.

```
1: $E \leftarrow 0$ {$E \in \mathbb{R}^{\dim(\vec{w})\times\dim(\vec{x})}$ is an "eligibility trace" workspace matrix.}
2: $t \leftarrow 0$
3: while $\vec{x}_t \notin \mathbb{T}$ do
4:     $\vec{u}_t \leftarrow A(\vec{x}_t, \vec{e}_t, \vec{z})$
5:     $\vec{x}_{t+1} \leftarrow f(\vec{x}_t, \vec{u}_t, \vec{e}_t)$
6:     $\vec{\delta} \leftarrow \left(\frac{\partial \bar{U}}{\partial \vec{x}}\right)_t + \left(\frac{\partial A}{\partial \vec{x}}\right)_t \left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma\left(\left(\frac{\partial \bar{f}}{\partial \vec{x}}\right)_t + \left(\frac{\partial A}{\partial \vec{x}}\right)_t \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t\right)\widetilde{G}_{t+1} - \widetilde{G}_t$
7:     $E \leftarrow E + \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t$
8:     $\vec{w} \leftarrow \vec{w} + \alpha E\vec{\delta}$
9:     $E \leftarrow \lambda\gamma E\left(\left(\frac{\partial \bar{f}}{\partial \vec{x}}\right)_t + \left(\frac{\partial A}{\partial \vec{x}}\right)_t \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t\right)$
10:    $\vec{z} \leftarrow \vec{z} - \beta\left(\frac{\partial A}{\partial \vec{z}}\right)_t \left(\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma\left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \widetilde{G}_{t+1}\right)$
11:    $t \leftarrow t + 1$
12: end while
13: $\vec{\delta} \leftarrow \left(\frac{\partial \Phi}{\partial \vec{x}}\right)_t - \widetilde{G}_t$
14: $E \leftarrow E + \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t$
15: $\vec{w} \leftarrow \vec{w} + \alpha E\vec{\delta}$
```

a learning rate of $\beta > 0$. If a different actor weight-update scheme was needed then these lines could be moved or replaced. Chapter 5 describes alternative action-network weight-update schemes and the greedy policy.

3.5.2 Derivation of the On-line VGL(λ) algorithm

To derive the on-line VGL(λ) algorithm (Alg. 3.3), Eq. (3.7) was rewritten in the case of $\vec{x}_t \notin \mathbb{T}$, as follows:

$$\begin{aligned}
G'_t &= \left(\frac{D\bar{U}}{D\vec{x}}\right)_t + \gamma\left(\frac{D\bar{f}}{D\vec{x}}\right)_t \left(\lambda G'_{t+1} + (1-\lambda)\widetilde{G}_{t+1}\right) \\
&= \left(\frac{D\bar{U}}{D\vec{x}}\right)_t + \gamma\left(\frac{D\bar{f}}{D\vec{x}}\right)_t \widetilde{G}_{t+1} + \lambda\gamma\left(\frac{D\bar{f}}{D\vec{x}}\right)_t \left(G'_{t+1} - \widetilde{G}_{t+1}\right) \\
\Rightarrow G'_t - \widetilde{G}_t &= \left(\left(\frac{D\bar{U}}{D\vec{x}}\right)_t + \gamma\left(\frac{D\bar{f}}{D\vec{x}}\right)_t \widetilde{G}_{t+1} - \widetilde{G}_t\right) + \lambda\gamma\left(\frac{D\bar{f}}{D\vec{x}}\right)_t \left(G'_{t+1} - \widetilde{G}_{t+1}\right) \\
&= \vec{\delta}_t + \lambda\gamma\left(\frac{D\bar{f}}{D\vec{x}}\right)_t \left(G'_{t+1} - \widetilde{G}_{t+1}\right), \qquad (3.10)
\end{aligned}$$

where we define

$$\vec{\delta}_t = \left(\frac{D\bar{U}}{D\vec{x}}\right)_t + \gamma \left(\frac{D\bar{f}}{D\vec{x}}\right)_t \widetilde{G}_{t+1} - \widetilde{G}_t.$$

Unrolling the recursion in $({G'}_t - \widetilde{G}_t)$ of Eq. (3.10) gives

$${G'}_t - \widetilde{G}_t = \vec{\delta}_t + \lambda\gamma \left(\frac{D\bar{f}}{D\vec{x}}\right)_t \vec{\delta}_{t+1} + \lambda^2\gamma^2 \left(\frac{D\bar{f}}{D\vec{x}}\right)_t \left(\frac{D\bar{f}}{D\vec{x}}\right)_{t+1} \vec{\delta}_{t+2} + \dots \tag{3.11}$$

The above recursion was derived ignoring the possibility of a terminal state being reached. If the trajectory does happen to reach a terminal state, at time step T, then by the definition of Eq. (3.7) we have ${G'}_T - \widetilde{G}_T = \left(\frac{\partial\bar{\Phi}}{\partial\vec{x}}\right)_T - \widetilde{G}_T$. So to include this possibility of a terminal state being reached, we can redefine $\vec{\delta}_t$ to be

$$\vec{\delta}_t = \begin{cases} \left(\frac{D\bar{U}}{D\vec{x}}\right)_t + \gamma \left(\frac{D\bar{f}}{D\vec{x}}\right)_t \widetilde{G}_{t+1} - \widetilde{G}_t & \text{if } \vec{x}_t \notin \mathbb{T}, \\ \left(\frac{\partial\bar{\Phi}}{\partial\vec{x}}\right)_t - \widetilde{G}_t & \text{if } \vec{x}_t \in \mathbb{T}. \end{cases} \tag{3.12}$$

Using this new definition of $\vec{\delta}_t$, the recursion in Eq. (3.11) terminates with a final $\vec{\delta}_T$ in the case that a terminal state is reached, and goes on infinitely if not.

Then substituting this recursion (Eq. (3.11)) into the VGL(λ) weight update equation (Eq. (3.6)) and reordering the terms gives:

$$\begin{aligned} \Delta\vec{w} &= \alpha \sum_{t=0}^{T} \left(\frac{\partial\widetilde{G}}{\partial\vec{w}}\right)_t \Omega_t \left(\vec{\delta}_t + \lambda\gamma \left(\frac{D\bar{f}}{D\vec{x}}\right)_t \vec{\delta}_{t+1} + \lambda^2\gamma^2 \left(\frac{D\bar{f}}{D\vec{x}}\right)_t \left(\frac{D\bar{f}}{D\vec{x}}\right)_{t+1} \vec{\delta}_{t+2} + \dots\right) \\ &= \alpha \sum_{t=0}^{T} \left(E_t \vec{\delta}_t\right) \end{aligned} \tag{3.13}$$

where E_t is a matrix defined to be

$$\begin{aligned} E_t = {} & \left(\frac{\partial\widetilde{G}}{\partial\vec{w}}\right)_t \Omega_t + \lambda\gamma \left(\frac{\partial\widetilde{G}}{\partial\vec{w}}\right)_{t-1} \Omega_{t-1} \left(\frac{D\bar{f}}{D\vec{x}}\right)_{t-1} + \lambda^2\gamma^2 \left(\frac{\partial\widetilde{G}}{\partial\vec{w}}\right)_{t-2} \Omega_{t-2} \left(\left(\frac{D\bar{f}}{D\vec{x}}\right)_{t-2} \left(\frac{D\bar{f}}{D\vec{x}}\right)_{t-1}\right) \\ & + \dots + \lambda^t\gamma^t \left(\frac{\partial\widetilde{G}}{\partial\vec{w}}\right)_0 \Omega_0 \left(\left(\frac{D\bar{f}}{D\vec{x}}\right)_0 \left(\frac{D\bar{f}}{D\vec{x}}\right)_1 \cdots \left(\frac{D\bar{f}}{D\vec{x}}\right)_{t-2} \left(\frac{D\bar{f}}{D\vec{x}}\right)_{t-1}\right). \end{aligned}$$

We see that E_t can be defined more simply by a recursion:

$$E_t = \left(\frac{\partial\widetilde{G}}{\partial\vec{w}}\right)_t \Omega_t + \lambda\gamma E_{t-1} \left(\frac{D\bar{f}}{D\vec{x}}\right)_{t-1}, \tag{3.14}$$

with $E_{-1} = 0$. We call the matrix E_t an "eligibility trace" matrix because it acts similarly to the eligibility trace described for TD(λ) (Sutton, 1988), appearing in Alg. 2.2. Algorithm 3.3 is then easily derived from Equations (3.12), (3.13) and (3.14).

3.5.3 Motivations for Introducing the λ Parameter into DHP

The motivations for using the λ parameter are very similar to those described for TD(λ) in Section 2.3.1, i.e. that choosing λ carefully can increase the stability of learning and sometimes increase learning speed, and can result in reduced variance of the stochastic quantities being learned.

In the case of VGL(λ) compared to TD(λ), having a high value of λ produces two further benefits:

1. When $\lambda = 1$, the algorithm VGLΩ(λ) is proven to converge even with a greedy policy and a non-linear critic function, under certain conditions, as described fully in Chapter 8.

2. VGL methods work well in deterministic environments without stochastic value exploration, whereas VL methods do not. Also VGL methods seem to work well in stochastic environments with a high value of λ, but VL methods do not. These two reasons mean that one of the principal benefits of having a low value of λ, i.e. that of variance reduction of stochastic quantities, is not always relevant or useful. Examples showing these differences are given in the experiments of this chapter (e.g. specifically in Figs. 3.7 and 3.8).

The above two reasons define two further good motivations for extending the DHP algorithm into VGL(λ).

3.5.4 Relationship of VGL(λ) to TD(λ)

Although VGL(λ) has been defined in this thesis as a modification to DHP, it was originally designed as a modification to TD(λ). This was done to overcome the difficulties encountered in deterministic control problems, which resulted in suboptimal trajectories such as that shown in Fig. 3.1.

VGL(λ) was derived from TD(λ) by differentiating each term in the TD(λ) weight update with respect to $\vec{x}$. Hence the route from TD(λ) to VGL(λ) is indicated in Fig. 1.1 by the lower-most horizontal arrow labelled "$\frac{\partial}{\partial \vec{x}}$".

To observe how this modification was made, first notice that the TD(λ) and VGL(λ) weight update equations, Equations (2.1) and (3.6), respectively, have very similar form. But the first one of these equations is designed to make approximated scalar values move towards target scalar values, and the second one is designed to make approximated value gradients move towards target value gradients.

Looking at how the individual terms in the TD(λ) weight update were converted into their corresponding value-gradient terms, we first note that $\widetilde{G} := \frac{\partial \widetilde{J}}{\partial \vec{x}}$ by definition when a scalar critic is used. Next, it will be proven that $G' \equiv \frac{\partial J'}{\partial \vec{x}}$, either approximately or exactly, depending on the circumstances. This will complete the demonstration of how VGL(λ) was derived from TD(λ) by differentiating each term in the TD(λ) weight update with respect to $\vec{x}$.

The target value gradient, ${G'}_t$ can be shown to be equivalent to $\frac{\partial J'}{\partial \vec{x}}$ as follows. The recursive definition for ${J'}_t$ was given by Eq. (2.3) for non-terminal states as,

$$
{J'}_t := U_t + \gamma\left(\lambda {J'}_{t+1} + (1-\lambda)\widetilde{J}_{t+1}\right).
$$

Since ${J'}_t$ is defined for an arbitrary trajectory, which can be expanded using $\vec{x}_{t+1} = f(\vec{x}_t, A(\vec{x}_t, \vec{e}_t, \vec{z}), \vec{e}_t)$, we can rewrite its recursive definition as

$$
\begin{aligned}
J'(\vec{x}_t, \mathbf{\vec{e}}_t, \vec{w}, \vec{z}) :=& U(\vec{x}_t, A(\vec{x}_t, \vec{e}_t, \vec{z}), \vec{e}_t) + \gamma\big[\lambda J'(f(\vec{x}_t, A(\vec{x}_t, \vec{e}_t, \vec{z}), \vec{e}_t), \mathbf{\vec{e}}_{t+1}, \vec{w}, \vec{z}) \\
&+ (1-\lambda)\widetilde{J}(f(\vec{x}_t, A(\vec{x}_t, \vec{e}_t, \vec{z}), \vec{e}_t), \vec{w})\big].
\end{aligned}
$$

Differentiating this fully with respect to $\vec{x}_t$ (and applying the chain rule, $\vec{u}_t = A(\vec{x}_t, \vec{e}_t, \vec{z})$, $\vec{x}_{t+1} = f(\vec{x}_t, \vec{u}_t, \vec{e}_t)$ and trajectory-shorthand notation) produces,

$$
\begin{aligned}
\left(\frac{\partial J'}{\partial \vec{x}}\right)_t =& \left(\frac{\partial U}{\partial \vec{x}}\right)_t + \left(\frac{\partial A}{\partial \vec{x}}\right)_t \left(\frac{\partial U}{\partial \vec{u}}\right)_t \\
&+ \gamma\left(\left(\frac{\partial f}{\partial \vec{x}}\right)_t + \left(\frac{\partial A}{\partial \vec{x}}\right)_t \left(\frac{\partial f}{\partial \vec{u}}\right)_t\right)\left(\lambda\left(\frac{\partial J'}{\partial \vec{x}}\right)_{t+1} + (1-\lambda)\left(\frac{\partial \widetilde{J}}{\partial \vec{x}}\right)_{t+1}\right) \\
=& \left(\frac{DU}{D\vec{x}}\right)_t + \gamma\left(\frac{Df}{D\vec{x}}\right)_t\left(\lambda\left(\frac{\partial J'}{\partial \vec{x}}\right)_{t+1} + (1-\lambda)\widetilde{G}_{t+1}\right) \\
\approx& \left(\frac{D\bar{U}}{D\vec{x}}\right)_t + \gamma\left(\frac{D\bar{f}}{D\vec{x}}\right)_t\left(\lambda\left(\frac{\partial J'}{\partial \vec{x}}\right)_{t+1} + (1-\lambda)\widetilde{G}_{t+1}\right) \qquad \text{(by Eq. (3.1))}
\end{aligned}
$$

This is the same recursion as the definition for ${G'}_t$ (Eq. (3.7)) for non-terminal states.

For terminal states $\vec{x}_t \in \mathbb{T}$, we have $J'_t := \Phi_t$ and therefore

$$\begin{aligned}\left(\frac{\partial J'}{\partial \vec{x}}\right)_t &= \left(\frac{\partial \Phi}{\partial \vec{x}}\right)_t \\ &\approx \left(\frac{\partial \bar{\Phi}}{\partial \vec{x}}\right)_t && \text{(by Eq. (3.1))} \\ &= G'_t && \text{(by Eq. (3.7), for } \vec{x}_t \in \mathbb{T})\end{aligned}$$

This proves that, for both terminal and non-terminal states,

$$G' \approx \frac{\partial J'}{\partial \vec{x}}, \tag{3.15}$$

and completes the demonstration that VGL(λ) is an approximate differentiated form of TD(λ). The approximation would become an exact equality if one of the three conditions at the end of Section 3.2.1 were satisfied. For example, if the random noise's effect is additive, as defined by Eq. (3.2), then we would have,

$$G' \equiv \frac{\partial J'}{\partial \vec{x}}. \tag{3.16}$$

This relationship makes it clear that whereas TD(λ) attempts to learn *values*, VGL(λ) attempts to learn *value gradients*. This connection between VGL(λ) and TD(λ) also shows in more detail how DHP is related to HDP(TD(0)), i.e. that if you differentiate the target J' for HDP with respect to $\vec{x}$, you obtain the target G' for DHP. This is justified because DHP and HDP(TD(0)) are just special cases of VGL(λ) and TD(λ) with $\lambda = 0$.

3.6 Case Study: Vertical-Lander, Revisited

The Vertical-Lander was described previously in Section 2.8, and the task was to control a vertical-descent spacecraft to land gently. This caused difficulties for the VL methods in that they converged to suboptimal trajectories in the absence of value exploration. It is now shown how the VGL algorithms cope with this exact same task, without value exploration.

The key result is shown in Fig. 3.6. In this figure, the trajectory (the blue curve) matches the optimal trajectory (the green curve) because the approximate value-

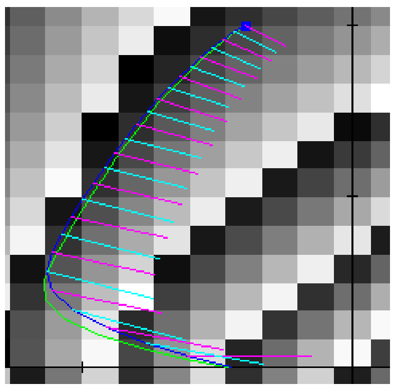


Figure 3.6: Trajectory Optimality under a Greedy Policy, for Value-Gradient Learning. This diagram shows the value gradients, $\widetilde{G}$, matching their targets, ${G'}_t$, well (the lines $\widetilde{G}$ match the lines G' closely in magnitude and direction). Consequently the trajectory (the blue curve) matches the optimal trajectory (the green curve). This fact, that simply learning the target value gradients exactly will produce a locally extremal or optimal trajectory, is a key motivating theorem for VGL, and is proven in Chapter 7 for deterministic environments. In this diagram, the grey-scale background indicates the magnitude of the critic function at each point in state space. The grey scale has changed significantly from the VL-learned value-function shown in Fig. 3.1, since now the value-gradients have been learned successfully. Note that the white bands are not discontinuities in this figure; it is just that the graph plotter ran out of colours and wrapped around from white to black again.

gradients ($\widetilde{G}_t$) match their targets (${G'}_t$) exactly, while operating under a greedy policy. This is a consequence of Pontryagin's Minimum Principle (described in Chapter 4). It happens for any value-gradient learning algorithm, e.g. DHP and VGL(λ), as proven in Chapter 7 for deterministic environments. This figure illustrates the key motivation for VGL methods: simply making the value-gradients match their targets, under a greedy policy, will bend the trajectory into a locally extremal or optimal shape, and hence locally solve the exploration-versus-exploitation dilemma (with respect to value exploration).

3.6.1 Using VGL to Solve the Vertical-Lander Problem

More detailed results for VGL algorithms applied to the Vertical-Lander Problem are now described, for the same set of experiments performed previously for TD(λ), in Section 2.8.2. The experimental details here are largely unchanged from before, except for the following details: For the VGL(λ) experiments, a vector critic was used. Hence the critic network needed three nodes in the output layer. Also, the critic network's final-layer activation function had a linear slope of 20. Learning rates used were $\alpha = 10^{-7}$ for the critic, and $\beta = 0.001$ for the actor.

Using these experimental parameters, the first experiment uses the stochastic-policy method, from a fixed trajectory start point, as described previously in Section 2.8.2. Fig. 3.7 shows results for VGL(1) and VGL(0) both with and without exploration through the stochastic policy (with exploration level σ). In comparison to the equivalent experiment done for TD(λ), shown previously in Fig. 2.2, these results for VGL(λ) show success both with and without value exploration. This is consistent with the principal motivation for VGL methods, in that local value exploration is automatic. Also the VGL(λ) results are many times faster in terms of number of iterations (i.e. by a factor of approximately 50), and in time too (in the implementation used by this thesis, each iteration of VGL took approximately 7% longer than the equivalent TD(λ) iteration). The VGL(λ) results achieve a consistently lower J value than the VL results. When $\sigma = 0$, the VGL(λ) results get the lowest J value of all, since then there is no stochastic distortion of the optimal policy. These were two further motivations for VGL methods.

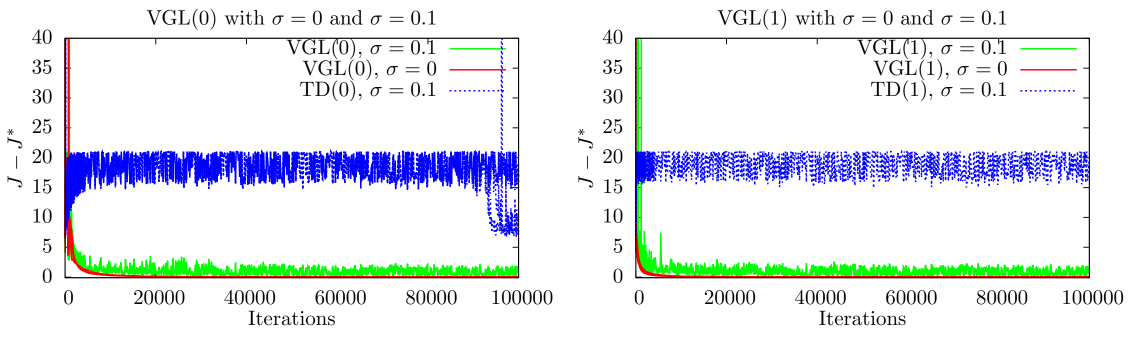


Figure 3.7: Results for Using a Stochastic Policy to Solve the Vertical-Lander Problem by VGL(λ). The results show five trials for each σ value in each graph. For comparison, the equivalent results for TD(λ) taken from Fig. 2.2 are also shown in dotted.

These results also show that large values of λ are less affected by variance in the learning signal. For example, TD(1) failed to solve this problem variant, but the VGL(1) results shown in Fig. 3.7 are just as good, or better, than the VGL(0) results in the same figure. These good results for VGL(1) happen even when $\sigma > 0$, which is the cause of the supposedly-problematic variance, so it initially appears that $\lambda = 1$ does not create problematic variance as much for VGL(λ) as it does for TD(λ); although this is an area for further research.

Fig. 3.8 shows the performance of VGL(λ) in this fixed trajectory-start-point problem across a range of values of λ, and the results show robustness against changes in the value of λ for this problem. In addition, the search for learning rate parameters α and β proved a lot easier for VGL(λ) than for TD(λ).

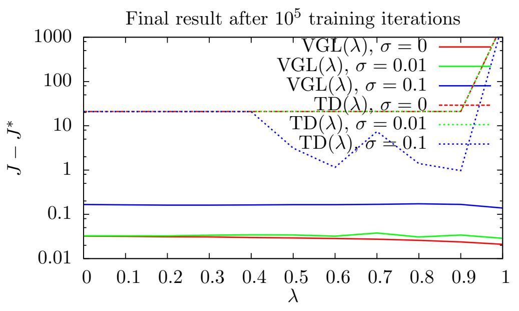


Figure 3.8: Results of VGL(λ) under various λ and noise-levels, σ, for solving the Vertical-Lander Problem, from a fixed trajectory start point. All results are generated for critic learning rate $\alpha = 10^{-6}$ and actor learning rate $\beta = 10^{-3}$. For each value of λ, the graph shows the value of $J - J^*$ after 10^5 training iterations. All results are gathered as the median result of 10 trials (i.e. from 10 different initial randomizations of the neural weights). TD(λ) results from Fig. 2.3 are included as dotted curves, for comparison.

Fig. 3.9 shows results for VGL(λ) applied to the exploring-starts experiments, previously done for TD(λ) in Section 2.8.2. The results for VGL(λ) are more consistent, faster and more stable than the equivalent results for TD(λ) shown in Fig. 2.4. Although more stable than the corresponding results for TD(λ), there is still a minor stability issue present here, i.e. one of the VGL(1) trial curves jumps to a large J value after 50,000 iterations; because as usual with a concurrently-learning actor and critic, convergence is not assured. The methods presented in Chapters 5 and 8 can be used to address this issue, i.e. the use of the algorithm VGLΩ(1) with a greedy policy produces stable proven convergence.

A final important experimental detail here was that clipping was used throughout the experiments in this section, and in the corresponding VL experiments (Section 2.8.2). Clipping involves careful truncation of the final time step of the trajectory, so that the agent stops exactly at $x_h = 0$ and not $x_h < 0$. Clipping is described fully in Chapter 10. Without clipping, the VGL experiments would not give quite so good

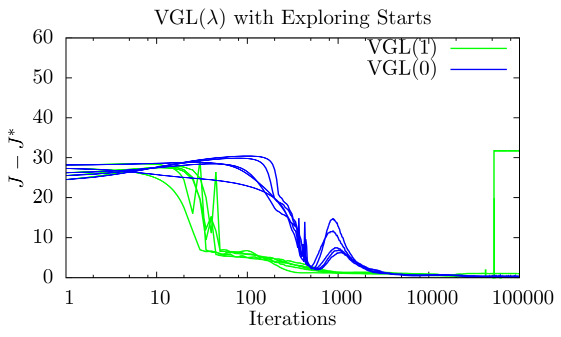


Figure 3.9: Results for Using Exploring Starts to Solve the Vertical-Lander Problem by VGL(λ). Each curve shows a different experimental trial (i.e. different weight initialisation); five different trials for VGL(1) and five trials for VGL(0). (For comparison to VL methods, see also Fig. 2.4.)

results, as detailed in Chapter 10 (e.g. see Fig. 10.6), but the VL results would largely be unaffected. This is a disadvantage of VGL methods compared to VL methods.

3.7 Quadratic-Optimisation Problem

The experiment in this section is designed to be simple and highly demonstrative of the advantages of VGL methods over VL methods, i.e. with respect to learning speed and automatic value exploration. Unlike the previous problem, this problem does not involve any requirement for clipping, and thus is easier to program and replicate the results. A short discussion on model-learning for this experiment is also given, in Section 3.7.1.

An environment is defined with $\vec{x} := x \in \mathbb{R}$ and $\vec{u} := u \in \mathbb{R}$, and model and cost functions:

$$f(x, t, u) := x + u \tag{3.17}$$

$$U(x, t, u) := (u)^2. \tag{3.18}$$

Each trajectory is defined to terminate on arriving at time step $t = 2$, and on termination a final instantaneous cost of

$$\Phi(\vec{x}) := (x)^2 \tag{3.19}$$

is given. The only actions used in the trajectory are u_0 and u_1; the total cost for this trajectory is $(u_0)^2 + (u_1)^2 + (x_0 + u_0 + u_1)^2$, and the theoretical optimal total cost for the whole trajectory is

$$J^* = (x_0)^2/3. \tag{3.20}$$

The action network was a MLP with two inputs, one output and one hidden layer of 4 nodes, short-cut connections from the input layer to the output layer, and with activation function $g(x) = \tanh(x)$ at all nodes. The weights $\vec{z}$ were initially randomised uniformly from the range $[-0.1, 0.1]$. The critic network $\widetilde{J}(\vec{x}, \vec{w})$ was a scalar critic, so that in this implementation, VGL(0) is equivalent to GDHP. The critic was identically dimensioned to the action network, with a weight vector $\vec{w}$ randomised initially in the same way. The activation function used for the critic was $g(x) = \tanh(x)$ at all nodes except for the output node, which used $g(x) = x$. The input vector to each neural network was (x, t).

Each trajectory was made to start at $x_0 = 0.8$. Learning rates for the critic and actor were both $\alpha = 0.1$ and $\beta = 0.1$, respectively, with discount factor $\gamma = 1$. To provide the facility for exploration, Gaussian noise with zero mean and variance σ^2 was added to the output of the action network to form the policy function, i.e. as given by Eq. (2.16).

The critic-learning algorithms tested were TD(1), TD(0), VGL(1) and VGL(0) (Algs. 2.2 and 3.2), all operating in batch mode. The experiments were repeated with noise ($\sigma = 0.01$) and without noise ($\sigma = 0$). Results averaged from 40 trials for each algorithm are shown in Fig. 3.10. The results show that the value-learning method (TD(λ)) could not cope without some random value exploration, but the VGL based methods (GDHP and VGL(λ)) work successfully for both σ values used. Also, in comparison to the VL methods, VGL methods are very fast (approximately 1,000 times faster to obtain an error $J - J^* < 0.1$). On the other hand, VGL methods require prior knowledge of the model functions (in order to use their derivatives), whereas VL methods do not. But when the model functions are available, the increased speed and automatic local value exploration provides a strong motivation to use VGL methods.

The following subsection makes a short discussion of how model-learning would affect the results of this experiment, because otherwise it might not be considered a fair comparison between VGL and VL.

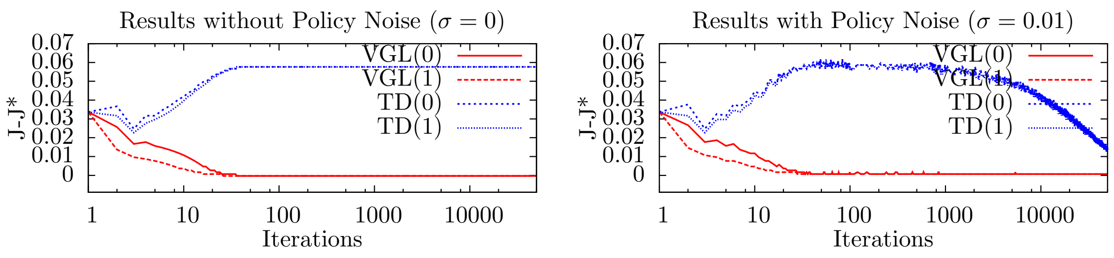


Figure 3.10: Algorithm performances for the quadratic-optimisation problem of Section 3.7, both with and without policy noise. The y axis shows $J - J^*$, where J^* is the optimal trajectory cost given by Eq. (3.20). Compared to the VL method (TD(λ)), the VGL method works well in the absence of stochastic exploration, and quickly attains $J = J^*$. The VL method fails without stochastic exploration here (i.e. it converges to a suboptimal policy), but does learn slowly and successfully in the presence of policy noise.

3.7.1 Model-Learning Details

In this section, an examination is made into the extra cost which would be incurred if model-learning was to be included into the above experiment.

Differentiating Eq. (3.17) gives $\frac{\partial f(\vec{x},\vec{u})}{\partial \vec{x}} = 1$ and $\frac{\partial f(\vec{x},\vec{u})}{\partial \vec{u}} = 1$. These are the only bits of model-knowledge that are required to implement VGL(λ). Hence it makes sense to train a model-network to output these derivatives directly instead of the underlying function $f(\vec{x}, \vec{u})$. Since these derivatives are constant, it can be assumed that they could be learned extremely quickly by a neural network with a decent acceleration algorithm, in maybe just a few iterations. For example, a line-search algorithm could be used to fully train the model-network in just one iteration. Hence the time taken to learn $\frac{\partial f}{\partial \vec{x}}$ and $\frac{\partial f}{\partial \vec{u}}$ would not significantly worsen the performance of VGL compared to VL.

VGL methods also require knowledge of the functions $\frac{\partial U(\vec{x},\vec{u})}{\partial \vec{x}}$, $\frac{\partial U(\vec{x},\vec{u})}{\partial \vec{u}}$ and $\frac{\partial \Phi(\vec{x})}{\partial \vec{x}}$, which for this problem are 0, $2u$ and $2x$, respectively. These are a bit harder to learn than the constant derivatives of $f(\vec{x}, \vec{u})$, however it can be argued that either: a) These are still not significantly demanding to learn; or b) These functions should be known anyway, since it is the job of the experimenter to decide which behaviours to reward or punish, and thus what cost functions to choose. In either case, model-learning does not make a very significant impact into the performance of VGL compared to VL, and therefore this problem is one in which VGL outperforms VL by a very large factor whether model learning is included or not.

The purpose of this section has been to address the fairness of comparing a model-based VGL algorithm against a model-free VL algorithm. The key point of this quadratic-optimisation experiment has been to show that local value exploration comes automatically with VGL, and the speed-up factor can be several orders of magnitude, compared to VL. Neither of these two conclusions is an automatic consequence of comparing a model-based algorithm to a model-free one.

3.8 Chapter Conclusions

The VGL(λ) algorithm has been defined and its relationship to its precursor algorithms DHP and TD(λ) has been explained. VGL(λ) extends the DHP algorithm by introducing a bootstrapping parameter, λ, which can affect learning speed and stability. Also, VGL(λ) can be viewed as a differentiated form of TD(λ); whereas TD methods learn values, VGL methods learn value-gradients.

The motivations for using VGL based methods (including DHP) in comparison to VL methods have been emphasised. These are that local value exploration is automatic; VGL methods can be many times faster than VL methods; and VGL methods work naturally in continuous-valued state spaces. The experiments confirmed these motivations, showing success for VGL in environments when no value exploration is used and where VL methods fail, and showing the speed-ups obtained through VGL methods. However, unlike VL methods, VGL methods require that the model functions are differentiable, and known or learnable.

The following chapters in the thesis will show how to speed up the VGL learning process further and make the convergence process much more reliable. Once the greedy policy is introduced (in Chapter 5), further experiments will be presented which show the massive speed up in learning that can be obtained by a greedy policy and the special Ω_t matrix (Eq. (3.8)) that was defined in this chapter. In Chapter 8 this combination, of greedy policy plus Ω_t matrix, will be proven to modify the critic-learning process into true gradient descent to become a very fast and reliable way to solve the ADPRL problem.

Chapter 4

Bellman's and Pontryagin's Optimality Principles

This chapter serves as an introduction to Pontryagin's optimality principle, and its relationship to Bellman's optimality principle. It also aims to illustrate the connection between VGL methods and Pontryagin's principle, and provide motivations for both.

The chapter starts with a review of Bellman's optimality principle of dynamic programming (Section 4.1), highlighting how the greedy policy can be interpreted as an essential component of Bellman's principle. Then the next section highlights how in continuous-valued state spaces, the greedy policy depends upon the value-gradient (Section 4.2), which therefore motivates the need to learn value gradients, if Bellman's principle is ever to be satisfied.

The importance of learning a value-gradient, in order to achieve optimality, acts as an introduction to Pontryagin's minimum principle, which is an optimality principle analogous to Bellman's principle, but where the condition depends entirely upon a value-gradient term being correct. Pontryagin's minimum principle (PMP) is defined in full, for deterministic discrete time environments, in Section 4.3. Section 4.4 attempts to make a visual explanation of Pontryagin's principle, spelling out its relationship to Bellman's principle. Section 4.5 describes how meeting the value-gradient condition all over state space will lead to global optimality, as opposed to just the local optimality which is often only considered. Section 4.6 reviews the Hamilton-Jacobi-Bellman Equation, which looks confusingly like a value-gradient learning equation, but actually is not one. Finally, Section 4.7 gives chapter conclusions.

4.1 Review of Bellman's Optimality Principle and Dynamic Programming

According to Bellman (1957), the principle of optimality is expressed as: "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." This can be stated mathematically as Bellman's optimality condition (Eq. (2.9)), i.e. there exists an optimal value function $J^*(\vec{x})$ which satisfies:

$$J^*(\vec{x}) = \begin{cases} \min_{\vec{u}\in\mathbb{A}} \mathbb{E}\left(U(\vec{x},\vec{u},\vec{e}) + \gamma J^*(f(\vec{x},\vec{u},\vec{e}))\right) & \text{if } \vec{x} \notin \mathbb{T} \\ \mathbb{E}\left(\Phi(\vec{x},\vec{e})\right) & \text{if } \vec{x} \in \mathbb{T}, \end{cases}$$

and if this optimal value function can be found, then optimal behaviour is determined by following a greedy policy on J^*, and we have therefore solved the ADPRL problem.

Dynamic Programming is the name of a systematic method that goes about finding a function $J^*(\vec{x})$ which solves Bellman's equation (Eq. (2.9)) by doing full state-space sweeps working backwards in time. However since dynamic programming requires full exploration of the state space, it suffers from the *curse of dimensionality.*

4.1.1 Split Form for Bellman's Optimality Condition

It was described in Section 2.4 that the training objectives for the critic network and action network were to achieve, as closely as possible:

1. $\widetilde{J}_t = \mathbb{E}\left({J'}_t\right)$ for all $\vec{x}_t \in \mathbb{S}$.

2. The action network is equivalent to the greedy policy.

It was also proven in Section 2.4 that if the above two goals could be met exactly, then the action network would produce optimal actions and satisfy Bellman's optimality principle. It was also noted earlier that due to the limitations of function approximation, and the nature of approximate dynamic programming, these two objectives may never be met exactly, but they form the objective of the main VL algorithms presented in Chapter 2.

The above two conditions are what we will call the *split form of Bellman's optimality condition.* This split form is more convenient to consider for the ADPRL problem formulation, for the reason that when an action network and a critic network are present, these two statements form the necessary and sufficient conditions for Bellman's Condition to be satisfied exactly. In contrast, Bellman's condition in the form of Eq. (2.9) does not explicitly refer to any action network or critic network, and it contains a min operator which does not appear in the main critic-learning algorithms described in Chapters 2-3.

A conclusion is that when an action network is used for the policy function, its training objective is to obey the greedy policy as closely as possible. This is because obeying the greedy policy is a necessary condition for Bellman's optimality principle. This conclusion is relevant to this thesis because the greedy policy is used to motivate value-gradients (in the next subsection), and it is also used in the theoretical result concerning the optimality of the value-gradient learning objective (Chapter 7).

4.2 A First-Order Taylor-Series Expansion of the Greedy Policy

As described in the previous subsection, the greedy policy needs satisfying if Bellman's optimality principle is to be satisfied (whether a greedy policy or an action network is used in the ADP architecture). Here we examine what information the greedy policy is basing its decisions on, and find that the value-gradient is what "drives" the greedy policy. By doing this we can emphasise the motivation for developing VGL methods, and also gain an intuition into how Pontryagin's Minimum Principle works and relates to Bellman's condition.

In continuous-valued state spaces, the greedy policy can be understood more easily by considering a first-order Taylor-series expansion of the greedy-policy function (Eq. (1.8)):

$$\begin{aligned}
\pi(\vec{x},\vec{w}) &= \arg\min_{\vec{u}\in\mathbb{A}} \mathbb{E}\left(U(\vec{x},\vec{u},\vec{e}) + \gamma\widetilde{J}(f(\vec{x},\vec{u},\vec{e}),\vec{w})\right) && \text{(greedy policy)}\\
&\approx \arg\min_{\vec{u}\in\mathbb{A}} \mathbb{E}\left(U(\vec{x},\vec{u},\vec{e}) + \gamma\widetilde{J}(\vec{x},\vec{w}) + \gamma\left(\frac{\partial\widetilde{J}}{\partial\vec{x}}\right)^T (f(\vec{x},\vec{u},\vec{e})-\vec{x})\right) && \text{(Taylor expansion)}\\
&= \arg\min_{\vec{u}\in\mathbb{A}} \mathbb{E}\left(U(\vec{x},\vec{u},\vec{e}) + \gamma\left(\frac{\partial\widetilde{J}}{\partial\vec{x}}\right)^T f(\vec{x},\vec{u},\vec{e})\right) && \text{(No dependency on } \vec{u}\text{)}
\end{aligned} \tag{4.1}$$

Hence we see the greedy policy at time step t is dependent on $\left(\frac{\partial\widetilde{J}}{\partial\vec{x}}\right)_t$ and not on $\widetilde{J}_t$. We can see from this that changing $\left(\frac{\partial\widetilde{J}}{\partial\vec{x}}\right)_t$ will immediately affect the greedy policy, but changing just $\widetilde{J}_t$ will not. This understanding forms a key motivation for developing VGL algorithms, and understanding PMP.[8]

4.3 Pontryagin's Minimum Principle

Pontryagin's Minimum Principle (PMP), also known in its variant form Pontryagin's Maximum Principle, is one of the two fundamental ideas of optimal control. PMP was formulated in 1956 by the Russian mathematician Lev Semenovich Pontryagin (1908-1988) and is described by Pontryagin et al. (1962). The other fundamental idea of optimal control is dynamic programming, developed in the United States by Bellman (1957). The two methods complement each other in the same way that VGL and VL complement each other.

PMP was originally defined for deterministic environments (Pontryagin et al., 1962), although some works do extend it to certain stochastic environments (E.g. Jacobson and Mayne, 1970, Sec. 5.4.7). In this thesis only the deterministic form will be considered, hence the environment functions will omit the noise arguments $\vec{e}$ when referring to PMP.

Unlike dynamic programming, PMP avoids the curse of dimensionality, as it is an optimality condition that must apply along a single trajectory, as opposed to over the whole state space. PMP is usually defined for continuous-time systems, but for this

[8]This analysis has just been a restatement of the argument given earlier in Fig. 3.1.

thesis it is simpler to only consider its discrete-time version. This is stated concisely by (Todorov, 2006, eq.18) as the following three equations which must be satisfied at all time steps t of a trajectory for the trajectory to be optimal:

$$\vec{x}_{t+1} = f(\vec{x}_t, \vec{u}_t), \tag{4.2a}$$

$$\vec{\lambda}_t = \begin{cases} \left(\frac{\partial U}{\partial \vec{x}}\right)_t + \gamma \left(\frac{\partial f}{\partial \vec{x}}\right)_t \vec{\lambda}_{t+1} & \text{if } \vec{x}_t \notin \mathbb{T}, \\ \left(\frac{\partial \Phi}{\partial \vec{x}}\right)_t & \text{if } \vec{x}_t \in \mathbb{T}, \end{cases} \tag{4.2b}$$

$$\vec{u}_t = \min_{\vec{u} \in \mathbb{A}} \left(U(\vec{x}_t, \vec{u}_t) + \gamma f(\vec{x}_t, \vec{u}_t)^T \vec{\lambda}_{t+1} \right). \tag{4.2c}$$

Here $\vec{\lambda}_t$ is called the costate vector. This also goes under various different names (adjoint vector, influence function, adjoint state, Lagrange multiplier). Equation (4.2b) specifies how the costate vectors are calculated recursively, by working backwards from the end of the trajectory. Eq. (4.2a) states how the state vector $\vec{x}_t$ is calculated recursively, by working forwards along the trajectory. If these forward and backwards passes match up according to actions governed by Eq. (4.2c), then PMP is satisfied.

PMP is a necessary condition for optimality. In other words, all optimal trajectories must satisfy PMP. However it is not a sufficient condition. Satisfying PMP for a trajectory only strictly ensures the trajectory is locally extremal. Extremal here means a local minimum, maximum or saddle point.

The fact that only local extremity is guaranteed sounds like a major disadvantage of PMP compared to Bellman's principle, however the big advantage of PMP is computational efficiency. PMP only needs satisfying along a single trajectory, whereas Bellman's principle needs to apply over the whole of the state space, and thus suffers from the curse of dimensionality.

In referring to the efficiency in solving the continuous-time versions of Eqs. (4.2a) to (4.2c), Todorov (2006) explains "*The remarkable property of the maximum principle is that it is an ordinary differential equation (ODE), even though we derived it starting from a partial differential equation (PDE). An ODE is a consistency condition which singles out specific trajectories without reference to neighbouring trajectories (as would be the case in a PDE) [...] The ODE is a system of* $2\dim(\vec{x})$ *scalar equations subject to* $2\dim(\vec{x})$ *scalar boundary conditions. Therefore we can solve this system with standard boundary-value solvers (such as Matlab's bvp4c function)*".

The concern that PMP is only guaranteed to find locally extremal trajectories is usually not a problem, as solution to the PMP equations usually results in a small distinct set of candidate trajectories, and it is a simple matter to choose the best one (Pontryagin et al., 1962, p.22). Furthermore, it is proven in Chapter 7 of this thesis that if the actions in the candidate trajectory are all either fully on or fully off (a situation known as bang-bang control) then that is a sufficient condition for PMP to deliver locally optimal trajectories. Finally, it is shown this chapter, in Section 4.5, that this limitation of only finding locally extremal trajectories does not apply to VGL when the state space is fully explored.

PMP can be proven using Bellman's equation as a start point (e.g. Todorov, 2006, sec. 3.1) or in discrete-time using Lagrange multipliers (e.g. Todorov, 2006, sec. 3.2). Chapter 7 of this thesis also delivers a discrete-time proof and produces the extra result for bang-bang control guaranteeing local optimality. The proof by Pontryagin et al. (1962) is the most thorough continuous-time analysis, and pays special attention to edge cases where the optimal actions change discontinuously over time.

A difference between PMP and ADPRL is that with PMP you need to solve a new ODE each time you want to find an optimal control from a new start point $\vec{x}_0 \in \mathbb{S}$.[9] In contrast, a fully trained ADPRL action network can deliver approximate optimal control from any start point $\vec{x}_0$, without any further significant calculation.

Limitations of PMP compared to the method of dynamic programming are that PMP is only applicable to deterministic trajectories, and also only where trajectories can be written as differential equations. Therefore it is limited to the functions $f(\vec{x}, \vec{u})$ and $U(\vec{x}, \vec{u})$ being smooth and differentiable. Dynamic programming is also applicable to situations where these functions are stochastic and not smooth.

4.4 The Relationship between Pontryagin and Bellman's Principles

Bellman's and Pontryagin's principles are related by the fact that the costate vector $\vec{\lambda}$ in PMP satisfies $\left(\frac{\partial J^*}{\partial \vec{x}}\right)_t \equiv \vec{\lambda}_t$, where $J^*(\vec{x})$ is Bellman's optimal value function. The

[9] Except in certain rare circumstances where it is possible to find exact analytical solutions to the value function and to the control function across the whole of $\mathbb{S}$. For example, see (Pontryagin et al., 1962, chap. 1, sec. 5).

following subsection aims to illustrate an intuition for understanding this connection, and understanding PMP from a Bellman perspective.

4.4.1 Pontryagin's Minimum Principle: A Visual Guide

Consider an optimal value-function surface, and an optimal trajectory embedded within it, such as is illustrated in the left diagram of Fig. 4.1. In trying to narrow down the amount of value-function surface which we must learn in order to discover this optimal trajectory, can we just learn the portion of the surface directly below the trajectory? This would leave a roller-coaster shaped object, as illustrated in the right diagram of Fig. 4.1.

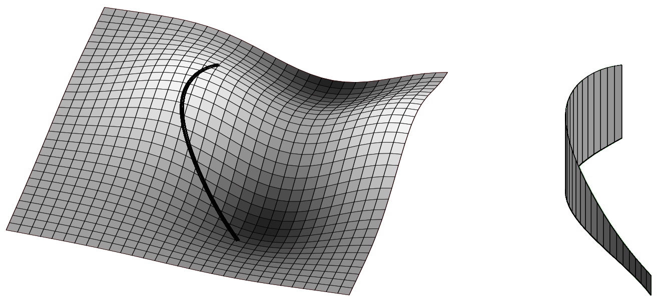


Figure 4.1: The left image shows a critic-function surface and an optimal trajectory curve embedded in it. The right-hand image illustrates that the slice of the critic function that lies directly below the trajectory would take the shape of a roller-coaster track.

The answer to this question is that just learning the roller-coaster's height profile is not enough. The greedy policy needs gradient information too, as was illustrated in Sec. 4.2. In the roller-coaster analogy, this requirement for gradient information would mean that the coaster track must have the correct shear too, as is illustrated in Fig. 4.2. That then gives us the least possible portion of the value-function surface in order to decide whether our trajectory is locally optimal or not.

In PMP, the costate vector, $\vec{\lambda}_t$, holds this "shear" information. The second PMP condition (Eq. (4.2b)) enforces the correct shear of the roller-coaster. The trajectory must simultaneously be found by a greedy policy (i.e. the third PMP condition, Eq. (4.2c)), and also the trajectory must be unrolled by the model function $f(\vec{x}, \vec{u})$ (i.e. the first PMP condition, Eq. (4.2a)). Hence these three conditions combine to give PMP.

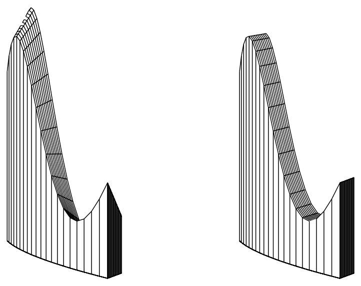


Figure 4.2: Roller-Coasters With and Without Track Shear. The left-hand image shows what a roller coaster with shear would look like. The right-hand image shows a track without shear, for comparison.

Note that in PMP, and likewise in VGL, it not sufficient to take a given fixed trajectory found by the greedy policy and fully learn the costate vectors (value-gradients) along it, in the hope of achieving optimality. By the time the costate vectors have changed, the third condition of PMP will be violated. We need to learn the costate vectors along the trajectory while simultaneously making the trajectory greedy with respect to those costate vectors. Doing both of these things at once will force the trajectory to bend around a lot. Only when this bending has settled down, and both of last two PMP conditions are achieved at once, does PMP apply; and only then can the trajectory be locally optimal.

4.5 The Connection of PMP to VGL and Global Control

VGL methods are very closely related to PMP in that both attempt to learn an optimal value gradient. If the VGL objective is attained all along a trajectory found by a greedy policy, i.e. if $\widetilde{G}_t = \mathbb{E}\left(G'_t\right)$ for all t, then $\widetilde{G}_t$ will become equal to $\vec{\lambda}_t$ for all t along that trajectory. This can be understood initially by observing the similarity between the recursion defining G'_t in DHP (Eq. (3.4)) and the recursion defining $\vec{\lambda}_t$ (Eq. (4.2b)). This equivalence is proven more rigorously in Chapter 7, where further details are given too (see Section 7.3.3). This means VGL can gain the efficiency benefits of PMP if just one trajectory is learned, i.e. meeting the VGL objective along one greedy trajectory is a sufficient condition for the trajectory to be locally extremal (this was demonstrated in Section 3.6, and is proven in more detail in Chapter 7).

However, PMP and VGL differ in that VGL attempts to learn a global function $\widetilde{G}(\vec{x},\vec{w})$ over the whole state space $\mathbb{S}$, but PMP deals with the $\vec{\lambda}_t$ values along a single trajectory. This means that if VGL learns the entire $\widetilde{G}(\vec{x},\vec{w})$ function for all $\vec{x} \in \mathbb{S}$ then it will have recovered Bellman's optimality condition, and thus have achieved global optimality, by the following reasoning. First we assume the environment is deterministic, therefore the VGL learning objective (i.e. to get as close as possible to achieving $\widetilde{G}_t = \mathbb{E}\left(G'_t\right)$ for all t) can omit the expectation symbol, and leads to the following:

$$\begin{aligned}
&\widetilde{G}(\vec{x},\vec{w}) = G'(\vec{x},\vec{w},\vec{z}) \quad \forall \vec{x} \in \mathbb{S} && && (4.3)\\
\Leftrightarrow\ &\frac{\partial \widetilde{J}(\vec{x},\vec{w})}{\partial \vec{x}} \approx \frac{\partial J'(\vec{x},\vec{w},\vec{z})}{\partial \vec{x}} \quad \forall \vec{x} \in \mathbb{S} && \text{(by Eqs. (3.3), (3.15))} &&\\
\Leftrightarrow\ &\widetilde{J}(\vec{x},\vec{w}) \approx J'(\vec{x},\vec{w},\vec{z}) + c \quad \forall \vec{x} \in \mathbb{S} && \text{(integrating the PDE)} && (4.4)
\end{aligned}$$

Solving the differential equation of the final line of Eq. (4.4) introduced an arbitrary constant $+c$. This arbitrary constant, c, is not important as it does not affect the greedy policy, which is invariant to constant increments of this kind. The approximation symbol in Eq. (4.4) could be changed to an equality (via Eq. (3.16)) if one of the three conditions for Eq. (3.16) were satisfied. Together with the greedy policy, this completes the two conditions required for Bellman's optimality condition to be satisfied (Sec. 4.1.1), and therefore we have recovered global optimal control over the whole state space (approximately or exactly; depending on the status of Eq. (3.15)).

The fact that the above conclusion is sometimes only an approximation is not a significant limitation of this VGL global-control proof. For example, if the conditions stated underneath Eq. (3.15) were satisfied (e.g. under conditions of additive noise, as defined by Eq. (3.2)), then the above VGL global-control proof would become exact. In other circumstances, when these conditions for exactness are not met, then Bellman's condition will only be approximately met. But it will be approximately met over the whole of the state-space, so all trajectories would presumably be approximately globally optimal as opposed to approximately locally optimal. So in either case, the conclusion that trajectories will be (approximately) globally optimal, and not (approximately) locally optimal, if the VGL condition is fully satisfied over all of $\mathbb{S}$, still seems to be valid.

This completes the analysis for deterministic environments. The situation for a stochastic environment is proven in the chapter appendix (Section 4.A).

This section has shown that learning a value gradient over the whole state space recovers Bellman's optimality principle (approximately or exactly), and therefore it is a sufficient condition for global optimality, i.e. it shows that VGL overcomes the "locally extremal" caveat that is associated with PMP.

4.6 The HJB Equation

This section attempts to clear up a common misconception of the similarity between the well-known HJB equation (described below) and the VGL method. Although the HJB equation contains a VGL term, it is argued in this section that learning algorithms based directly on the HJB equation will be doing VL and not VGL.

The Hamilton–Jacobi–Bellman (HJB) equation applies to a continuous-time ADPRL problem. It will be described here in its deterministic form, hence the noise vector $\vec{e}$, and the expectation operator $\mathbb{E}()$, will be omitted from all functions in this subsection. In a continuous-time environment the state evolution equation changes from the discrete-time equation (Eq. (1.1)) into a differential equation:

$$\frac{\partial \vec{x}(t)}{\partial t} = \underline{f}(\vec{x}(t), \vec{u}(t))$$

and the total un-discounted trajectory-cost function changes from a discrete summation (Eq. (1.4)) into an integral:

$$J(\vec{x}_t, \vec{z}) = \int_t^T \underline{U}(\vec{x}(s), \vec{u}(s))ds$$

where $\underline{f}$ and $\underline{U}$ are function names given to the continuous-time counterparts to the discrete-time functions $f(\vec{x}, \vec{u})$ and $U(\vec{x}, \vec{u})$.

In the continuous-time ADPRL problem, the HJB equation is:

$$\frac{\partial J^*(\vec{x}, t)}{\partial t} + \min_{\vec{u}} \left(\left(\frac{\partial J^*(\vec{x}, t)}{\partial \vec{x}} \right)^T \cdot \underline{f}(\vec{x}(t), \vec{u}) + \underline{U}(\vec{x}(t), \vec{u}) \right) = 0$$

The reason the HJB equation is often confused for a VGL equation is that it contains a value-gradient term, $\frac{\partial J^*(\vec{x},t)}{\partial \vec{x}}$. However there are two reasons that the HJB equation is not a VGL equation:

1. The HJB equation is not a weight update at all. It is an optimality condition that must be true over the whole state space to imply Bellman's Condition.

2. If a variant on HJB was used as the basis for a weight update, then the vector inner product with $\underline{f}(\vec{x}(t), \vec{u})$ results in a projection of the value-gradient along the trajectory's direction. This means that learning via a variant of the HJB equation is not going to learn the value-gradient directions that are necessary for obviating local value exploration. To use the roller-coaster analogy again from Sec. 4.4.1, this projection exactly removes all of the track-shear information, and retains only the track's height profile.

4.7 Chapter Conclusions

This chapter has provided a review of Bellman's and Pontryagin's optimality principles, including details of how they relate to each other. By taking a starting point of Bellman's principle, which is the founding equation of much of RL, the chapter has argued that in continuous-valued state spaces, Bellman's principle depends upon a greedy policy, and a greedy policy depends upon value gradients, and therefore learning value-gradients is necessary for optimality.

PMP has been described as an optimality principle which specifies that correct value-gradients are a necessary condition for trajectory optimality. The chapter has described briefly how PMP relates to value-gradient learning algorithms, and how learning value-gradients all over the state space will act as a sufficient condition for global optimality.

Finally the HJB equation has been described, but only to point out that it is not a true "value-gradient learning" equation.

4.A Chapter Appendix: Extension of VGL Global-Control Proof to Stochastic Environments

This appendix forms an extension to Section 4.5. The derivation of Eq. (4.4) from Eq. (4.3) made an assumption of a deterministic environment. This appendix shows how to make the same deduction for stochastic environments. Hence, the objective of this appendix is to prove that

$$\widetilde{G}(\vec{x},\vec{w}) = \mathbb{E}\left(G'(\vec{x},\vec{\mathbf{e}},\vec{w},\vec{z})\right) \quad \forall\vec{x}\in\mathbb{S} \tag{4.5}$$

leads to

$$\widetilde{J}(\vec{x},\vec{w}) = \mathbb{E}\left(J'(\vec{x},\vec{\mathbf{e}},\vec{w},\vec{z})\right) + c \quad \forall\vec{x}\in\mathbb{S}. \tag{4.6}$$

and hence satisfaction of the Bellman Condition for globally-optimal control.

In a stochastic environment, the trajectory depends on the unknown noise vector $\vec{\mathbf{e}}$, which was defined in Section 1.2 to represent all of the random numbers used in generating the entire stochastic trajectory. This noise vector has a probability distribution $P_{\mathbf{e}}(\vec{\mathbf{e}})$ and sample space $\mathbf{E}$. Hence the expectation operator applied to any function f of the noise vector $\vec{\mathbf{e}}$ is defined by the following integral:

$$\mathbb{E}\left(f(\vec{\mathbf{e}})\right) := \int_{\mathbf{E}} P_{\mathbf{e}}(\vec{\mathbf{e}})\, f(\vec{\mathbf{e}}) d\vec{\mathbf{e}}, \tag{4.7}$$

with

$$\int_{\mathbf{E}} P_{\mathbf{e}}(\vec{\mathbf{e}})\, d\vec{\mathbf{e}} = 1. \tag{4.8}$$

Using this notation, we can now show how Eq. (4.5) leads to Eq. (4.6), as follows.

$$\begin{aligned}
\widetilde{G}(\vec{x},\vec{w}) &= \mathbb{E}\left(G'(\vec{x},\vec{\mathbf{e}},\vec{w},\vec{z})\right) \quad \forall\vec{x}\in\mathbb{S} && \text{(by Eq. (4.5))}\\
&= \int_{\mathbf{E}} P_{\mathbf{e}}(\vec{\mathbf{e}})\, G'(\vec{x},\vec{\mathbf{e}},\vec{w},\vec{z}) d\vec{\mathbf{e}} \quad \forall\vec{x}\in\mathbb{S} && \text{(by Eq. (4.7))}
\end{aligned}$$

Integrating both sides with respect to $\vec{x}$, across the whole state space $\mathbb{S}$, gives,

$$\begin{aligned}
\int_{\mathbb{S}} \widetilde{G}(\vec{x},\vec{w}) d\vec{x} &= \int_{\mathbb{S}} \left(\int_{\mathbf{E}} P_{\mathbf{e}}(\vec{\mathbf{e}})\, G'(\vec{x},\vec{\mathbf{e}},\vec{w},\vec{z}) d\vec{\mathbf{e}}\right) d\vec{x} \quad \forall\vec{x}\in\mathbb{S}\\
&= \int_{\mathbf{E}} \left(\int_{\mathbb{S}} P_{\mathbf{e}}(\vec{\mathbf{e}})\, G'(\vec{x},\vec{\mathbf{e}},\vec{w},\vec{z}) d\vec{x}\right) d\vec{\mathbf{e}} \quad \forall\vec{x}\in\mathbb{S} && \text{(swap integration order)}\\
&= \int_{\mathbf{E}} P_{\mathbf{e}}(\vec{\mathbf{e}}) \left(\int_{\mathbb{S}} G'(\vec{x},\vec{\mathbf{e}},\vec{w},\vec{z}) d\vec{x}\right) d\vec{\mathbf{e}} \quad \forall\vec{x}\in\mathbb{S} && \text{(constant term extracted)}
\end{aligned}$$

$$\Rightarrow \int_{\mathbb{S}} \frac{\partial \widetilde{J}(\vec{x},\vec{w})}{\partial \vec{x}} d\vec{x} \approx \int_{\mathbf{E}} P_{\mathbf{e}}(\vec{\mathbf{e}}) \left(\int_{\mathbb{S}} \frac{\partial J'(\vec{x},\vec{\mathbf{e}},\vec{w},\vec{z})}{\partial \vec{x}} d\vec{x} \right) d\vec{\mathbf{e}} \quad \forall \vec{x} \in \mathbb{S} \quad \text{(by Eqs. (3.3), (3.15))}$$

$$\begin{aligned}
\Rightarrow \quad \widetilde{J}(\vec{x},\vec{w}) &\approx \int_{\mathbf{E}} P_{\mathbf{e}}(\vec{\mathbf{e}}) \big(J'(\vec{x},\vec{\mathbf{e}},\vec{w},\vec{z}) + c \big) d\vec{\mathbf{e}} \quad \forall \vec{x} \in \mathbb{S} \\
&= \int_{\mathbf{E}} P_{\mathbf{e}}(\vec{\mathbf{e}})\, J'(\vec{x},\vec{\mathbf{e}},\vec{w},\vec{z}) d\vec{\mathbf{e}} + c \int_{\mathbf{E}} P_{\mathbf{e}}(\vec{\mathbf{e}}) d\vec{\mathbf{e}} \quad \forall \vec{x} \in \mathbb{S} \quad \text{(constant term extracted)} \\
&= \mathbb{E}\big(J'(\vec{x},\vec{\mathbf{e}},\vec{w},\vec{z}) \big) + c \quad \forall \vec{x} \in \mathbb{S} \quad \text{(by Eqs. (4.7) and (4.8))}
\end{aligned}$$

The line which exchanges the order of integration needs further justification. For this to be valid, by Fubini's theorem from mathematical analysis, we require that the integrand (i.e. $P_{\mathbf{e}}(\vec{\mathbf{e}})\, G'(\vec{x},\vec{\mathbf{e}},\vec{w},\vec{z})$) is finite and defined everywhere in $\mathbb{S}$ and $\mathbf{E}$. Since the initial assumption was that $\mathbb{E}(G') \equiv \widetilde{G}$ for all $\vec{x} \in \mathbb{S}$, it is reasonable to assume that $G'(\vec{x},\vec{\mathbf{e}},\vec{w},\vec{z})$ is always well-defined and finite. Therefore, for the product $P_{\mathbf{e}}(\vec{\mathbf{e}})\, G'(\vec{x},\vec{\mathbf{e}},\vec{w},\vec{z})$ to be finite everywhere, we require that $P_{\mathbf{e}}(\vec{\mathbf{e}})$ is always finite. If there were any infinite spikes in $P_{\mathbf{e}}(\vec{\mathbf{e}})$, then the random variable $\vec{\mathbf{e}}$ must have been a discrete random variable instead of a continuous one. So assuming that $\vec{\mathbf{e}}$ is a well-behaved continuous random variable, then the proof that Eq. (4.6) is a consequence of Eq. (4.5) is complete.

If $\vec{\mathbf{e}}$ was a discrete random variable, then the proof could still be made by rewriting the expectation equation (Eq. (4.7)) as $\mathbb{E}(f(\vec{\mathbf{e}})) := \sum_S P_{\mathbf{e}}(\vec{\mathbf{e}})\, f(\vec{\mathbf{e}})$. The result would then easily follow, since the $\sum$ and $\int$ signs always commute.

Like in the equivalent deterministic proof given in Section 4.5, the effect of the approximation symbol is not that significant, and should not affect the global versus local conclusion; only whether the Bellman equation is satisfied exactly or approximately.

Chapter 5

Action-Network Weight-Update Methods, and Efficient Greedy Policies

In this chapter, various alternative action-network weight-update schemes are described (in Sec. 5.1). However, in certain situations, the action network's weight update can be completely removed and the ADPRL architecture of two neural networks can be reduced down to just one. In this case a greedy policy would replace the action network, as described in Sec. 5.2. This simpler architecture can be beneficial since the learning process for a single network can be accelerated more easily, as we no longer need worry about the effects of two mutually interacting networks being trained simultaneously. In this chapter, greedy-policy functions are derived which are applicable to continuous-valued state spaces, with differentiable environment functions, and work under the assumption that the model function, $\bar{f}(\vec{x}, \vec{u})$, is linear in $\vec{u}$. Most of these policies (with one exception, in Section 5.2.1) require continuous-valued action spaces.

The greedy policies derived in this chapter are closed-form solutions for minimising the model-based approximate Q function, defined earlier in Eq. (3.9), as:

$$\widetilde{Q}(\vec{x}, \vec{u}, \vec{w}) := \bar{U}(\vec{x}, \vec{u}) + \gamma \widetilde{J}(\bar{f}(\vec{x}, \vec{u}), \vec{w}). \tag{5.1}$$

Hence the closed-form greedy policies in this chapter are of the form:

$$\pi(\vec{x},\vec{w}) = \arg\min_{\vec{u}\in\mathbb{A}} \left(\widetilde{Q}(\vec{x},\vec{u},\vec{w})\right). \tag{5.2}$$

Here the functions $\bar{f}(\vec{x},\vec{u})$ and $\bar{U}(\vec{x},\vec{u})$ are the deterministic learned model and cost functions, defined in Eq. (1.9). If these are learned perfectly then they would equal the expectations of the true stochastic model and cost functions, $\mathbb{E}\left(f(\vec{x},\vec{u},\vec{e})\right)$ and $\mathbb{E}\left(U(\vec{x},\vec{u},\vec{e})\right)$, respectively (as described by Eq. (1.10)). Hence the minimisation in Eq. (5.2) is slightly different from the one required for the true ADPRL greedy policy (Eq. (1.8)), since the true model and cost functions have been replaced by $\bar{f}(\vec{x},\vec{u})$ and $\bar{U}(\vec{x},\vec{u})$. This replacement is equivalent to moving the expectation operator that appears in Eq. (1.8) to a different position. Therefore, in general stochastic environments, Eq. (5.2) is only an approximation to the true ADPRL greedy policy. In deterministic environments, however, it is exact, because then the expectation operators have no effect.

To clarify the distinction between Eq. (5.2) and Eq. (1.8), the greedy policy given by Eq. (5.2) will be referred to as being *greedy on the model-based approximate Q function*, or the "greedy-on-$\widetilde{Q}$" policy, for short. Note that since the model-based $\widetilde{Q}$ function uses the learned $\bar{f}(\vec{x},\vec{u})$ and $\bar{U}(\vec{x},\vec{u})$ functions, which themselves are deterministic (since they are independent of $\vec{e}$), this $\widetilde{Q}$ function is therefore deterministic.

It is preferable to use the learned model and cost functions in Eqs. (5.3) and (5.2) over the stochastic ones, for the practical reason that their derivatives are known and independent of $\vec{e}$ (which may not be observable). This is one reason why Eq. (5.2) is used as the basis for the greedy policies presented in this chapter. The other reason is that it sometimes has a simple closed-form solution, which makes it computationally efficient to work with. However its use comes with the caveat that it will only be an approximation to the true ADPRL greedy policy, in general stochastic environments.

Empirical results comparing performance with an action network to performance with he above greedy policy are given in Sec. 5.3. These results show that a massive speed-up of learning can take place with a greedy policy, and also make a demonstration of the effectiveness of the special Ω_t matrix that was previously defined in Eq. (3.8).

5.1 Action-Network Weight Updates

The ADPRL algorithms presented so far are for training a critic function. To make the agent behave optimally, the action-network function $A(\vec{x}, \vec{e}, \vec{z})$ also needs training. The HDP, TD(λ), DHP and VGL(λ) pseudocode presented in Chapters 2-3 were presented in actor-critic form, i.e. including an action-network weight update (in addition to the main critic weight update). The action-network weight update used was the following standard ADP one:

$$\Delta\vec{z} = -\beta\left(\frac{\partial A}{\partial\vec{z}}\right)_t\left(\left(\frac{\partial\bar{U}}{\partial\vec{u}}\right)_t + \gamma\left(\frac{\partial\bar{f}}{\partial\vec{u}}\right)_t\widetilde{G}_{t+1}\right), \tag{5.3}$$

where $\beta > 0$ is a learning rate. This action-network weight update appears in line 7 of the DHP algorithm (Alg. 3.1), in line 9 of VGL(λ) (Alg. 3.2), in line 7 of TD(0)/HDP (Alg. 2.1); and also in (Prokhorov and Wunsch, 1997a, Eq. 10).

This action-network weight update is model based, because it uses the derivatives of the learned model function $\bar{f}(\vec{x}, \vec{u})$. This makes it potentially a very fast learning algorithm. It is equivalent to $\Delta\vec{z} = -\beta\left(\frac{\partial A}{\partial\vec{z}}\right)_t\left(\frac{\partial\widetilde{Q}}{\partial\vec{u}}\right)_t$, which is direct gradient descent on $\widetilde{Q}(\vec{x}_t, A(\vec{x}_t, \vec{e}_t, \vec{z}), \vec{w})$ with respect to $\vec{z}$, where $\widetilde{Q}$ is the model-based approximate Q function (Eq. (5.1)).

Consequently the objective of the above weight update is to make the action network eventually satisfy, as closely as possible,

$$\mathbb{E}\left(A(\vec{x}, \vec{e}, \vec{z})\right) = \arg\min_{\vec{u}\in\mathbb{A}}\left(\widetilde{Q}(\vec{x}, \vec{u}, \vec{w})\right) \quad \forall\vec{x},$$

which is equivalent to the greedy-on-$\widetilde{Q}$ policy (Eq. (5.2)).

Many other weight updates for the action network are possible. Some important ones are illustrated in Table 5.1. All of them try to make the action network eventually satisfy the greedy policy as closely as possible, which as described in Sec. 4.1.1, is necessary for Bellman's Optimality Principle to apply. Some of these weight updates are model based, and some are model free. The model-free ones all require that the policy function $A(\vec{x}, \vec{e}, \vec{z})$ is stochastic, as detailed in Table 5.1. The two model-free equations do not involve the learned deterministic functions $\bar{f}(\vec{x}, \vec{u})$ and $\bar{U}(\vec{x}, \vec{u})$, and therefore are more accurate in stochastic environments than the model-based equations presented in the table.

Equation	Model Free?
DHP Standard Action-Network Weight Update (Prokhorov and Wunsch, 1997a, eq. 10): $\Delta\vec{z} = -\beta \sum_t \left(\frac{\partial A}{\partial \vec{z}}\right)_t \left(\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \widetilde{G}_{t+1}\right).$	No
Learn to copy the greedy policy (Ferrari and Stengel, 2004): $\Delta\vec{z} = \beta \sum_t \left(\frac{\partial A}{\partial \vec{z}}\right)_t (\pi(\vec{x}_t, \vec{w}) - A(\vec{x}_t, \vec{e}_t, \vec{z}))$ where $\pi(\vec{x}_t, \vec{w})$ denotes the greedy policy.	No
Stochastic real-valued (SRV) unit algorithm (Doya, 2000; Gullapalli, 1990) $\Delta\vec{z} = -\beta n_t \left(\frac{\partial A}{\partial \vec{z}}\right)_t (U_t + \gamma \widetilde{J}_{t+1} - \widetilde{J}_t)$, where actions are chosen by $\vec{u}_t = A(\vec{x}_t, \vec{z}) + n_t$, where n_t is a random variable with mean 0 and standard deviation ϵ.	Yes
A TD action-network weight update, modified from (Sutton and Barto, 1998, ch.6.6), where $P_u(\vec{u}\vert\vec{x}, \vec{z})$ denotes the probability density function (parameterised by a weight vector $\vec{z}$) for choosing action $\vec{u}$ from state $\vec{x}$. $\Delta\vec{z} = -\beta \frac{\partial P_u(\vec{u}\vert\vec{x},\vec{z})}{\partial \vec{z}} \frac{(U_t + \gamma \widetilde{J}_{t+1} - \widetilde{J}_t)}{P_u(\vec{u}\vert\vec{x},\vec{z})}.$	Yes

Table 5.1: Some Common ADPRL Action-Network Weight Updates

5.1.1 Organisation of Concurrent Critic / Action-Network Weight Updates

The action-network weight update can be done concurrently with the critic weight update, or by iteratively doing several critic weight updates followed by several action-network weight updates. These schemes both come under the classification of generalised policy iteration (Sec. 2.7.5).

One form of generalised policy iteration is to train the critic network to completion in between every single action-network weight update (henceforth referred to as *policy iteration with a perfectly trained critic*). An opposite scheme to this is *value iteration*, which means training the action network to completion in between every single critic-network weight update. The pseudocode presented in this thesis (e.g. DHP and VGL(λ)) have had a separate learning rate for the critic network and the action network, so represent a blending between these two opposite schemes. The pseudocode would need modifying to implement either one of the opposite schemes exactly.

Policy iteration with a perfectly trained critic (PIPTC), or approximations to this scheme, are popular in the RL literature, in order to take advantage of the convergence results described in Section 2.7.5. As already described in Section 2.7.5, two drawbacks with these convergence results are the high computational expense of PIPTC, and the requirement for the critic and action-network's function approximators to be "compatible".

In value iteration, the innermost loop is to train the action network to completion. Since the objective of action-network training is to make the action network emulate a greedy policy (as proven in Section 4.1.1), in some circumstances we may as well omit the action network altogether, and just use the greedy policy directly, via Eq. (1.8). To use this equation, we need knowledge of the model and cost functions, so this is a model-based approach. Furthermore, the greedy policy is only practical when it is efficient to compute, which is possible in certain situations described in the next section.

PIPTC and value iteration have relative pros and cons. From a model-based point of view, value iteration seems more desirable, for two reasons. Firstly, to overcome the slowness concerns about PIPTC. And secondly, to take advantage of the fact that value iteration is equivalent, in some circumstances, to the efficient model-based greedy-policy methods described in this chapter.

Using the greedy policy (and/or value iteration) can make convergence analysis more difficult, since the critic-learning algorithm is now dealing with a changing policy function. This is not the case with the PIPTC scheme, where the action network is held fixed during the critic-training process. Consequently, a greedy policy can make some critic-learning algorithms diverge, even though they were proven to converge with a fixed action network (such as TD(1) and VGL(1); see Chapter 9 for examples). On the other hand though, the greedy policy does behave very predictably, which can sometimes benefit convergence analysis. Convergence analysis for a critic-learning algorithm combined with a greedy policy is difficult, but an example is provided for VGLΩ(1) in Chapter 8.

This completes the summary of various action-network training methods, and the description of the model-based benefits of using a greedy policy. The next section describes some efficient implementations of a greedy policy.

5.2 Closed-Form Greedy-Policy Solutions

In this section closed-form solutions to the greedy-on-$\widetilde{Q}$ equation (Eq. (5.2)) are derived. Therefore for general stochastic environments, the greedy policies derived in this chapter are only approximations to the true ADPRL greedy policy, defined by Eq. (1.8). It is proven in Appendix 5.A that in one specific kind of stochastic environment, where the effect of the noise is independent of the action chosen, the greedy-on-$\widetilde{Q}$ policy becomes exactly equivalent to the true ADPRL greedy policy. Also in deterministic environments, the greedy-on-$\widetilde{Q}$ policy is always equivalent to the true ADRPL greedy policy.

There are several alternative possibilities for a closed-form greedy policy of this kind. All of them rely on the value gradient; all of them are model-based; and all them are only applicable when the function $\bar{f}(\vec{x}, \vec{u})$ is linear in $\vec{u}$, and when the effect of the stochastic noise is independent of $\vec{u}$. These preconditions allow the closed-form solutions to be found. If the preconditions are not satisfied then the closed-form solutions would not be valid, and numerical solution methods would have to be used instead; but these are not described in this thesis.

The first two policies (Section 5.2.1 and Section 5.2.2) are first-order approximations to the greedy policy, and become increasingly accurate approximations in continuous-time system dynamics. The second of these two relies on a mathematical technique proposed by Doya (2000) for generating sigmoidal control. The third policy, described in Sec. 5.2.3, works well in discrete-time environments, but is only theoretically justified when a vector critic (see Section 3.4.1) is used. This extends the method of Heydari and Balakrishnan (2011) to be applicable to non-linear function approximators and also to VGL(λ). Finally, a quadratic-cost greedy policy which is commonly used in deterministic control theory is described in Section 5.2.4.

5.2.1 First-Order Bang-Bang Greedy Policy

We saw in Section 4.2 that a first-order Taylor-series expansion of the greedy policy gives Eq. (4.1), i.e.

$$\pi(\vec{x}_t, \vec{w}) \approx \arg\min_{\vec{u}_t \in \mathbb{A}} \mathbb{E} \left(U(\vec{x}_t, \vec{u}_t, \vec{e}_t) + \gamma \left(\frac{\partial \widetilde{J}}{\partial \vec{x}} \right)_t^T f(\vec{x}_t, \vec{u}_t, \vec{e}_t) \right).$$

When the learned model and cost functions are used, as is the case when minimising the model-based approximate Q-function (which is the objective for this chapter, i.e. to solve Eq. 5.2), then the expectation operator can be dropped. This gives,

$$\pi(\vec{x}_t, \vec{w}) \approx \arg\min_{\vec{u}_t \in \mathbb{A}} \left[\bar{U}(\vec{x}_t, \vec{u}_t) + \gamma \left(\frac{\partial \widetilde{J}}{\partial \vec{x}} \right)_t^T \bar{f}(\vec{x}_t, \vec{u}_t) \right]. \tag{5.4}$$

Although this is only an approximation to the true greedy-on-$\widetilde{Q}$ policy, the approximation becomes increasingly accurate as the sampling time of the underlying system dynamics ($\Delta\tau$) tends to zero. The approximation becomes exact in this continuous-time limit ($\Delta\tau \to 0$). So it is practical to work with this first-order approximation, and for this reason we treat it as an equality for the basis of a closed-form greedy policy. In Section 5.2.3 this approximation will be improved into a true equality even when $\Delta\tau$ is finite.

If the functions $\bar{U}(\vec{x}, \vec{u})$ and $\bar{f}(\vec{x}, \vec{u})$ are both linear in $\vec{u}$ then the above minimum can be written as:

$$\pi(\vec{x}_t, \vec{w}) \approx \arg\min_{\vec{u}_t \in \mathbb{A}} \left(\vec{u}^T \left(\left(\frac{\partial \bar{U}}{\partial \vec{u}} \right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}} \right)_t \left(\frac{\partial \widetilde{J}}{\partial \vec{x}} \right)_t \right) \right). \tag{5.5}$$

Due to the linearity in $\vec{u}$, the derivatives $\frac{\partial \bar{f}}{\partial \vec{u}}$ and $\frac{\partial \bar{U}}{\partial \vec{u}}$ are constant in the above expression. Similarly, $\left(\frac{\partial \widetilde{J}}{\partial \vec{x}} \right)_t$ is constant with respect to $\vec{u}_t$. Hence the choice of $\vec{u}_t$ which minimises the above expression will lie on the boundary of $\mathbb{A}$. Furthermore, if $\mathbb{A} = \mathbb{R}^n$ then there is no boundary to $\mathbb{A}$, and the actions chosen by Eq. (5.5) will be infinite, which is not realistic. Consequently, with this policy and the assumed linearity in $\vec{u}$, we cannot have an unbounded action space. Therefore we require that either the action space is bounded (and specifically we require that $\mathbb{A} \neq \mathbb{R}^n$), or we require that $\bar{U}(\vec{x}, \vec{u})$ is nonlinear in $\vec{u}$ (such that it keeps the action $\vec{u}$ finite).

First we consider imposing bounds on $\vec{u}$ via $\mathbb{A}$. For simplicity we will define $\mathbb{A}$ such that each component of $\vec{u}$ is in $[-1, 1]$, i.e. $\mathbb{A}$ is a unit hypercube. Then Eq. (5.5) simplifies to

$$\pi(\vec{x}, \vec{w})^i \approx \operatorname{sgn} \left(-\frac{\partial \bar{U}(\vec{x}, \vec{u})}{\partial u^i} - \gamma \frac{\partial \bar{f}(\vec{x}, \vec{u})}{\partial u^i} \widetilde{G}(\vec{x}, \vec{w}) \right), \tag{5.6}$$

for each component u^i of $\vec{u}$, where $\text{sgn}(\cdot) \in \{-1, 0, 1\}$ is the "sign" function from computer languages, and $\frac{\partial \widetilde{J}}{\partial \vec{x}}$ has been replaced by $\widetilde{G}$. Here $\pi(\vec{x}, \vec{w})$ denotes the greedy-policy function, i.e. this is a replacement for the action network $A(\vec{x}, \vec{e}, \vec{z})$.

This has achieved the goal of forcing finite action components within $\mathbb{A}$, however this is "bang-bang" control, i.e. actions are always either fully on or fully off (ignoring the remote possibility of the sgn function returning zero). Bang-bang control is not very practical to work with because it can damage physical controllers. Algorithmically it is not convenient either, because it can make convergence assurance difficult (since actions change unpredictably and discontinuously), and furthermore it was an assumption of the VGL algorithm definitions that the policy function would be differentiable (which bang-bang control is not). One way to address the lack-of-differentiability problem, and at the same time extend the VGL algorithms to be applicable to discrete-valued action spaces, is to replace all the instances of $\frac{\partial A}{\partial \vec{x}}$ by zero in the VGL algorithm. Successful applications using this modified VGL algorithm, with the above bang-bang greedy policy, have been performed (e.g. Fairbank, 2001).

However it would be much more practical if the sgn function above could somehow be changed into a smooth sigmoidal function $g(x)$, e.g. $g(x) = \tanh(x)$, but it is challenging to see how to achieve this without breaking the rules of the greedy-policy definition. Fortunately we can legitimately achieve this goal by following a method by Doya (2000), which is described in the next section.

5.2.2 First-Order Greedy Policy with Sigmoidal Control, for Continuous-Time Environments

Suppose we want the sgn function of the greedy-on-$\widetilde{Q}$ policy in Eq. (5.6) to be replaced by a smooth sigmoid function $g(x) : \mathbb{R} \to \mathbb{R}$, so that $(\pi(\vec{x}_t, \vec{w}))^i = g\left(-\left(\frac{\partial \bar{U}}{\partial u^i}\right)_t - \gamma \left(\frac{\partial \bar{f}}{\partial u^i}\right)_t \widetilde{G}_t\right)$ for all components i.

A common choice for $g(x)$ is $g(x) := \tanh(x/c)$ which ensures that each component of $\vec{u}$ is bound to $[-1, 1]$. Here c is a real positive constant that defines how "sharp" the sigmoidal shape is. As $c \to 0$, the sigmoidal shape becomes like a step shape, i.e. $g(x) \to \text{sgn}(x)$. Common possible functions for $g(x)$ are given in Table 5.2, with associated useful details.

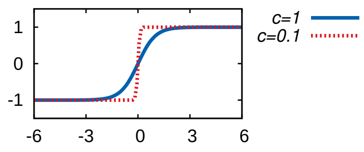
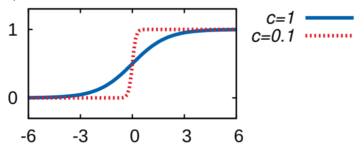
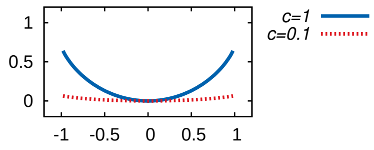
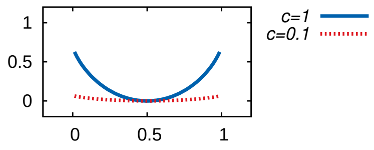
$g(x)$	$\tanh(x/c)$ 	$\frac{1}{1+e^{-x/c}}$ (Logistic Sigmoid) 
Range	$[-1, 1]$	$[0, 1]$
$g^{-1}(x)$	$c\,\mathrm{artanh}(x)$	$c\ln\left(\frac{x}{1-x}\right)$
$\int g^{-1}(x)dx$	$\frac{c}{2}\ln(1-x^2) + cx\,\mathrm{artanh}(x)$ 	$c\left(\ln(2-2x) + x\ln\left(\frac{x}{1-x}\right)\right)$ 
Derivative, $g'(x)$	$\frac{1}{c}\,\mathrm{sech}^2(x/c)$	$\frac{e^{-x/c}}{c(1+e^{-x/c})^2}$

Table 5.2: Common Choices for $g(x)$ in the Closed-Form Greedy Policy

Following the method of Doya (2000), we can define an "action-cost function" of

$$U^c(\vec{u}) = \sum_{i=1}^{\dim(\vec{u})} \int_k^{(u^i)} g^{-1}(x)dx, \tag{5.7}$$

where $(u^1, u^2, ..., u^{\dim(\vec{u})})$ are the components of $\vec{u}$. The value k is an arbitrary real constant, but it is neatest to set k to be the midpoint of the range of $g(x)$.

Note that differentiating $U^c(\vec{u})$ with respect to one of the components of $\vec{u}$ gives

$$\frac{\partial U^c(\vec{u})}{\partial u^i} = g^{-1}(u^i).$$

Then if we extend the function $g(x)$ and its inverse g^{-1} to be applicable to a vector argument, such that each element of the vector is acted upon component-wise, then we have

$$\frac{\partial U^c(\vec{u})}{\partial \vec{u}} = g^{-1}(\vec{u}). \tag{5.8}$$

Adding the action-cost function on to the original linear-in-$\vec{u}$ cost function $\bar{U}(\vec{x}, \vec{u})$, we obtain a new non-linear cost function $\bar{U}^{\text{tot}}$:

$$\bar{U}^{\text{tot}}(\vec{x},\vec{u}) := \bar{U}(\vec{x},\vec{u}) + U^c(\vec{u})\Delta\tau. \tag{5.9}$$

In this equation, $\Delta\tau$ is the sampling time for the underlying system being simulated/observed. $\Delta\tau$ is present here to make the $U^c(\vec{u})$ term have comparable magnitude to $\bar{U}(\vec{x},\vec{u})$, because we assume that $\bar{U}(\vec{x},\vec{u})$ is proportional to $\Delta\tau$. This assumption is made because this greedy-policy method is designed for continuous-time systems and therefore cost (or reward) is flowing into the system at a constant rate. If this assumption is not true, then the $\Delta\tau$ can be removed from the above equation, and from all of the following equations.

Under this action-cost function, the greedy policy requires us to minimise Eq. (5.4), i.e.

$$\vec{u} = \arg\min_{\vec{u}\in\mathbb{A}} \left(\bar{U}^{\text{tot}}(\vec{x},\vec{u}) + \gamma\bar{f}(\vec{x},\vec{u})\widetilde{G}(\vec{x},\vec{w})\right).$$

Using differentiation with respect to $\vec{u}$ to calculate this minimum, we consider

$$\begin{aligned}
\vec{0} &= \frac{\partial}{\partial\vec{u}}\left(\bar{U}^{\text{tot}}(\vec{x},\vec{u}) + \gamma\bar{f}(\vec{x},\vec{u})\widetilde{G}(\vec{x},\vec{w})\right) \\
&= \frac{\partial}{\partial\vec{u}}\left(\bar{U}(\vec{x},\vec{u}) + U^c(\vec{u})\Delta\tau + \gamma\bar{f}(\vec{x},\vec{u})\widetilde{G}(\vec{x},\vec{w})\right) && \text{(by Eq. (5.9))} \\
&= \frac{\partial\bar{U}(\vec{x},\vec{u})}{\partial\vec{u}} + g^{-1}(\vec{u})\Delta\tau + \gamma\frac{\partial\bar{f}(\vec{x},\vec{u})}{\partial\vec{u}}\widetilde{G}(\vec{x},\vec{w}). && \text{(by Eq. (5.8))}
\end{aligned}$$

Solving for $\vec{u}$, and using the fact that both $\frac{\partial\bar{U}(\vec{x},\vec{u})}{\partial\vec{u}}$ and $\frac{\partial\bar{f}(\vec{x},\vec{u})}{\partial\vec{u}}$ can be treated as constants (due to their linearity in $\vec{u}$), we obtain:

$$\begin{aligned}
g^{-1}(\vec{u})\Delta\tau &= -\frac{\partial\bar{U}(\vec{x},\vec{u})}{\partial\vec{u}} - \gamma\frac{\partial\bar{f}(\vec{x},\vec{u})}{\partial\vec{u}}\widetilde{G}(\vec{x},\vec{w}) \\
\Rightarrow\vec{u} &= g\left(\left(-\frac{\partial\bar{U}(\vec{x},\vec{u})}{\partial\vec{u}} - \gamma\frac{\partial\bar{f}(\vec{x},\vec{u})}{\partial\vec{u}}\widetilde{G}(\vec{x},\vec{w})\right)/\Delta\tau\right).
\end{aligned}$$

If we define

$$\vec{q}(\vec{x},\vec{w}) := \left(-\frac{\partial\bar{U}(\vec{x},\vec{u})}{\partial\vec{u}} - \gamma\frac{\partial\bar{f}(\vec{x},\vec{u})}{\partial\vec{u}}\widetilde{G}(\vec{x},\vec{w})\right)/\Delta\tau, \tag{5.10}$$

then the greedy-on-$\widetilde{Q}$ policy can be written very concisely as

$$\pi(\vec{x},\vec{w}) = g\left(\vec{q}(\vec{x},\vec{w})\right). \tag{5.11}$$

This is the greedy-policy function $\pi(\vec{x},\vec{w})$ with the sigmoid function as we required. It is smoothly differentiable, and efficient to calculate. All it requires is linearity in $\vec{u}$ of the functions $\bar{f}$ and $\bar{U}$, the sampling time $\Delta\tau$ to be sufficiently small to approximate the continuous-time motion. Again, we use the function name $\pi(\vec{x},\vec{w})$ to distinguish the greedy policy from a general policy function (action network) $A(\vec{x},\vec{e},\vec{z})$, and also to emphasise that the greedy policy uses the same weight vector as the critic network.

Note that there is no $\vec{u}$ argument in the definition of $\vec{q}(\vec{x},\vec{w})$ in Eq. (5.10), because the assumed linearity in $\vec{u}$ implies that all of the terms in the right-hand side are independent of $\vec{u}$.

5.2.2.1 Implementing VGL with a Sigmoidal Continuous-Time Closed-Form Greedy Policy

The above subsection derived the efficient continuous-time sigmoidal closed-form greedy-policy function $\pi(\vec{x},\vec{w})$. For this to be used in the VGL(λ) algorithm, the derivative $\frac{\partial \pi}{\partial \vec{x}}$ will be required, for example to implement the $\frac{\partial A}{\partial \vec{x}}$ which appears in the VGL algorithm definition (via Eq. (3.5)). Also, to calculate the target value gradient G'_t by Eq. (3.7), the quantity $\frac{D\bar{U}^{\text{tot}}}{D\vec{x}}$ will need calculating. Also, although not used in the VGL(λ) algorithm, it is useful to have an exact equation for $\frac{\partial \pi}{\partial \vec{w}}$. We calculate all of these derivatives in this section.

The derivatives of the policy $\pi(\vec{x},\vec{w})$ can be found from the chain rule:

$$\begin{aligned}
\left(\frac{\partial \pi^i}{\partial \vec{x}}\right)_t &= \left(\frac{\partial g(q^i)}{\partial \vec{x}}\right)_t && \text{(by (5.11))}\\
&= g'\left((\vec{q}_t)^i\right)\left(\frac{\partial q^i}{\partial \vec{x}}\right)_t && \text{(by chain rule)}\\
&= g'\left((\vec{q}_t)^i\right)\frac{\partial}{\partial \vec{x}}\left(\left(-\left(\frac{\partial \bar{U}}{\partial u^i}\right)_t - \gamma\left(\frac{\partial \bar{f}}{\partial u^i}\right)_t \widetilde{G}(\vec{x}_t,\vec{w})\right)/\Delta\tau\right) && \text{(by Eq. (5.10))}\\
&= -\frac{g'\left((\vec{q}_t)^i\right)}{\Delta\tau}\left(\left(\frac{\partial^2 \bar{U}}{\partial \vec{x}\partial u^i}\right)_t + \gamma\left(\frac{\partial^2 \bar{f}}{\partial \vec{x}\partial u^i}\right)_t \widetilde{G}_t + \gamma\left(\frac{\partial \widetilde{G}}{\partial \vec{x}}\right)_t\left(\frac{\partial \bar{f}}{\partial u^i}\right)_t^T\right),
\end{aligned}$$

where $g'(x)$ is the first derivative of $g(x)$, and again we define it to also accept a vector argument by being applicable component-wise to each vector element. This expression can be simplified if we define D_t as a diagonal matrix with

$$D_t := \operatorname{diag}(g'(\vec{q}_t)), \tag{5.12}$$

giving,

$$\left(\frac{\partial \pi}{\partial \vec{x}}\right)_t = -\left(\left(\frac{\partial^2 \bar{U}}{\partial \vec{x} \partial \vec{u}}\right)_t + \gamma \left(\frac{\partial^2 \bar{f}}{\partial \vec{x} \partial \vec{u}}\right)_t \widetilde{G}_t + \gamma \left(\frac{\partial \widetilde{G}}{\partial \vec{x}}\right)_t \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t^T\right) D_t / \Delta\tau. \tag{5.13}$$

Similarly,

$$\begin{aligned}
\left(\frac{\partial \pi^i}{\partial \vec{w}}\right)_t &= \left(\frac{\partial g(q^i)}{\partial \vec{w}}\right)_t && \text{(by (5.11))} \\
&= g'\left((\vec{q}_t)^i\right) \left(\frac{\partial q^i}{\partial \vec{w}}\right)_t && \text{(by chain rule)} \\
&= g'\left((\vec{q}_t)^i\right) \frac{\partial}{\partial \vec{w}} \left(\left(-\left(\frac{\partial \bar{U}}{\partial u^i}\right)_t - \gamma \left(\frac{\partial \bar{f}}{\partial u^i}\right)_t \widetilde{G}_t\right) / \Delta\tau\right) && \text{(by Eq. (5.10))} \\
&= -\gamma \frac{g'\left((\vec{q}_t)^i\right)}{\Delta\tau} \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \left(\frac{\partial \bar{f}}{\partial u^i}\right)_t^T .
\end{aligned}$$

This final line used the fact that $\frac{\partial \bar{U}}{\partial \vec{u}}$ and $\frac{\partial \bar{f}}{\partial \vec{u}}$ are not functions of $\vec{w}$ (so both of these two terms were treated as constants in the differentiation process). Rewriting using vector notation and the diagonal matrix D_t from Eq. (5.12) gives,

$$\left(\frac{\partial \pi}{\partial \vec{w}}\right)_t = -\gamma \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t^T D_t / \Delta\tau. \tag{5.14}$$

The quantity $\frac{D\bar{U}^{\text{tot}}}{D\vec{x}}$ needs calculating too for a correct VGL(λ) implementation (for example it is needed by Eq. (3.7)). This derivative is found as follows:

$$\begin{aligned}
\left(\frac{D\bar{U}^{\text{tot}}}{D\vec{x}}\right)_t &= \left(\frac{\partial \bar{U}}{\partial \vec{x}}\right)_t + \left(\frac{\partial \pi}{\partial \vec{x}}\right)_t \left(\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \left(\frac{\partial U^c}{\partial \vec{u}}\right)_t \Delta\tau\right) && \text{(by Eqs. (3.5),(5.9))} \\
&= \left(\frac{\partial \bar{U}}{\partial \vec{x}}\right)_t + \left(\frac{\partial \pi}{\partial \vec{x}}\right)_t \left(\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + g^{-1}(\vec{u}_t)\Delta\tau\right) && \text{(by Eq. (5.8))} \\
&= \left(\frac{\partial \bar{U}}{\partial \vec{x}}\right)_t + \left(\frac{\partial \pi}{\partial \vec{x}}\right)_t \left(\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \vec{q}_t\Delta\tau\right) && \text{(by Eq. (5.11))} \\
&= \left(\frac{D\bar{U}}{D\vec{x}}\right)_t + \left(\frac{\partial \pi}{\partial \vec{x}}\right)_t \vec{q}_t\Delta\tau. && \text{(by Eq. (3.5))}
\end{aligned} \tag{5.15}$$

Alg. 5.1 gives pseudocode for the VGL(λ) algorithm, with all the necessary modifications to work with the above defined greedy-on-$\widetilde{Q}$ policy. Eqs. (5.10) and (5.11) are implemented in lines 4 and 5 of the algorithm, respectively. The $\frac{\partial \pi}{\partial \vec{x}}$ term appearing in line 14 would need implementing by Eqs. (5.12) and (5.13).

Eq. (5.15) manifests itself as a modification to line 14 of the algorithm, which now contains the extra $\vec{q}_t\Delta\tau$ term, compared to the original version of the algorithm, Alg. 3.2. This is quite a subtle modification to the algorithm, which is a consequence of the original modification to the cost function $\bar{U}(\vec{x},\vec{u})$ to include the extra "action cost" term, $U^c(\vec{u})$. This modification reflects the fact that in demanding smoothly bound actions, we have modified the ADPRL problem to optimise a different objective function. This seems to be a necessary consequence for wanting to impose smooth action constraints, while still following the rules of the ADPRL problem specification (as defined in Section 1.2).

5.2.3 First-Order Greedy Policy with Sigmoidal Control, for Discrete-Time Environments

The greedy-on-$\widetilde{Q}$ policy derived above, in Section 5.2.2, was designed for when $\Delta\tau$ was sufficiently small, so that continuous-time motion could be approximated. Only in the limit $\Delta\tau \to 0$ would the greedy-on-$\widetilde{Q}$ policy generated be "exact". Consequently, for any finite $\Delta\tau > 0$, the approximation used in the first-order Taylor-series expansion of the greedy policy will have some amount of error in it, and so the trajectories generated by that first-order policy will not be exactly correct.

Algorithm 5.1 VGL(λ) with a Sigmoidal Continuous-Time Greedy Policy. Batch-Mode Implementation for Episodic Environments.

```
1:  $t \leftarrow 0$
2:  {Unroll trajectory...}
3:  while $\vec{x}_t \notin \mathbb{T}$ do
4:      $\vec{q}_t \leftarrow \left(-\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t - \gamma \left(\frac{\partial \vec{f}}{\partial \vec{u}}\right)_t \widetilde{G}_t\right) / \Delta\tau$
5:      $\vec{u}_t \leftarrow g(\vec{q}_t)$
6:      $\vec{x}_{t+1} \leftarrow f(\vec{x}_t, \vec{u}_t, \vec{e}_t)$
7:      $t \leftarrow t+1$
8:  end while
9:  $T \leftarrow t$
10: $\vec{p} \leftarrow \left(\frac{\partial \Phi}{\partial \vec{x}}\right)_t$
11: {Backwards pass...}
12: $\Delta\vec{w} \leftarrow \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t \left(\vec{p} - \widetilde{G}_t\right)$
13: for $t = T-1$ to $0$ step $-1$ do
14:     $G'_t \leftarrow \left(\frac{\partial \bar{U}}{\partial \vec{x}}\right)_t + \gamma \left(\frac{\partial \vec{f}}{\partial \vec{x}}\right)_t \vec{p} + \left(\frac{\partial \pi}{\partial \vec{x}}\right)_t \left(\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \vec{f}}{\partial \vec{u}}\right)_t \vec{p} + \vec{q}_t \Delta\tau\right)$
15:     $\Delta\vec{w} \leftarrow \Delta\vec{w} + \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t \left(G'_t - \widetilde{G}_t\right)$
16:     $\vec{p} \leftarrow \lambda G'_t + (1-\lambda)\widetilde{G}_t$
17: end for
18: $\vec{w} \leftarrow \vec{w} + \alpha \Delta\vec{w}$
```

While this approximation error can be made arbitrarily small by letting $\Delta\tau$ become small, doing so has the disadvantage of raising the computational expense of unrolling a trajectory, since the number of steps in a trajectory is inversely proportional to $\Delta\tau$. So this is a small but noticeable problem of the greedy policy described in Section 5.2.2. It is possible to solve this problem by moving to a fully continuous-time algorithm, and then the equations of motion given by Eqs. (1.1), (1.2) and (1.3) would turn into continuous-time differential equations, and the VGL(λ) weight update (Eq. (3.6)) would turn from a discrete-time summation into a continuous-time integral. Details of this continuous-time version of VGL(λ) are given by Fairbank (2008). Once differential equations are used, advanced solvers (such as Runge-Kutta) could be used to avoid the computational complexity issues, while still maintaining arbitrarily high accuracy.

However there is a more direct, discrete-time solution to this problem. In this method we change the greedy policy's dependency on $\widetilde{G}_{t+1}$ into a dependency on $\widetilde{G}_t$. This will let the greedy-on-$\widetilde{Q}$ policy become exact instead of approximate.

We choose $\bar{U}(\vec{x}, \vec{u}) := \bar{U}^{\text{tot}}(\vec{x}, \vec{u})$ by Eq. (5.9) and use Doya's action-cost function $U^c(\vec{u})$ given by Eq. (5.7) again, and then the greedy-on-$\widetilde{Q}$ policy function changes from Eq. (5.2) into:

$$\pi(\vec{x}_t, \vec{w}) = \arg\min_{\vec{u}_t \in \mathbb{A}} \left[\bar{U}^{\text{tot}}(\vec{x}_t, \vec{u}_t) + \gamma \widetilde{J}(\bar{f}(\vec{x}_t, \vec{u}_t), \vec{w})\right].$$

Then to minimise this function, we differentiate with respect to $\vec{u}$ and equate to zero:

$$\begin{aligned}
\vec{0} &= \frac{\partial}{\partial \vec{u}_t}\left(\bar{U}^{\text{tot}}(\vec{x}_t,\vec{u}_t) + \gamma \widetilde{J}(\bar{f}(\vec{x}_t,\vec{u}_t),\vec{w})\right) \\
&= \left(\frac{\partial \bar{U}^{\text{tot}}}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \left(\frac{\partial \widetilde{J}}{\partial \vec{x}}\right)_{t+1} && \text{(by chain rule)} \\
&= \left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \left(\frac{\partial U^c}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \left(\frac{\partial \widetilde{J}}{\partial \vec{x}}\right)_{t+1} && \text{(by Eq. (5.9))} \\
&= \left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + g^{-1}(\vec{u}_t) + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \widetilde{G}_{t+1}. && \text{(by Eq. (5.8))} \qquad (5.16)
\end{aligned}$$

The final line of Eq. (5.16) can be solved in closed-form if we assume that the three terms $\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t$, $\left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t$ and $\widetilde{G}_{t+1}$ are all constant with respect to $\vec{u}_t$, i.e. if we can assume that they are linear in $\vec{u}_t$.

Unfortunately the term $\widetilde{G}_{t+1} \equiv \widetilde{G}(\bar{f}(\vec{x}_t,\vec{u}_t),\vec{w})$ is not linear in $\vec{u}_t$, since the function $\widetilde{G}(\vec{x},\vec{w})$ is implemented by the critic network which is generally non-linear. Therefore Eq. (5.16) cannot be solved in closed form; a numerical solver would be required. There are many possible solutions to this problem, of varying efficiency and accuracy, but they nearly all have a major problem in that Eq. (5.16) can have multiple solutions. When this happens, making an infinitesimal change to $\vec{w}$ or $\vec{x}$ can make the solution in $\vec{u}$ change discontinuously, i.e. breaking the differentiability of the function $\pi(\vec{x},\vec{w})$, and thus making it not suitable for VGL methods.

The neatest solution to this problem is make the following two changes. Firstly, change the $\widetilde{G}_{t+1}$ term in Eq. (5.16) to use $\widetilde{G}_t$ instead, and thus obtain

$$\vec{0} = \left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + g^{-1}(\vec{u}_t) + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \widetilde{G}_t. \qquad (5.17)$$

And secondly, change the VGL(λ) weight update to use a target of ${G'}_{t+1}$ instead of the usual ${G'}_t$. This changes the VGL(λ) weight update from Eq. (3.6) into,

$$\Delta \vec{w} = \alpha \sum_t \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t \left({G'}_{t+1} - \widetilde{G}_t\right). \qquad (5.18)$$

The modified Eq. (5.17) is now a linear equation, and hence it has a closed-form solution, which solves to give the greedy-on-$\widetilde{Q}$ policy as,

$$\pi(\vec{x}_t, \vec{w}) = g\left(-\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t - \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \widetilde{G}(\vec{x}_t, \vec{w})\right). \tag{5.19}$$

This change, of replacing Eq. (5.16) by Eq. (5.17), follows the method used in the "Single Network Adaptive Critic" by Heydari and Balakrishnan (2011). This approach would at first appear to violate the construction of the greedy policy objective derived in Eq. (5.16). But the accompanying change of the VGL(λ) training algorithm to Eq. (5.18) restores the greedy policy objective, in the case that the training target can be met exactly. Since the training target cannot usually be met exactly, and anyway would take many training iterations to be met approximately, this approach mixes up the responsibilities of the greedy policy's minimisation objective with the critic's training objective; but this is only a slight mix up, assuming $\widetilde{G}_t$ is fairly close in value to $\widetilde{G}_{t+1}$, which will be true for a smooth $\widetilde{G}(\vec{x}, \vec{w})$ function and for small $\Delta\tau$.

Eq. (5.19) is efficient, smooth and differentiable, making it suitable for VGL methods. Ignoring the $\Delta\tau$ factor, it is identical to the continuous-time sigmoidal closed-form greedy policy (Eqs. (5.10)-(5.11)), but its usage is defined differently because it means the training algorithm needs to make $\widetilde{G}(\vec{x}_t, \vec{w})$ move towards ${G'}_{t+1}$ instead of the usual ${G'}_t$. Pseudocode for the VGL(λ) algorithm using this method and this greedy policy is given in Alg. 5.2. This algorithm was generated by modifying all instances of ${G'}_t$ in Alg. 5.1 to ${G'}_{t+1}$. This solves the problem of the greedy-on-$\widetilde{Q}$ policy only being an approximation when $\Delta\tau$ was large.

Algorithm 5.2 VGL(λ) with a Sigmoidal Discrete-Time Greedy Policy. Batch-Mode Implementation for Episodic Environments.

```
1: $t \leftarrow 0$
2: {Unroll trajectory...}
3: while $\vec{x}_t \notin \mathbb{T}$ do
4:     $\vec{q}_t \leftarrow -\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t - \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \widetilde{G}_t$
5:     $\vec{u}_t \leftarrow g(\vec{q}_t)$
6:     $\vec{x}_{t+1} \leftarrow f(\vec{x}_t, \vec{u}_t, \vec{e}_t)$
7:     $t \leftarrow t + 1$
8: end while
9: $T \leftarrow t$
10: $\vec{p} \leftarrow \left(\frac{\partial \Phi}{\partial \vec{x}}\right)_t$
11: {Backwards pass...}
12: $\Delta\vec{w} \leftarrow \vec{0}$
13: ${G'}_t \leftarrow \vec{p}$
14: for $t = T - 1$ to $0$ step $-1$ do
15:     ${G'}_t \leftarrow \left(\frac{\partial \bar{U}}{\partial \vec{x}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{x}}\right)_t \vec{p} + \left(\frac{\partial \pi}{\partial \vec{x}}\right)_t \left(\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \vec{p} + \vec{q}_t\right)$
16:     $\Delta\vec{w} \leftarrow \Delta\vec{w} + \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t \left({G'}_{t+1} - \widetilde{G}_t\right)$
17:     $\vec{p} \leftarrow \lambda {G'}_{t+1} + (1 - \lambda)\widetilde{G}_t$
18: end for
19: $\vec{w} \leftarrow \vec{w} + \alpha\Delta\vec{w}$
```

Simply redefining the VGL(λ) algorithm in this way (Eq. (5.18)) seems to have been quite a radical move, and results in the VGL(λ) algorithm attempting to move the values $\left(\frac{\partial \widetilde{J}}{\partial \vec{x}}\right)_t$ towards $\left(\frac{\partial J'}{\partial \vec{x}}\right)_{t+1}$. This could involve some "stretching" of the critic function's surface, from the points at t to the points at $t+1$, and means the learned critic-gradient function $\widetilde{G}(\vec{x},\vec{w})$ may not be the gradient of a true scalar field (for example it may violate the zero-curl condition). Thus it might be necessary to use a vector-critic implementation of $\widetilde{G}(\vec{x},\vec{w})$, since as it was noted in Section 3.4.1, vector critics obey fewer constraints than scalar critics.

5.2.4 Linear Model and Quadratic-Cost Functions

To complete the discussion of closed-form greedy policies, an efficient greedy-policy function is now described which often appears in the deterministic ADP and control-theory literature (for example, see Todorov, 2006; Wang et al., 2009), where a quadratic-cost function of the following form is used:

$$U(\vec{x},\vec{u}) = \frac{1}{2}\vec{u}^T R(\vec{x})\vec{u} + q(\vec{x}),$$

where $R(\vec{x}) \in \mathbb{R}^{\dim(\vec{u})\times\dim(\vec{u})}$ is a matrix function of $\vec{x}$, and $q(\vec{x}) \in \mathbb{R}$ is a scalar function of $\vec{x}$.

With this cost function, the greedy policy (Eq. 1.7) can be written as:

$$\vec{u} = \arg\min_{\vec{u}\in\mathbb{A}} \left(\frac{1}{2}\vec{u}^T R(\vec{x})\vec{u} + q(\vec{x}) + \gamma\widetilde{J}\left(f(\vec{x},\vec{u}),\vec{w}\right)\right).$$

Additionally, if the model function $f(\vec{x},\vec{u})$ is linear in $\vec{u}$, then this minimisation simplifies down to a closed-form solution. Equating the derivative to zero gives:

$$\begin{aligned}\vec{0} &= \frac{\partial}{\partial \vec{u}_t}\left(\frac{1}{2}\vec{u}_t^T R(\vec{x}_t)\vec{u}_t + q(\vec{x}_t) + \gamma\widetilde{J}\left(f(\vec{x}_t,\vec{u}_t),\vec{w}\right)\right)\\ &= R(\vec{x}_t)\vec{u}_t + \gamma\left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t\left(\frac{\partial \widetilde{J}}{\partial \vec{x}}\right)_{t+1}\\ \Rightarrow \vec{u}_t &= -\gamma R(\vec{x}_t)^{-1}\left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \widetilde{G}_{t+1}. \end{aligned} \tag{5.20}$$

Even though this function for $\vec{u}_t$ depends on a future quantity ($\widetilde{G}_{t+1}$), which would therefore make Eq. (5.20) unsuitable for use in a policy function, the techniques presented in Section 5.2.3 can be used to sidestep this problem. Therefore variants on Eq. (5.20) are commonly used in the ADP theory literature.

When Eq. (5.20) is used in control theory, the terms $\widetilde{G}_{t+1}$ are often used as independent learning parameters for $0 \leq t < T$. This is called "open-loop" control. This means the agent knows the value of $\widetilde{G}_{t+1}$ before it arrives at state $\vec{x}_{t+1}$, and therefore can easily use Eq. (5.20) to generate its action at state $\vec{x}_t$.

Unlike the two sigmoidal closed-form greedy policies described earlier, the actions produced by this equation are not in any way bound, therefore the sigmoidal greedy policies of the previous two sections are preferable for most ADPRL environments.

5.3 Empirical Results for a Greedy Policy

The previous sections of this chapter derived closed-form approximate greedy-policy solutions for linear-in-$\vec{u}$ learned model functions, which were efficient to calculate, bound to a suitable range, and most importantly, always differentiable, making them ideal for efficient and robust use in VGL. This section will show the effectiveness of the greedy policy on the Vertical-Lander problem and the classic Cart-Pole ADPRL-benchmark problem.

All of the experiments in this section are for deterministic environments, and hence the greedy-on-$\widetilde{Q}$ function is the same as the true greedy policy; although it will still be approximate since these experiments use the continuous-time greedy policy defined in Sec. 5.2.2 with a finite $\Delta\tau$ sampling time.

Initially, the performance of the sigmoidal greedy policy defined in Section 5.2.2 is compared to a conventional actor-critic architecture, to show that the greedy policy behaves very similarly to an actor-critic architecture with a high $\frac{\text{Actor learning-rate}}{\text{Critic learning-rate}}$ ratio. Experiments then show how a learning accelerator like RPROP can be used with a greedy policy, which is possible since there is only one neural network being optimised, however this can lead to instability because critic-learning algorithms are not generally true gradient descent. Then the effectiveness of the special Ω_t matrix described in Eq. (3.8) is demonstrated, with respect to its ability to stabilise algorithm convergence, in both the Vertical-Lander problem, and the Cart-Pole problem, and this solves the previously encountered instability issues.

5.3.1 Fixed-Duration Vertical-Lander Problem: Problem Specification

This test problem is a modification of the Vertical-Lander problem defined in Section 2.8.1. The principal modification is that the termination condition is changed to being when exactly 200 time steps have passed, i.e. trajectories have a fixed duration in this version of the problem. The reason for this change is that it avoids the need for clipping, which therefore simplifies the narrative. The original Vertical-Lander problem terminated when the spacecraft hit the ground or ran out of fuel, which created the need for clipping, as described in Chapter 10.

So in this modified version of the problem, in which trajectories terminate after a certain number of time steps have passed, fuel is irrelevant, and the state vector is

$$\vec{x} := (x_h, x_v, t)^T,$$

where these three quantities refer to height, velocity and integer time step, respectively. Another modification made in this version of the problem is the introduction of an action-cost function, which allows the greedy-policy methods of this chapter to be used.

The action $\vec{u} := u_a \in [0,1]$ is one-dimensional and produces the upward accelerations of the spacecraft. The model function $f(\vec{x}, u_a)$ is defined to be,

$$f((x_h, x_v, t)^T, u_a) =(x_h + x_v \Delta\tau, x_v + (u_a - k_g)\Delta\tau, t+1)^T.$$

for gravity constant k_g and time-sampling constant $\Delta\tau$.

The state-transition cost function is an action-cost function $\int g^{-1}(x)dx$ for a logistic-sigmoid function, $g(x)$. Following Table 5.2, this integral yields,

$$U(\vec{x}, u_a) =c\left(\ln(2-2u_a) + u_a \ln\left(\frac{u_a}{1-u_a}\right)\right)\Delta\tau \tag{5.21}$$

where $c = 0.01$ is constant. This will force the greedy policy to only generate values in the range of $g(x)$, i.e. in $[0,1]$, and will allow the efficient greedy-policy methods from this chapter to be used. Other than this action cost, there is no fuel-cost needed in this experiment.

The terminal-cost function is the same as before, i.e.

$$\Phi(\vec{x}_T) := \frac{1}{2}m(x_v)^2 + m(k_g)x_h, \tag{5.22}$$

where m is the spacecraft mass. To minimise this terminal-cost, the objective is for the spacecraft to lose as much mechanical energy as possible in the fixed given time. Although this is not literally simulating the spacecraft landing problem (since there is no solid ground to land on), it is still simulating the challenging part of the descent, i.e. to lose as much mechanical energy as possible by the time a certain end point is reached. The optimal strategy for this task is the same as in the previous problem, i.e. to leave the thruster switched off for as long as possible in the early stages of the journey, and at the last possible minute produce a maximum continuous burst of thrust to reduce the kinetic energy as much as possible.

Constants used in this version of the lander-problem were $m = 0.02$, $k_g = 0.2$, $\Delta\tau = 0.4$. In all of the experiments, the trajectory start states were fixed at $x_h = 1600$, $x_v = -2$ and $t = 0$; and learning used a discount factor of $\gamma = 1$.

A vector critic, $\widetilde{G}(\vec{x}, \vec{w})$, was provided by a fully connected MLP with 3 input units, two hidden layers of 6 units each, and 3 units in the output layer. Additional short-cut connections were present fully connecting all pairs of layers. The weights were initially randomised uniformly in the range $[-.1, .1]$. The activation functions were logistic sigmoid functions in the hidden layers, and the identity function in the output layer. The input to the MLP was $(x_h/1600, x_v/40, t/200)^T$ and the output gave $\widetilde{G}$ directly.

The action network was identical in design to the critic, except there was only one output node, and this had a logistic sigmoid function as its activation function. The output of the action network gave the spacecraft's acceleration u_a directly.

5.3.1.1 Results for Actor-Critic Versus Greedy-Policy Architectures

Experiments were performed using the greedy policy method, described in Section 5.2.2 and with pseudocode for VGL(λ) in Alg. 5.1, using $\frac{\partial \pi}{\partial \vec{x}}$ defined by Eq. (5.13); and also with a standard actor-critic architecture with pseudocode for VGL(λ) in Alg. 3.2. The results are presented in Fig. 5.1.

The actor-critic experiment used a high actor learning rate compared to the critic learning rate (i.e. $0.01 \gg 10^{-6}$). This was to make the actor-critic results comparable to

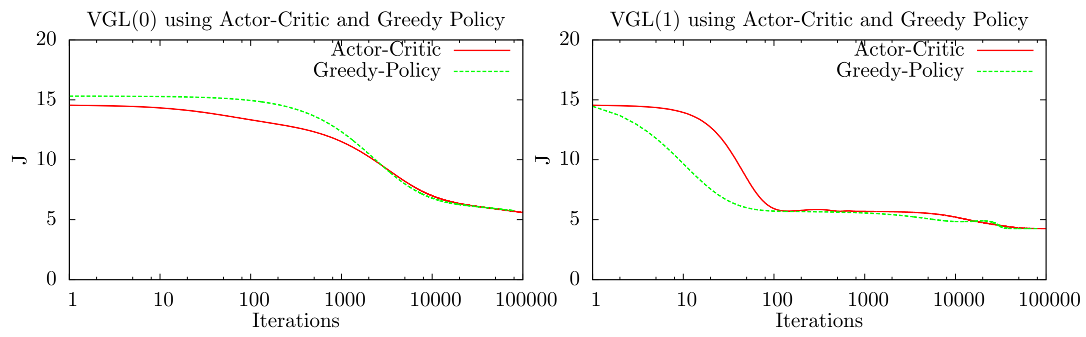


Figure 5.1: Greedy-Policy Versus Actor-Critic Performance on the Fixed-Duration Vertical-Lander Problem. The left graph shows learning by VGL(0) and the right graph shows learning by VGL(1). Both graphs show learning performance by an actor-critic architecture (with learning rates $\alpha = 10^{-6}$ and $\beta = 0.01$) compared to learning performance by a greedy policy (with learning rate $\alpha = 10^{-6}$). Each curve shows algorithm performance averaged over 40 trials.

those of the greedy policy, since a greedy policy is approximately equivalent to having $\beta \gg \alpha$. The greedy-policy results show very similar performance to the actor-critic results for both VGL(1) and VGL(0), which therefore constitutes a successful result for the greedy-policy implementation.

The discrepancy in results between the actor-critic and greedy-policy curves for the early iterations will be because the greedy-policy immediately acts like a very well-trained action-network, but with the actor-critic architecture, the action-network starts off with completely randomised weights.

The graphs also show that the VGL(1) algorithm produces a lower total cost J than the VGL(0) algorithm does, and does it faster. It is thought that this is because in this problem the major part of the total cost comes as a final impulse, so it is advantageous to have a long look-ahead (i.e. a high λ value) for fast and stable learning.

The similarity in results between actor-critic and greedy-policy shown in Fig. 5.1 indicates that in this experiment, the greedy policy can successfully replace the action network. This can potentially raise efficiency, because with the greedy policy, the learning rate for the critic could be raised significantly without worrying about the action-network ever being "left behind". In the next subsection, much higher learning rates for the critic are used.

5.3.1.2 Speeding up Learning with RPROP

Using a greedy policy, there are no longer two mutually interacting neural networks whose training could be interfering with each other. With the simpler architecture of just one neural network (the critic) to contend with, it is worth trying to speed up learning using one of the many available neural-network fast optimisation algorithms.

One of the simplest of such algorithms is RPROP (Riedmiller and Braun, 1993). Results for VGL(λ) with a greedy policy, accelerated by RPROP, are shown in Fig. 5.2. The graphs show faster learning than attained in the previous subsection, but the learning progress now is highly volatile. It seems the aggressive acceleration by RPROP can cause large instability in the VGL(1) and DHP algorithms. This is because neither of these two algorithms is true gradient descent when used with a greedy policy.

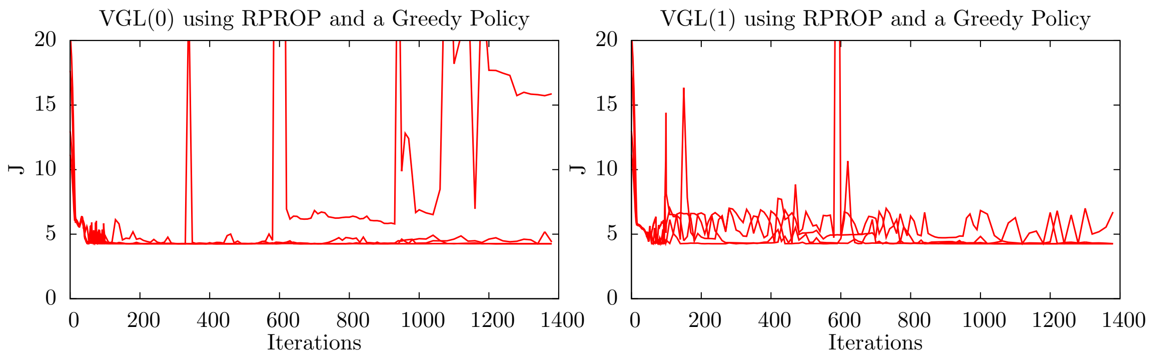


Figure 5.2: VGL(0) and VGL(1) performance on the Fixed-Duration Vertical-Lander, with a Greedy Policy, using RPROP. Each graph shows the performance of a learning algorithm for each of five different weight initialisations. Hence the ensemble of curves in each graph gives some idea of an algorithm's reliability and volatility.

However if the special Ω_t matrix defined by Eq. (3.8) is used,

$$\Omega_t = \begin{cases} \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)^T_{t-1} \left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{u} \partial \vec{u}}\right)^{-1}_{t-1} \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_{t-1} & \text{for } t > 0 \\ 0 & \text{for } t = 0, \end{cases}$$

where $\widetilde{Q}(\vec{x}, \vec{u}, \vec{w})$ is the model-based approximate Q-function defined by Eq. (8.4), and when $\lambda = 1$, then the algorithm referred to as VGLΩ(1) is obtained. VGLΩ(1) is true gradient descent on the sampled trajectory cost $\widehat{J}$, when combined with a greedy policy of the kind defined in this chapter, as proven in Chapter 8. The performance of this algorithm is shown in Fig, 5.3, and this shows the minimum being reached stably and many times quicker than in the actor-critic or non-RPROP case.

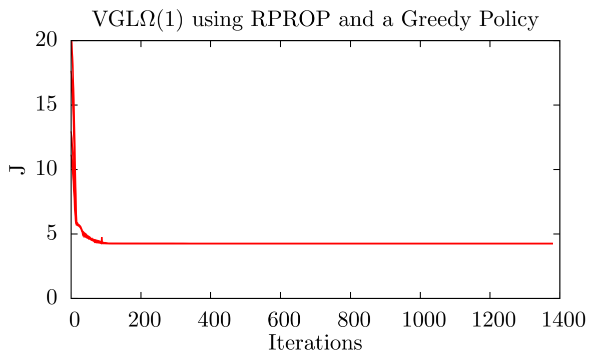


Figure 5.3: VGLΩ(1), with a Greedy Policy, using RPROP, on the Fixed-Duration Vertical-Lander Problem. The graph shows the performance of VGLΩ(1) for five different weight initialisations. Since the ensemble of curves here are almost all on top of each other, this shows that the algorithm is very consistent and stable. This is because VGLΩ(1) is true gradient descent on the sampled trajectory cost $\widehat{J}$, when combined with a policy that is greedy on the model-based approximate Q-function, which is the kind of greedy policy defined in this chapter

.

This represents a significant breakthrough in making learning with a greedy policy exhibit reliable and monotonic progress. The Ω_t equation that achieves this is only proven to work for VGL(1), and counterexamples exist for its use with DHP (as proven in Chapter 9).

5.3.2 Cart-Pole Experiment

The VGL(λ) and VGLΩ(λ) algorithms, with a greedy policy, are now applied to the well known Cart-Pole benchmark problem described in Fig. 5.4. The equation of motion for this system (Barto et al. (1983); Florian (2007); Lendaris and Paintz (1997)), in the absence of friction, is:

$$\ddot{\theta} = \frac{g\sin\theta - \cos\theta\left[\frac{F+ml\dot{\theta}^2\sin\theta}{m_c+m}\right]}{l\left[\frac{4}{3} - \frac{m\cos^2\theta}{m_c+m}\right]} \tag{5.23}$$

$$\ddot{x} = \frac{F + ml\left[\dot{\theta}^2\sin\theta - \ddot{\theta}\cos\theta\right]}{m_c+m} \tag{5.24}$$

where gravitational acceleration, $g = 9.8ms^{-2}$; cart's mass, $m_c = 1kg$; pole's mass, $m = 0.1kg$; half pole length, $l = 0.5m$; $F \in [-10, 10]$ is the force applied to the cart, in Newtons. The motion was integrated using the Euler method with a time constant $\Delta\tau = 0.02$, which, for a state vector $\vec{x} \equiv (x, \theta, \dot{x}, \dot{\theta})^T$, gives a model function $f(\vec{x}, \vec{u}, \vec{e}) = \vec{x} + (\dot{x}, \dot{\theta}, \ddot{x}, \ddot{\theta})^T \Delta\tau$.

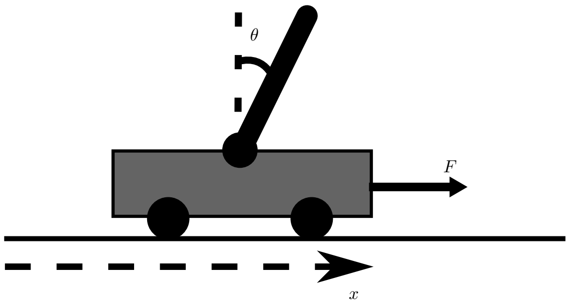


Figure 5.4: Cart-pole Benchmark Problem. A pole with a pivot at its base is balancing on a cart. The objective is to apply a changing horizontal force F to the cart which will move the cart backwards and forwards so as to balance the pole vertically. State variables are pole angle, θ, in radians, and cart position, x, plus their derivatives with respect to time, $\dot{\theta}$ and $\dot{x}$.

To achieve the objective of balancing the pole and keeping the cart close to the origin, $x = 0$, the following cost function is defined:

$$U(\vec{x}, t, u) = \gamma^t \left(5x^2 + 50\theta^2 + c\left(\ln(2 - 2u) + u \ln\left(\frac{u}{1-u}\right)\right)\right)\Delta\tau, \tag{5.25}$$

to be applied at each time step, where θ is in radians, and the term with coefficient c is the action-cost term defined from Table 5.2 corresponding to $\int g^{-1}(x)dx$ for a sigmoidal-logistic function, $g(x)$. This will ensure a greedy policy generates $u \in [0, 1]$. To generate the control force F_t from this, it is rescaled by $F_t := 20u_t - 10$, to achieve the desired range $F_t \in [-10, 10]$. The choice of c was here was $c = 10$.

Each trajectory was defined to last exactly 300 time steps, i.e. 6 seconds of real time, regardless of whether the pole swung below the horizontal or not, and with no constraint on the magnitude of x. This choice of cost function and the use of fixed duration trajectories is similar to that used by Lendaris and Paintz (1997) and Doya (2000), but differs to the trajectory termination criterion commonly used by the RL community (e.g. by Barto et al., 1983) which relies upon a non-differentiable step cost function, and hence is not appropriate for VGL based methods. This alternative, non-differentiable, cost function can be tackled by VGL methods if clipping is used, and this problem is addressed and solved separately in Chapter 10, but this current experiment will use the differentiable cost function defined by Eq. (5.25).

The discount factor used was $\gamma = 0.96$, in Eq. (5.25). This discount factor is placed in the definition of U so that the sharp truncation of trajectories terminating after 6 seconds is replaced by a smooth decay. This is preferable to the way that Algorithm 5.1 implements discount factors, which effectively treats each time step as creating a brand new cost-to-go function to minimise. Hence the value of γ that was used in Alg. 5.1 was $\gamma = 1$.

Training used a vector-critic MLP network, with 4 inputs, a single hidden layer of 12 units and 4 output nodes, with extra short-cut connections from the input layer to the output layer. The activation functions used were hyperbolic tangent functions at all nodes except for the output layer which used the identity function. Network weights were initially randomised uniformly from the range $[-0.1, 0.1]$. To ensure the state vector was suitably scaled for input to the MLP, a rescaled state vector, $\vec{x}' := (0.16x, 4\theta/\pi, \dot{x}, 4\dot{\theta})^T$, was used throughout the implementation. As noted by Lendaris and Paintz (1997), choosing an appropriate state-space scaling was critical to success with DHP on this problem.

In this experiment, a greedy policy was used, so there was no action network. Again, learning took place by Alg. 5.1, using $\frac{\partial \pi}{\partial \vec{x}}$ defined by Eq. (5.13).

Learning took place on 10 trajectories with fixed start points randomly chosen with $|x| < 2.4$, $|\theta| < \pi/15$, $|\dot{x}| < 5$, $|\dot{\theta}| < 5$, which are similar to the starting conditions used by Barto et al. (1983). The exact derivatives of the model and cost functions were made available to the algorithms. Four algorithms were tested and their results are shown in Fig. 5.5. Both VGL(1) and VGL(0) showed quite volatile performance when accelerated by RPROP. The results again show that VGLΩ(1) had less volatility and better performance than both VGL(1) and VGL(0), which demonstrates the effectiveness of the Ω_t matrix used. For comparison, results for an actor-only architecture (i.e. with no critic) trained entirely by BPTT and RPROP are given (the BPTT algorithm is described by Werbos (1990a), and will be defined in full in Chapter 6). This demonstrates that the minimum attained by the VGL algorithms is suitably low. It was observed that when this minimum was reached, the pole was being balanced effectively with the cart remaining close to $x = 0$, which indicates the problem being solved successfully.

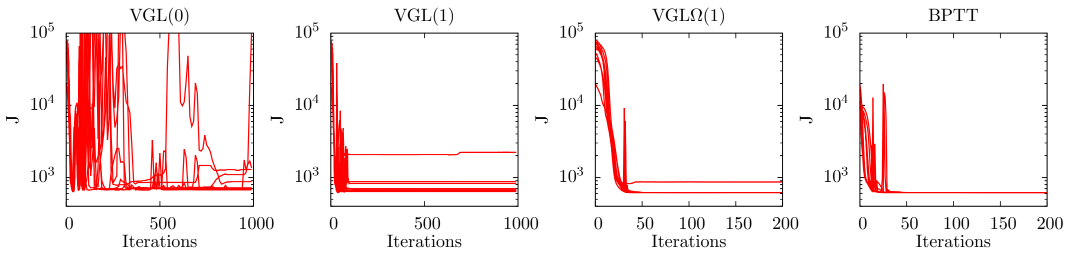


Figure 5.5: Cart-Pole Solutions by VGL(0), VGL(1), VGLΩ(1) and BPTT, with a Greedy Policy. All algorithms were used in conjunction with RPROP. Each graph shows the performance of a learning algorithm for each of ten different random weight initialisations; hence each ensemble of curves gives some idea of an algorithm's reliability and volatility. The BPTT result is provided for comparison.

The results show the Cart-Pole problem being solved effectively. With the use of the Ω_t matrix and a greedy policy, fast learning and largely monotonic progress has been obtained for a critic-learning algorithm (with the brief non-monotonicity down to the aggressive acceleration of RPROP and/or discontinuities in the cost-to-go function surface in weight space). The ADPRL critic architecture has replicated the performance of BPTT, a proven gradient descent method.

5.4 Chapter Conclusions

Several action-network weight updates have been defined, but this chapter has argued that when using a model-based approach in a continuous-valued state space, it is better to omit the action network and just use a greedy policy. The equivalence of using a greedy policy to an actor-critic architecture, under appropriate learning rate parameters, was demonstrated empirically in Section 5.3.1.1. Later experiments demonstrated speed-up methods which were applicable for a greedy policy, because then the actor-critic architecture reduced down to just one neural network, which made acceleration methods such as RPROP viable. This yielded a massive increase in learning speed, under stable convergence when combined with VGLΩ(1).

Several greedy-policy methods have been defined for linear-in-$\vec{u}$ models; bang-bang control, sigmoidal control (continuous time), sigmoidal control (discrete time) and quadratic control. All of the methods minimised the model-based approximate Q function, according to Eq. (5.2), as opposed to the true ADPRL objective greedy policy given by Eq. (1.8), which means the greedy policies presented in this chapter will only be approximations in the most general stochastic case. In deterministic environments, the greedy policies derived here are exact. They can also be exact in the limited stochastic case where the effect of the noise is additive and independent of the actions chosen (as proven in the chapter appendix, Section 5.A).

Out of the various greedy policy approximations presented, the sigmoidal-control methods seem to be the best for VGL algorithms, because action limits can be rigorously enforced while maintaining differentiability of the policy (an essential requirement for VGL methods). The greedy-policy method proposed here introduces an extra action-cost term, which therefore changes the ADPRL problem to be solved, and which might therefore be viewed as a contentious issue; but the choice of a low c value can make this action-cost term arbitrarily small, therefore changing the original problem definition by an arbitrarily small amount. The benefits of using this method with respect to efficiency and differentiability (a necessary requirement for use with VGL methods) seem to outweigh the concerns about having to amend the problem's cost function.

5.A Chapter Appendix: Equivalence of the Approximated Greedy-on-$\widetilde{Q}$ Policy to the True ADPRL Greedy Policy under Noise that is Independent of the Action

It has been described in this chapter (particularly in Section 5.2) that when the environment is stochastic, the greedy-on-$\widetilde{Q}$ policies derived are only approximations to the true ADPRL greedy policy (Eq. (1.8)). The objective of this appendix is to show that in some closed-form greedy-policy situations, when the effect of random noise is independent of $\vec{u}$, the greedy-on-$\widetilde{Q}$ policy becomes equivalent to the true ADPRL greedy policy.

In general, the displacement the agent makes in one time step is given by $(f(\vec{x}_t, \vec{u}_t, \vec{e}_t) - \vec{x}_t)$. In a continuous-time system, when this displacement is calculated by the Euler method with sampling time $\Delta\tau$, the displacement in one time step could be decomposed into

$$(f(\vec{x}_t, \vec{u}_t, \vec{e}_t) - \vec{x}_t) \equiv \underline{f}(\vec{x}_t, \vec{u}_t)\Delta\tau + \Delta\xi_f(\vec{x}_t, \vec{e}_t), \tag{5.26}$$

where $\underline{f}$ is a deterministic function, and $\Delta\xi_f(\vec{x}_t, \vec{e}_t)$ is a pure-noise vector of dimension $\dim(\vec{x})$, which we are assuming here is independent of $\vec{u}$. By defining the deterministic function $\underline{f}(\vec{x}_t, \vec{u}_t)\Delta\tau := \mathbb{E}\left(f(\vec{x}_t, \vec{u}_t, \vec{e}_t) - \vec{x}_t\right)$, and taking expectations of both sides of Eq. (5.26), we can therefore deduce that

$$\mathbb{E}\left(\Delta\xi_f(\vec{x}_t, \vec{e}_t)\right) \equiv \vec{0}. \tag{5.27}$$

In discrete time, the evolution of the noise part of Eq. (5.26) can be treated as a random walk. For a random-walk, the expected distance-squared from the start point is proportional to the number of time steps passed. Hence the variance of the position in a random walk is proportional to t. Similarly, in the continuous-time limit (a Wiener process), the noise vector $(\Delta\xi_f)$'s variance is proportional to $\Delta\tau$. This variance will be given by a $\dim(\vec{x}) \times \dim(\vec{x})$ matrix ν:

$$\mathbb{E}\left((\Delta\xi_f)(\Delta\xi_f)^T\right) = \nu(\vec{x}_t, \vec{e}_t)\Delta\tau, \tag{5.28}$$

which has also been assumed to be independent of $\vec{u}$. Unlike with deterministic functions, where Taylor-series expansions to first order were sufficient (e.g. as done in Section 5.2.1), the fact that ν in Eq. (5.28) has a magnitude proportional to $\Delta\tau$ means that Taylor-series expansions of the greedy-policy minimisation objective need to be made to second-order.

We can also decompose the cost function, $U(\vec{x}, \vec{u}, \vec{e})$, into

$$U(\vec{x}, \vec{u}, \vec{e}) \equiv \underline{U}(\vec{x}, \vec{u})\Delta\tau + \Delta\xi_U(\vec{x}, \vec{e}), \tag{5.29}$$

where

$$\mathbb{E}\left(\Delta\xi_U(\vec{x}, \vec{e})\right) = 0. \tag{5.30}$$

A second-order Taylor-series expansion of the greedy policy (Eq. (1.8)) proceeds as follows:

$$\pi(\vec{x}_t, \vec{w}) := \arg\min_{\vec{u} \in \mathbb{A}} \mathbb{E}\left(U(\vec{x}_t, \vec{u}_t, \vec{e}_t) + \gamma \widetilde{J}(f(\vec{x}_t, \vec{u}_t, \vec{e}_t), \vec{w})\right) \qquad \text{(greedy policy)}$$

$$\approx \arg\min_{\vec{u}_t \in \mathbb{A}} \mathbb{E}\Bigg[U(\vec{x}_t, \vec{u}_t, \vec{e}_t) + \gamma \left(\frac{\partial \widetilde{J}}{\partial \vec{x}}\right)_t^T (f(\vec{x}_t, \vec{u}_t, \vec{e}_t) - \vec{x}_t)$$
$$+ \frac{\gamma}{2}(f(\vec{x}_t, \vec{u}_t, \vec{e}_t) - \vec{x}_t)^T \left(\frac{\partial^2 \widetilde{J}}{\partial \vec{x} \partial \vec{x}}\right)_t (f(\vec{x}_t, \vec{u}_t, \vec{e}_t) - \vec{x}_t)\Bigg]. \qquad \text{(Taylor Series)}$$

Substituting Eqs. (5.26) and (5.29) into this Taylor-series expansion, and omitting t subscripts, gives:

$$\pi(\vec{x}, \vec{w}) \approx \arg\min_{\vec{u} \in \mathbb{A}} \mathbb{E}\Bigg[\underline{U}(\vec{x}, \vec{u})\Delta\tau + \Delta\xi_U(\vec{x}, \vec{e}) + \gamma \left(\frac{\partial \widetilde{J}}{\partial \vec{x}}\right)^T (\underline{f}(\vec{x}, \vec{u})\Delta\tau + \Delta\xi_f(\vec{x}, \vec{e}))$$
$$+ \frac{\gamma}{2}(\underline{f}(\vec{x}, \vec{u})\Delta\tau + \Delta\xi_f(\vec{x}, \vec{e}))^T \left(\frac{\partial^2 \widetilde{J}}{\partial \vec{x} \partial \vec{x}}\right) (\underline{f}(\vec{x}, \vec{u})\Delta\tau + \Delta\xi_f(\vec{x}, \vec{e}))\Bigg].$$

If $\Delta\tau$ is small enough so that we can ignore the higher-order terms $(\Delta\tau)^2$ and $(\Delta\tau)(\Delta\xi_f)$, then we obtain,

$$\pi(\vec{x}, \vec{w}) \approx \arg\min_{\vec{u} \in \mathbb{A}} \mathbb{E}\Bigg[\underline{U}(\vec{x}, \vec{u})\Delta\tau + \Delta\xi_U(\vec{x}, \vec{e}) + \gamma \left(\frac{\partial \widetilde{J}}{\partial \vec{x}}\right)^T (\underline{f}(\vec{x}, \vec{u})\Delta\tau + \Delta\xi_f(\vec{x}, \vec{e}))$$
$$+ \frac{\gamma}{2}(\Delta\xi_U(\vec{x}, \vec{e}))^T \left(\frac{\partial^2 \widetilde{J}}{\partial \vec{x} \partial \vec{x}}\right) (\Delta\xi_U(\vec{x}, \vec{e}))\Bigg]$$
$$\approx \arg\min_{\vec{u} \in \mathbb{A}} \mathbb{E}\Bigg[\underline{U}(\vec{x}, \vec{u})\Delta\tau + \gamma \left(\frac{\partial \widetilde{J}}{\partial \vec{x}}\right)_t^T \underline{f}(\vec{x}, \vec{u})\Delta\tau$$
$$+ \frac{\gamma}{2}\sum_{ij}\left(\nu(\vec{x}, \vec{e})^{ij}\left(\frac{\partial^2 \widetilde{J}}{\partial \vec{x}^i \partial \vec{x}^j}\right)\right)\Delta\tau\Bigg]. \qquad \text{(by Eqs. (5.27), (5.30) \& (5.28))}$$

Since the ν term is independent of $\vec{u}$, it cannot affect the min operator in the above expression, and hence the ν term can be omitted. Therefore the greedy policy simplifies to,

$$\begin{aligned}\pi(\vec{x},\vec{w}) &\approx \arg\min_{\vec{u}\in\mathbb{A}} \mathbb{E}\left[\underline{U}(\vec{x},\vec{u})\Delta\tau + \gamma\left(\frac{\partial\widetilde{J}}{\partial\vec{x}}\right)^T \underline{f}(\vec{x}_t,\vec{u}_t)\Delta\tau\right] \\ &= \arg\min_{\vec{u}\in\mathbb{A}} \left[\underline{U}(\vec{x},\vec{u})\Delta\tau + \gamma\left(\frac{\partial\widetilde{J}}{\partial\vec{x}}\right)^T \underline{f}(\vec{x},\vec{u})\Delta\tau\right], \qquad (5.31)\end{aligned}$$

where in this final line, the expectation symbol was dropped because all of its argument's terms are deterministic functions.

Substituting $\underline{f}(\vec{x},\vec{u})\Delta\tau \equiv (\mathbb{E}(f(\vec{x},\vec{u},\vec{e})) - \vec{x}) \equiv (\bar{f}(\vec{x},\vec{u}) - \vec{x})$ and $\underline{U}(\vec{x},\vec{u})\Delta\tau \equiv \mathbb{E}(U(\vec{x},\vec{u},\vec{e})) \equiv \bar{U}(\vec{x},\vec{u})$, we see that the greedy policy described for Eq. (5.31) is equivalent to the Taylor-series approximation derived for the greedy-on-$\widetilde{Q}$ function (Eq. (5.4)).

The conclusion of this appendix is that when the variances of the noise terms ξ_f and ξ_U are independent of $\vec{u}$, the greedy-on-$\widetilde{Q}$ policy derived in Section 5.2.2 will be equivalent to the true ADPRL greedy policy in stochastic environments (assuming $\Delta\tau$ is sufficiently small).

Chapter 6

Critic-free ADPRL

It is possible to do ADPRL without a critic. In this chapter two methods which do this are described: Policy-Gradient Learning (PGL) and Backpropagation through time (BPTT). Both of these methods perform a kind of stochastic gradient descent on the sampled total trajectory cost, $\widehat{J}(\vec{x}, \vec{\mathbf{e}}, \vec{z})$, with respect to the weight vector $\vec{z}$ of the action network.

Although this thesis is primarily about critic-based algorithms, these two critic-free algorithms are included because one of them, BPTT, becomes equivalent to VGL$\Omega(1)$, under certain conditions. This equivalence to VGL$\Omega(1)$, and the circumstances under which it arises, is proven in Chapter 8. The equivalence proof also provides the convergence proof for VGL$\Omega(1)$, therefore BPTT is a very important algorithm for this thesis.

BPTT is a model-based method to calculate $\frac{\partial \widehat{J}(\vec{x}, \vec{\mathbf{e}}, \vec{z})}{\partial \vec{z}}$ for a given trajectory generated by a given differentiable policy function $A(\vec{x}, \vec{e}, \vec{z})$, and a trajectory start point $\vec{x}_0$. When combined with a gradient-descent weight update, e.g. $\Delta\vec{z} = -\beta\left(\frac{\partial \widehat{J}}{\partial \vec{z}}\right)_0$, it can be used to train the action network $A(\vec{x}, \vec{e}, \vec{z})$ so as to produce optimal control. BPTT is applicable to environments with continuous-valued state and action spaces, with differentiable environment functions, and deterministic or stochastic, differentiable policy functions. The algorithm is exact when the noise in the environment is non-existent or additive, or if the conditions listed previously for VGL algorithms in Section 3.2.1 apply, but otherwise it only approximately calculates the derivative $\frac{\partial \widehat{J}(\vec{x}, \vec{\mathbf{e}}, \vec{z})}{\partial \vec{z}}$.

PGL is a model-free stochastic method which, over a period of several weight updates, accumulates a mean weight update which is gradient descent on the total trajectory cost. There are several PGL algorithms in the RL literature, but this chapter will concentrate on one of the most important and representative ones, REINFORCE, by Williams (1992). REINFORCE is applicable to environments with discrete or continuous-valued state and action spaces, and stochastic policy functions.

Table 6.1 summarises the differences between the two approaches.

Equation	Model Free?
BPTT (Werbos, 1990a) $\Delta\vec{z} = -\beta\frac{\partial\widehat{J}(\vec{x},\vec{\mathbf{e}},\vec{z})}{\partial\vec{z}}$.	No
Policy Gradient Learning (REINFORCE, Williams (1992)) $\mathbb{E}\left(\Delta\vec{z}\right) = -\beta\frac{\partial\mathbb{E}\left(\widehat{J}(\vec{x},\vec{\mathbf{e}},\vec{z})\right)}{\partial\vec{z}}$.	Yes

Table 6.1: Common Policy-Gradient Action-Network Weight Updates

Note that PGL is a very different thing from VGL, even though both have the word "gradient" in the title. Policy gradients are gradients with respect to $\vec{z}$, and value gradients are gradients with respect to $\vec{x}$.

In this chapter, in Section 6.1 the BPTT method is described and the algorithm is derived and its convergence properties are discussed. In Section 6.2, the REINFORCE algorithm is described, and the mean of its weight update is proven to be equivalent to gradient descent on the total trajectory cost. Section 6.3 demonstrates a closer theoretical connection between PGL and BPTT than may have been known otherwise, and the motivations for doing this are given. Finally chapter conclusions are given in Section 6.4, summarising the key differences and pros and cons between BPTT and PGL.

6.1 Backpropagation Through Time for Control Problems

BPTT can be used to calculate the derivatives of any differentiable scalar function with respect to the weights of a neural network. BPTT may be misunderstood as being limited to training recurrent neural networks to learn static time-series data, but BPTT has much more flexible applications than this, as described by Werbos (1990a). To apply BPTT in solving a control problem, it can be used to find the derivatives of $\widehat{J}(\vec{x}_0, \vec{\mathbf{e}}_0, \vec{z})$ with respect to $\vec{z}$, so as to enable gradient descent on $\widehat{J}$, via $\Delta\vec{z} = -\beta\left(\frac{\partial\widehat{J}}{\partial\vec{z}}\right)_0$, for some small positive learning rate β. Fig. 6.1 summarises how BPTT can be used to solve control problems.

BPTT for control is
Gradient descent on $\widehat{J}(\vec{x}, \vec{\mathbf{e}}, \vec{z})$
with respect to $\vec{z}$
for some given action network, $A(\vec{x}, \vec{e}, \vec{z})$.

Figure 6.1: Application of BPTT for Control Problems.

Gradient descent of this kind will naturally find local minima of $\widehat{J}(\vec{x}_0, \vec{\mathbf{e}}_0, \vec{z})$, and has good convergence properties when the surface $\widehat{J}(\vec{x}_0, \vec{\mathbf{e}}_0, \vec{z})$ is smooth with respect to $\vec{z}$. If the minimisation is extended to minimise $\widehat{J}(\vec{x}_0, \vec{\mathbf{e}}_0, \vec{z})$ from many different start points $\vec{x}_0 \in \mathbb{S}$, then BPTT can be used to find a general trained action-network $A(\vec{x}, \vec{e}, \vec{z})$ which solves the ADPRL problem very effectively, subject to the limitations described in Section 6.1.2. The whole BPTT training process works without the use of a critic at all.

6.1.1 BPTT for Control, Algorithm Derivation

To calculate the BPTT gradient-descent equation, we first rewrite the sampled trajectory cost (Eq. (1.4)) recursively as

$$\widehat{J}(\vec{x}_t, \vec{\mathbf{e}}_t, \vec{z}) := \begin{cases} U(\vec{x}_t, A(\vec{x}_t, \vec{e}_t, \vec{z}), \vec{e}_t) + \gamma \widehat{J}(f(\vec{x}_t, \vec{u}_t, \vec{e}_t), \vec{\mathbf{e}}_{t+1}, \vec{z}) & \text{if } \vec{x}_t \notin \mathbb{T} \\ \Phi(\vec{x}_t, \vec{e}_t) & \text{if } \vec{x}_t \in \mathbb{T}. \end{cases} \tag{6.1}$$

Using trajectory-shorthand notation (Section 2.1.2) and the vector-function differentiation notation defined in Section 3.3, we calculate the gradient of Eq. (6.1) with respect to $\vec{z}$ by differentiation using the chain rule and substitution of Eqs. (1.1) and (1.3):

$$\left(\frac{\partial \widehat{J}}{\partial \vec{z}}\right)_t = \begin{cases} \left(\frac{\partial A}{\partial \vec{z}}\right)_t \left(\left(\frac{\partial U}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial f}{\partial \vec{u}}\right)_t \left(\frac{\partial \widehat{J}}{\partial \vec{x}}\right)_{t+1} \right) + \gamma \left(\frac{\partial \widehat{J}}{\partial \vec{z}}\right)_{t+1} & \text{if } \vec{x}_t \notin \mathbb{T} \\ \vec{0} & \text{if } \vec{x}_t \in \mathbb{T}. \end{cases}$$

Since it is often not practical to have access to the true derivatives $\frac{\partial U}{\partial \vec{u}}$ and $\frac{\partial f}{\partial \vec{u}}$, for example since the noise vector $\vec{e}$ is often not observable, or for other reasons highlighted in Section 3.2.1, it is more convenient to replace the above equation by,

$$\left(\frac{\partial \widehat{J}}{\partial \vec{z}}\right)_t \approx \begin{cases} \left(\frac{\partial A}{\partial \vec{z}}\right)_t \left(\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \left(\frac{\partial \widehat{J}}{\partial \vec{x}}\right)_{t+1}\right) + \gamma \left(\frac{\partial \widehat{J}}{\partial \vec{z}}\right)_{t+1} & \text{if } \vec{x}_t \notin \mathbb{T} \\ \vec{0} & \text{if } \vec{x}_t \in \mathbb{T}. \end{cases} \tag{6.2}$$

where $\bar{f}(\vec{x}, \vec{u})$ and $\bar{U}(\vec{x}, \vec{u})$ are the deterministic, learned model and cost functions. Since usually $\frac{\partial \bar{f}}{\partial \vec{u}} \approx \frac{\partial f}{\partial \vec{u}}$ and $\frac{\partial \bar{U}}{\partial \vec{u}} \approx \frac{\partial U}{\partial \vec{u}}$ (by Eq. (3.1)), the above equation uses an approximation instead of an equality. Hence in this case the BPTT pseudocode derived below will only approximately find the learning gradient $\frac{\partial \widehat{J}}{\partial \vec{z}}$. However if it is the case that $\vec{e}$ is perfectly observable and the true functions $f(\vec{x}, \vec{u}, \vec{e})$ and $U(\vec{x}, \vec{u}, \vec{e})$ are perfectly known, or one of the other three situations highlighted in Section 3.2.1 applies, then the true environment functions could be used in the above equation instead of the learned ones. In this case, Eq. (6.2) would become exact, and hence the BPTT pseudocode derived below would be exact.

Expanding this recursion and substituting it into the gradient-descent equation $\Delta\vec{z} = -\beta \left(\frac{\partial \widehat{J}}{\partial \vec{z}}\right)_0$ gives,

$$\Delta\vec{z} = -\beta \sum_{t\geq 0} \gamma^t \left(\frac{\partial A}{\partial \vec{z}}\right)_t \left(\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \left(\frac{\partial \widehat{J}}{\partial \vec{x}}\right)_{t+1}\right). \tag{6.3}$$

This weight update is BPTT with gradient descent to minimise $\widehat{J}(\vec{x}_0, \vec{\mathbf{e}}_0, \vec{z})$ with respect to the weight vector $\vec{z}$ of an action network $A(\vec{x}, \vec{e}, \vec{z})$. It refers to the quantity $\frac{\partial \widehat{J}}{\partial \vec{x}}$ which can be found recursively by differentiating Eq. (6.1) and using the chain rule, giving

$$\begin{aligned} \left(\frac{\partial \widehat{J}}{\partial \vec{x}}\right)_t &= \begin{cases} \left(\frac{DU}{D\vec{x}}\right)_t + \gamma \left(\frac{Df}{D\vec{x}}\right)_t \left(\frac{\partial \widehat{J}}{\partial \vec{x}}\right)_{t+1} & \text{if } \vec{x}_t \notin \mathbb{T} \\ \left(\frac{\partial \Phi}{\partial \vec{x}}\right)_t & \text{if } \vec{x}_t \in \mathbb{T}, \end{cases} \\ &\approx \begin{cases} \left(\frac{D\bar{U}}{D\vec{x}}\right)_t + \gamma \left(\frac{D\bar{f}}{D\vec{x}}\right)_t \left(\frac{\partial \widehat{J}}{\partial \vec{x}}\right)_{t+1} & \text{if } \vec{x}_t \notin \mathbb{T} \\ \left(\frac{\partial \Phi}{\partial \vec{x}}\right)_t & \text{if } \vec{x}_t \in \mathbb{T}, \end{cases} \qquad \text{(by Eq. (3.1))} \end{aligned} \tag{6.4}$$

where the notation $\frac{D}{D\vec{x}}$ is defined by Eq. (3.5), and again the learned model and cost functions have used instead of the true ones.

Equation (6.4) can be understood to be propagating the quantity $\left(\frac{\partial \widehat{J}}{\partial \vec{x}}\right)_{t+1}$ backwards through the action network, model and cost functions to obtain $\left(\frac{\partial \widehat{J}}{\partial \vec{x}}\right)_t$, and giving the BPTT algorithm its name. Pseudocode for the full BPTT algorithm is given in Alg. 6.1. In the pseudocode, the variable name $\vec{p}_t$ holds the quantity $\left(\frac{\partial \widehat{J}}{\partial \vec{x}}\right)_t$. The pseudocode involves the derivatives of the action network, i.e. $\frac{\partial A}{\partial \vec{x}}$ and $\frac{\partial A}{\partial \vec{z}}$, which both would involve neural-network backpropagation, assuming $A(\vec{x}, \vec{e}, \vec{z})$ was implemented by a neural network. This neural-network backpropagation should be considered as a sub-module necessary to implement Alg. 6.1, and should not be confused with BPTT itself.

Algorithm 6.1 Backpropagation Through Time for Control.

```
1: Unroll full trajectory from start state $\vec{x}_0$ using Alg. 1.1, and retain the variables $\vec{x}_t$, $\vec{u}_t$ and $T$.
2: $\frac{\partial \widehat{J}}{\partial \vec{z}} \leftarrow \vec{0}$
3: $\vec{p}_T \leftarrow \left(\frac{\partial \bar{\Phi}}{\partial \vec{x}}\right)_T$
4: for $t = T-1$ to 0 step $-1$ do
5:     $\frac{\partial \widehat{J}}{\partial \vec{z}} \leftarrow \frac{\partial \widehat{J}}{\partial \vec{z}} + \gamma^t \left(\frac{\partial A}{\partial \vec{z}}\right)_t \left(\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \vec{p}_{t+1}\right)$
6:     $\vec{p}_t \leftarrow \left(\left(\frac{\partial \bar{U}}{\partial \vec{x}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{x}}\right)_t \vec{p}_{t+1}\right) + \left(\frac{\partial A}{\partial \vec{x}}\right)_t \left(\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \vec{p}_{t+1}\right)$
7: end for
8: $\vec{z} \leftarrow \vec{z} - \beta \frac{\partial \widehat{J}}{\partial \vec{z}}$
```

Algorithm 6.1 does a forwards pass along the trajectory (via Alg. 1.1) followed by a backwards pass. This means it is only applicable to episodic trajectories, and is not an on-line weight update. However using methods similar to the way the on-line VGL(λ) algorithm was calculated, it is possible to generate an on-line version of BPTT. This has been done by Williams and Zipser (1989) and is called “Real Time Recurrent Learning” (RTRL).

6.1.2 Convergence Properties of BPTT Algorithm

BPTT is gradient descent on the function $\widehat{J}(\vec{x}, \vec{\mathbf{e}}, \vec{z})$ with respect to $\vec{z}$. If the surface of $\widehat{J}(\vec{x}, \vec{\mathbf{e}}, \vec{z})$ is bounded below and smooth in $\vec{z}$ space, and if the learning rate β is sufficiently small, then convergence will be assured towards a local minimum of the surface.

In certain stochastic environments, the BPTT gradient calculation is only approximate, as detailed in Section 6.1.1. In these cases, it may be that the BPTT learning process is not true gradient descent, and this creates a small possibility of the convergence guarantee failing.

A separate relevant detail for BPTT in stochastic environments is that the mean of the BPTT gradient-descent equation will be $\mathbb{E}\left(\Delta\vec{z}\right) = -\beta\mathbb{E}\left(\frac{\partial\widehat{J}(\vec{x},\vec{\mathbf{e}},\vec{z})}{\partial\vec{z}}\right)$, and therefore if convergence to a local minimum is attained, then that minimum will satisfy,

$$\mathbb{E}\left(\frac{\partial\widehat{J}(\vec{x},\vec{\mathbf{e}},\vec{z})}{\partial\vec{z}}\right) = \vec{0}. \tag{6.5}$$

The local minimum described by Eq. (6.5) is subtly different from the minimum sought by the ADPRL objective which was to minimise $\mathbb{E}\left(\widehat{J}(\vec{x},\vec{\mathbf{e}},\vec{z})\right)$ with respect to $\vec{z}$, therefore resulting in,

$$\frac{\partial\mathbb{E}\left(\widehat{J}(\vec{x},\vec{\mathbf{e}},\vec{z})\right)}{\partial\vec{z}} = \vec{0}. \tag{6.6}$$

The fact that Eq. (6.5) $\neq$ Eq. (6.6) implies that the minimum attained by BPTT will not in general be exactly the same as the minimum sought by ADPRL, when the environment is stochastic. Therefore BPTT is seeking a minimum that is only an approximation of the ADPRL objective. However the following theorem gives a significant stochastic situation under which these two kinds of minima do become exactly equal.

Theorem 6.1. *If $\widehat{J}(\vec{x}_0,\vec{\mathbf{e}}_0,\vec{z})$ is a smooth function with respect to both $\vec{z}$ and $\vec{\mathbf{e}}_0$, and if the probability distribution of the noise vector $\vec{\mathbf{e}}_0$ is twice-differentiable and independent of $\vec{z}$ then,*

$$\mathbb{E}\left(\frac{\partial\widehat{J}(\vec{x}_0,\vec{\mathbf{e}}_0,\vec{z})}{\partial\vec{z}}\right) \equiv \frac{\partial\mathbb{E}\left(\widehat{J}(\vec{x}_0,\vec{\mathbf{e}}_0,\vec{z})\right)}{\partial\vec{z}}.$$

Proof. For a trajectory starting at state $\vec{x}_0$, the noise vector for the whole trajectory was defined in Section 1.2 to be $\vec{\mathbf{e}}_0$. This noise vector has a sample space $\mathbf{E}$ and probability distribution $P_{\mathbf{e}}\left(\vec{\mathbf{e}}_0\right)$. Therefore any function f of the random variable $\vec{\mathbf{e}}_0$ will have an expectation defined by,

$$\mathbb{E}\left(f(\vec{\mathbf{e}}_0)\right) := \int_{\mathbf{E}} P_{\mathbf{e}}\left(\vec{\mathbf{e}}\right) f(\vec{\mathbf{e}}) d\vec{\mathbf{e}}. \tag{6.7}$$

Therefore,

$$\begin{aligned}
\frac{\partial \mathbb{E}\left(\widehat{J}(\vec{x}_0, \vec{\mathbf{e}}_0, \vec{z})\right)}{\partial \vec{z}} &= \frac{\partial}{\partial \vec{z}} \left(\int_{\mathbf{E}} P_{\mathbf{e}}\left(\vec{\mathbf{e}}\right) \widehat{J}(\vec{x}_0, \vec{\mathbf{e}}, \vec{z}) d\vec{\mathbf{e}} \right) && \text{(by Eq. (6.7))} \\
&= \int_{\mathbf{E}} \frac{\partial}{\partial \vec{z}} \left(P_{\mathbf{e}}\left(\vec{\mathbf{e}}\right) \widehat{J}(\vec{x}_0, \vec{\mathbf{e}}, \vec{z}) \right) d\vec{\mathbf{e}} && \text{(exchange order of } \textstyle\int, \frac{\partial}{\partial \vec{z}}) \\
&= \int_{\mathbf{E}} P_{\mathbf{e}}\left(\vec{\mathbf{e}}\right) \frac{\partial}{\partial \vec{z}} \left(\widehat{J}(\vec{x}_0, \vec{\mathbf{e}}, \vec{z}) \right) d\vec{\mathbf{e}} && \text{(constant extracted)} \\
&= \mathbb{E}\left(\frac{\partial}{\partial \vec{z}} \left(\widehat{J}(\vec{x}_0, \vec{\mathbf{e}}_0, \vec{z}) \right) \right) && \text{(by Eq. (6.7))}
\end{aligned}$$

The above line that exchanges the order of the integral and differential is valid according the Leibniz Integral Rule, only if the integrand $\left(P_{\mathbf{e}}\left(\vec{\mathbf{e}}\right) \widehat{J}(\vec{x}, \vec{\mathbf{e}}, \vec{z})\right)$, and its derivative with respect to $\vec{z}$, are continuous with respect to both $\vec{\mathbf{e}}$ and $\vec{z}$. Also for the Leibniz Integral Rule to be valid, we require that the sample space $\mathbf{E}$ is not dependent on $\vec{z}$. All of these are preconditions of the theorem.

$\square$

This theorem proves that under certain conditions, the BPTT stochastic local minimum attained will be the correct minimum sought by the ADPRL optimization problem. The theorem makes the requirement for $\widehat{J}(\vec{x}, \vec{\mathbf{e}}, \vec{z})$ to be a smooth function of $\vec{z}$, however this was already a necessary condition for assured convergence by gradient descent, as already mentioned above. The theorem's additional requirements for $\widehat{J}$ to be a smooth function of $\vec{\mathbf{e}}$, and for the probability distribution $P_{\mathbf{e}}\left(\vec{\mathbf{e}}\right)$ to be smooth, are new necessary conditions placed on BPTT for true optimality. The requirement for the probability distribution $P_{\mathbf{e}}\left(\vec{\mathbf{e}}\right)$ to be independent of $\vec{z}$ is easily satisfied without placing any unnecessary constraints on the ADPRL problem definition, as described in Section 1.2.1.

It should be noted that the function $\widehat{J}(\vec{x}, \vec{\mathbf{e}}, \vec{z})$ is often not a smooth function of $\vec{z}$, even if the environment functions $f(\vec{x}, \vec{u}, \vec{e})$, $U(\vec{x}, \vec{u}, \vec{e})$ and $\Phi(\vec{x}, \vec{e})$ are smooth. For example, in the Vertical-Lander problem defined earlier, the line through state space given by $x_h = x_v = 0$ is a critical boundary between the spacecraft landing with a perfect zero velocity, and between the spacecraft missing the landing altogether and taking off again into a second ascent. As this critical boundary is crossed, the journey duration changes discontinuously, and therefore there is a large cliff in the surface of $\widehat{J}(\vec{x}, \vec{\mathbf{e}}, \vec{z})$ in $\vec{z}$-space along this critical boundary.

Certain other discontinuities in the $\widehat{J}$ surface exist due to the discrete sampling of the time step variable t. These discontinuities are described and solved in Chapter 10 using "clipping".

6.2 The REINFORCE Algorithm

REINFORCE is a policy-gradient learning (PGL) algorithm by Williams (1992). It does gradient descent on the total trajectory cost with respect to the weight vector of the action network. It was initially defined for a reinforcement problem for episodic trajectories with length of just one time step, and with no terminal trajectory cost $\Phi(\vec{x}, \vec{e})$. In this case the total trajectory cost is equal to just $\widehat{J}(\vec{x}_0, \mathbf{\vec{e}}_0, \vec{z}) \equiv U(\vec{x}_0, \vec{u}_0, \vec{e}_0)$. It is possible to extend REINFORCE to multi-step trajectories (see Williams, 1992, for details), but for simplicity, that is not considered in this thesis.

The algorithm is only defined for stochastic policies. It requires that the action $\vec{u}_0$ is a random variable, sampled from a probability distribution $P_u(\vec{u}_0|\vec{x}_0, \vec{z})$ for a given weight vector $\vec{z}$ and state input $\vec{x}_0$. Using this notation, the REINFORCE weight update is defined as:

$$\Delta\vec{z} = -\beta \frac{\partial \ln(P_u(\vec{u}_0|\vec{x}_0, \vec{z}))}{\partial \vec{z}} (U(\vec{x}_0, \vec{e}_0, \vec{u}_0) - b) \tag{6.8}$$

where $\beta > 0$ is small constant, and $b \in \mathbb{R}$ is an arbitrary constant "baseline" parameter.

For this weight update, Williams (1992) proves that it obeys

$$\mathbb{E}\left(\Delta\vec{z}\right) = -\beta \frac{\partial \mathbb{E}\left(U(\vec{x}_0, \vec{e}_0, \vec{u}_0)\right)}{\partial \vec{z}} \tag{6.9}$$

which means that the average of this weight update equals gradient descent on the average trajectory cost, with respect to $\vec{z}$. The expectation operator in the right-hand side of this equation is operating on the random variables $\vec{u}_0$ and $\vec{e}_0$. Williams' proof is summarised below, in Section 6.2.1.

The baseline parameter, b, does not appear in the above mean weight update (Eq. (6.9)), and so choosing this parameter in Eq. (6.8) is a free choice. However choosing an optimal value of b can make the variance of the stochastic weight update smaller, and with a small variance the expectation in Eq. (6.9) will form faster (through its accumulation through stochastic sampling). Hence the choice of b can affect learning speed. In PGL methods, minimising the variance of the stochastic weight update is a major research issue and goal, as described by Peters and Schaal (2006).

Since the weight update for REINFORCE satisfies Eq. (6.9), the minimum attained by REINFORCE will naturally solve the ADPRL objective given by Eq. (6.6). The expectation operator in the right-hand side of Eq. (6.9) has a smoothing effect so that, when combined with the stochastic policy described by the distribution $P_u(\vec{u}_0|\vec{x}_0, \vec{z})$, the surface of $\mathbb{E}\left(\widehat{J}\right)$ is smoother than the surface of $\widehat{J}$ in $\vec{z}$-space. This creates a benefit to REINFORCE in that it can cope with non-smooth surfaces of $\widehat{J}$ in $\vec{z}$-space, compared to BPTT which cannot easily cope with discontinuities in this surface.

6.2.1 Outline of Williams' proof

This section gives a outline of the key sequence of ideas from Williams' proof of how the weight update given by Eq. (6.8) produces a mean given by Eq. (6.9). In this section we will consider the environment to be deterministic. This choice is made to simplify the equations, because then the only random variable is $\vec{u}_0$. Hence the cost function $U(\vec{x}_0, \vec{u}_0)$ will omit the noise argument, $\vec{e}$.

Since the random variable $\vec{u}_0$ is sampled from the probability distribution $P_u(\vec{u}_0|\vec{x}_0, \vec{z})$, any function f of this random variable $\vec{u}_0$ will have an expectation defined by:

$$\mathbb{E}\left(f(\vec{u}_0)\right) := \int_{\mathbb{A}} P_u(\vec{u}|\vec{x}_0, \vec{z}) f(\vec{u}) d\vec{u}. \tag{6.10}$$

Williams' proof, when modified to apply to a continuous-valued action space $\mathbb{A}$ and deterministic $U(\vec{x}_0, \vec{u}_0)$ function, is as follows:

$$\begin{aligned}
\mathbb{E}\left(\Delta\vec{z}\right) &= -\beta\mathbb{E}\left(\frac{\partial \ln(P_u(\vec{u}_0|\vec{x}_0, \vec{z}))}{\partial\vec{z}}(U(\vec{x}_0, \vec{u}_0) - b)\right) && \text{(by Eq. (6.8))}\\
&= -\beta\mathbb{E}\left(\frac{1}{P_u(\vec{u}_0|\vec{x}_0, \vec{z})}\frac{\partial P_u(\vec{u}_0|\vec{x}_0, \vec{z})}{\partial\vec{z}}(U(\vec{x}_0, \vec{u}_0) - b)\right) && \text{(by chain rule)}\\
&= -\beta\int_{\mathbb{A}}\frac{\partial P_u(\vec{u}|\vec{x}_0, \vec{z})}{\partial\vec{z}}(U(\vec{x}_0, \vec{u}) - b)d\vec{u} && \text{(by Eq. (6.10))}\\
&= -\beta\frac{\partial}{\partial\vec{z}}\int_{\mathbb{A}} P_u(\vec{u}|\vec{x}_0, \vec{z})(U(\vec{x}_0, \vec{u}) - b)d\vec{u} && \text{(commutative)}\\
&= -\beta\frac{\partial\mathbb{E}\left(U(\vec{x}_0, \vec{u}_0) - b\right)}{\partial\vec{z}} && \text{(by Eq. (6.10))}
\end{aligned}$$

$$= -\beta \frac{\partial \mathbb{E}\left(U(\vec{x}_0, \vec{u}_0)\right)}{\partial \vec{z}}. \qquad \text{(Since } b \text{ is constant)} \tag{6.11}$$

This proof has made the extension to continuous-valued action spaces $\mathbb{A}$ compared to the original proof by Williams, and is a streamlined version of the original proof.[10] In the above sequence of algebra, the step labelled "commutative" is under the assumption that the integrand $P_u(\vec{u}_0|\vec{x}_0, \vec{z})(U(\vec{x}_0, \vec{u}_0) - b)$, and its derivatives with respect to $\vec{z}$, are continuous with respect to both $\vec{u}_0$ and $\vec{z}$, because integration and differentiation can only ever be assured to commute under these circumstances (as noted in the proof of Theorem 6.1). Williams' version of this proof did not need this assumption because his proof applied to discrete-valued action spaces $\mathbb{A}$.

6.3 Proof of Equivalence under Certain Circumstances of BPTT to PGL

Both BPTT and PGL work by gradient descent on the total trajectory cost with respect to $\vec{z}$, the weight vector of the action network. However it is curious that the BPTT weight update relies upon the terms $\frac{\partial A(\vec{x},\vec{e},\vec{z})}{\partial \vec{z}}$ and $\frac{\partial U(\vec{x},\vec{u},\vec{e})}{\partial \vec{u}}$, but the PGL weight update does not explicitly use them anywhere. It turns out that these two derivative terms are key in proving equivalence between BPTT and VGL(λ), in Chapter 8, and hence we now analyse whether the PGL weight update can ever produce these two terms. Apart from mere curiosity, this analysis is potentially useful to try to create a theoretical bridge between these two different policy-gradient methods, which may be useful one day for transferring convergence proofs or model-free behaviour, or creating hybrid algorithms, between different research areas of ADPRL.

For simplicity, this section will only consider a one-dimensional action vector, i.e. $\vec{u}_0 \equiv u_0 \in \mathbb{R}$, and $\mathbb{A} \equiv (-\infty, \infty)$, and also only consider the case where the cost function $U(\vec{x}, \vec{u})$ is deterministic (and hence omits the $\vec{e}$ argument).

[10] For correspondence between the two proofs, this proof is equivalent to (Williams, 1992, Lemma 1). The line commented "(commutative)" corresponds to Williams' Fact 1, but this thesis' version is modified to a continuous-valued action space. The final line ("since b is constant") corresponds to Williams' application of (Williams, 1992, "Fact 2"). Williams' proof goes one step further and also extends the expectation operator to apply to a random start state $\vec{x}_0$ too, which would also be a straight-forward extension of this proof. Note that Williams' notation uses $g(\cdot)$ in place of $P_u(\cdot)$.

First note that when the trajectory length is one time-step, the BPTT weight update simplifies into,

$$\begin{aligned}\Delta\vec{z} &= -\beta\frac{\partial\widehat{J}(\vec{x}_0,\vec{\mathbf{e}}_0,\vec{z})}{\partial\vec{z}} \\ &= -\beta\frac{\partial U(\vec{x}_0,A(\vec{x}_0,\vec{e}_0,\vec{z}))}{\partial\vec{z}} \\ &= -\beta\left(\frac{\partial A}{\partial\vec{z}}\right)_0\left(\frac{\partial U}{\partial u}\right)_0 .\end{aligned} \tag{6.12}$$

In this section, we want to show how the very different looking REINFORCE weight update (Eq. (6.8)) can be made to look like Eq. (6.12).

To find this connection, a key difference that needs to be overcome is that the BPTT policy function $A(\vec{x},\vec{e},\vec{z})$ specifies the actual action value $\vec{u}_0$ to be taken, by Eq. (1.3), but the REINFORCE method explicitly specifies a full probability distribution $P_u(u_0|\vec{x}_0,\vec{z})$ to choose its action. To bridge this difference between the two methods, we can assume that the probability distribution $P_u(u_0|\vec{x}_0,\vec{z})$ is some additive noise on top of some deterministic single-valued function, $A(\vec{x},\vec{z})$. In effect we are assuming the full distribution $P_u(u_0|\vec{x}_0,\vec{z})$ is parameterised by a single number $A(\vec{x},\vec{z})$. This assumption is not too restrictive and is not too dissimilar to probability distributions considered by (Williams, 1992, sec. 7.2) which are parameterised by just one or two numbers (e.g. a mean and a standard deviation).

To make this requirement on the probability distribution $P_u(u_0|\vec{x}_0,\vec{z})$ more precise, and also for the following equivalence proof to work, we need to assume that $P_u(u_0|\vec{x}_0,\vec{z})$ depends only on the *relative* values of u_0 and $A(\vec{x}_0,\vec{z})$, i.e. it can be written

$$P_u(u_0|\vec{x}_0,\vec{z}) \equiv p(u_0 - A(\vec{x}_0,\vec{z})), \tag{6.13}$$

for some real-valued probability distribution $p(x)$. This assumption means we are assuming the whole probability distribution is clustered around the single value $A(\vec{x}_0,\vec{z})$, and if this single value shifts left or right, then so will the whole distribution curve.

Also, we assume that this probability distribution $p(x)$ becomes zero as x becomes infinitely large or small, i.e. we assume

$$\lim_{x\to\pm\infty} p(x) = 0; \tag{6.14}$$

and we also assume that the function $U(\vec{x}_0, u)$ is bounded, i.e.

$$|U(\vec{x}_0, u)| < k, \quad \forall u \in \mathbb{A} \tag{6.15}$$

for some fixed $k > 0 \in \mathbb{R}$.

We also need to make use of the following lemma:

Lemma 6.1. *For any real-valued differentiable function $f : \mathbb{R} \to \mathbb{R}$, we have*

$$\frac{\partial f(x-y)}{\partial x} \equiv -\frac{\partial f(x-y)}{\partial y}.$$

Proof.

$$\begin{aligned}
\frac{\partial f(x-y)}{\partial x} &= \frac{\partial f(x-y)}{\partial (x-y)} \frac{\partial (x-y)}{\partial x} && \text{(by chain rule)} \\
&= \frac{\partial f(x-y)}{\partial (x-y)} \\
&= \frac{\partial f(x-y)}{\partial y} \frac{\partial y}{\partial (x-y)} && \text{(by chain rule)} \\
&= -\frac{\partial f(x-y)}{\partial y}.
\end{aligned}$$

□

We can now make the equivalence proof of the REINFORCE weight update to the BPTT weight update, as follows: Starting from the third line of the proof leading to Eq. (6.11) above, we have:

$$\begin{aligned}
\mathbb{E}(\Delta \vec{z}) &= -\beta \int_{-\infty}^{\infty} \frac{\partial P_u(u|\vec{x}_0, \vec{z})}{\partial \vec{z}} (U(\vec{x}_0, u) - b) du \\
&= -\beta \int_{-\infty}^{\infty} \frac{\partial p(u - A(\vec{x}_0, \vec{z}))}{\partial \vec{z}} (U(\vec{x}_0, u) - b) du && \text{(by Eq. (6.13))} \\
&= -\beta \int_{-\infty}^{\infty} \frac{\partial A(\vec{x}_0, \vec{z})}{\partial \vec{z}} \frac{\partial p(u - A(\vec{x}_0, \vec{z}))}{\partial A(\vec{x}_0, \vec{z})} (U(\vec{x}_0, u) - b) du && \text{(chain rule)} \\
&= -\beta \frac{\partial A(\vec{x}_0, \vec{z})}{\partial \vec{z}} \int_{-\infty}^{\infty} \frac{\partial p(u - A(\vec{x}_0, \vec{z}))}{\partial A(\vec{x}_0, \vec{z})} (U(\vec{x}_0, u) - b) du && \text{(constant extracted)} \\
&= \beta \frac{\partial A(\vec{x}_0, \vec{z})}{\partial \vec{z}} \int_{-\infty}^{\infty} \frac{\partial p(u - A(\vec{x}_0, \vec{z}))}{\partial u} (U(\vec{x}_0, u) - b) du && \text{(by Lemma 6.1)} \\
&= \beta \frac{\partial A(\vec{x}_0, \vec{z})}{\partial \vec{z}} \Big(\left[p(u - A(\vec{x}_0, \vec{z}))(U(\vec{x}_0, u) - b) \right]_{u=-\infty}^{u=\infty}
\end{aligned}$$

$$
\begin{aligned}
&\quad - \int_{-\infty}^{\infty} p(u - A(\vec{x}_0,\vec{z})) \frac{\partial (U(\vec{x}_0,u) - b)}{\partial u} du \Big) && \text{(integration by parts)} \\
&= -\beta \frac{\partial A(\vec{x}_0,\vec{z})}{\partial \vec{z}} \int_{-\infty}^{\infty} p(u - A(\vec{x}_0,\vec{z})) \frac{\partial (U(\vec{x}_0,u) - b)}{\partial u} du && \text{(by Eqs. (6.14)\&(6.15))} \\
&= -\beta \frac{\partial A(\vec{x}_0,\vec{z})}{\partial \vec{z}} \int_{-\infty}^{\infty} P_u(u|\vec{x}_0,\vec{z}) \frac{\partial (U(\vec{x}_0,u) - b)}{\partial u} du && \text{(by Eq. (6.13))} \\
&= -\beta \frac{\partial A(\vec{x}_0,\vec{z})}{\partial \vec{z}} \int_{-\infty}^{\infty} P_u(u|\vec{x}_0,\vec{z}) \frac{\partial U(\vec{x}_0,u)}{\partial u} du && \text{(since } b \text{ is constant)} \\
&= -\beta \frac{\partial A(\vec{x}_0,\vec{z})}{\partial \vec{z}} \mathbb{E}\left(\frac{\partial U(\vec{x}_0,u_0)}{\partial u_0} \right). && \text{(by Eq. (6.10))}
\end{aligned}
$$

This completes the demonstration of how the terms $\frac{\partial A(\vec{x}_0,\vec{z})}{\partial \vec{z}}$ and $\frac{\partial U(\vec{x}_0,u_0)}{\partial u_0}$ can be found to appear from the REINFORCE weight update, under the assumptions given by Eqs. (6.13) to (6.15), and this demonstration strengthens the theoretical connection between REINFORCE and BPTT. It is the integration-by-parts step in the above proof that transfers the differential operator over from the probability distribution $P_u(u|\vec{x}_0,\vec{z})$ onto the cost function $U(\vec{x},\vec{u},\vec{e})$; and the $\frac{\partial A(\vec{x},\vec{z})}{\partial \vec{z}}$ term arises from use of the chain rule and the assumption that probability distribution $P_u(u|\vec{x}_0,\vec{z})$ can be parameterised about the value $A(\vec{x},\vec{z})$.

6.4 Chapter Conclusions

BPTT and REINFORCE are two methods which work by stochastic gradient descent on the function $J(\vec{x},\vec{\mathbf{e}},\vec{z})$ with respect to $\vec{z}$. These two algorithms have been derived and stated in this chapter, and a theoretical connection between them has been given in Section 6.3.

Although these two methods go about achieving the same gradient-descent goal, the two methods differ subtly, and their algorithmic implementations look very different. To summarise the key differences between the two methods, we have:

- BPTT is a model-based method but REINFORCE is a model-free method.

- BPTT does automatic local value exploration; REINFORCE needs explicit stochastic value exploration. This is analogous to the difference in the value-exploration needs between VGL(λ) and TD(λ) as explained in Section 3.1.

- BPTT can work with either a stochastic or a deterministic policy function, but REINFORCE is designed explicitly to work with a stochastic policy function only.

- The gradient calculation performed by BPTT will only be approximate, unless one of the conditions highlighted at the end of Section 3.2.1 (e.g. additive noise in the environment functions) applies. The REINFORCE gradient calculation works with general noise distributions in the environment functions without any approximation.

- BPTT gets an effective learning gradient in just one trajectory sample, but REINFORCE must take multiple trajectory samples in order to accumulate the mean gradient-descent weight update. Reducing the variance in the REINFORCE weight update (and in related PGL algorithms) is a major research topic, but it is not a concern for BPTT based algorithms.

- In a stochastic environment, after several trajectory samples from the same fixed start point $\vec{x}$, BPTT would accumulate the mean $\mathbb{E}\left(\frac{\partial \widehat{J}}{\partial \vec{x}}\right)$, but the REINFORCE weight update would accumulate the mean $\frac{\partial \mathbb{E}(\widehat{J})}{\partial \vec{x}}$. Although these look similar, they are subtly different, because in general the mean of a derivative is different from the derivative of a mean. This can result in BPTT attaining a different minimum of $J(\vec{x}, \vec{\mathbf{e}}, \vec{z})$ than is intended by the ADPRL optimization objective, although the problem disappears under certain smoothness conditions, as detailed in Section 6.1.2.

- BPTT requires smoothness of the function $\widehat{J}(\vec{x}, \vec{\mathbf{e}}, \vec{z})$ with respect to $\vec{z}$, but REINFORCE does not.

The final point above gives an advantage for REINFORCE in that it does not need to consider “clipping”, which is a technical issue that needs addressing for BPTT and related methods (such as VGL(λ)) to be implemented correctly. Clipping is described fully in Chapter 10.

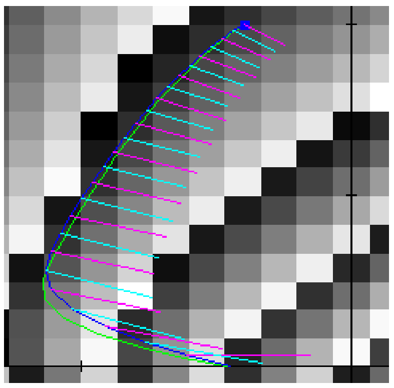
Part II

Theoretical Results

Chapter 7

Trajectory Local Optimality

It has been described in earlier chapters that learning the value-gradients along a trajectory can bend the trajectory into a locally extremal, and often locally optimal, shape. This fact was described in Fig. 3.6, which is repeated here in Fig. 7.1. This trajectory-optimality theorem is proven in this chapter. The theorem is only proven for deterministic environments, and is only applicable for differentiable environment functions and for continuous-valued state and action spaces.



Key:
Blue line: Actual trajectory
Green line: Theoretical optimal trajectory
Magenta lines: Target gradients (G')
Cyan lines: Actual gradients ($\widetilde{G}$)

Figure 7.1: Trajectory Optimality Occurs when the Value-Gradients are Learned, under a Greedy Policy. This picture, repeated from Fig. 3.6, shows the key fact motivating VGL methods compared to VL methods: Merely learning the target value-gradients, under a greedy policy, will necessarily bend the trajectory into a locally extremal or locally optimal shape. Hence this picture concisely illustrates the trajectory-optimality principle of VGL, which is proven for deterministic environments in this chapter.

This optimality result explains why learning the value-gradients will automatically bend trajectories into locally optimal shapes, and thus explain the automatic local value-exploration exhibited by VGL methods. With VL methods, an optimality condition equivalent to this does not exist, but counterexamples do (e.g. Fig. 3.1).

Note that, as with the optimality target condition for TD(λ), i.e. $\widetilde{J}_t = \mathbb{E}\left(J'_t\right)$ for all $\vec{x}_t \in \mathbb{S}$, it will not in general be possible to attain the VGL objective $\widetilde{G}_t = \mathbb{E}\left(G'_t\right)$ for all t exactly, due to the limitations of function approximation. But under the assumption of using a universal function approximator, it would thoeretically be possible to get arbitrarily close to this objective. Also, assuming that the greedy policy changes smoothly when the value-gradient changes (which is a reasonable assumption made and justified in Lemma 8.4 of Chapter 8), and assuming that the total trajectory cost $J(\vec{x}_0, \vec{z})$ is a smooth function of the actions chosen (which is a necessary assumption for assured convergence of BPTT, as discussed in Section 6.1.2), it follows that getting arbitrarily close to the VGL objective will result in getting arbitrarily close to the trajectory local-extremality/optimality result proven in this chapter.

Also note that this chapter does not describe how to ensure progress is made towards the VGL objective $\widetilde{G}_t = \mathbb{E}\left(G'_t\right)$; this is dealt with in the convergence proof chapter (Chapter 8) only for the algorithm variant VGLΩ(1). For other values of λ, empirical results are available in Chapters 3 and 5 which show learning progress can be made, however Chapter 9 gives a specific divergence example showing learning progress can also fail.

The proof in this chapter will make some simplifying assumptions:

1. Only deterministic environments are considered. It would possibly require significant further work to extend the result to stochastic environments.

2. It is assumed environment functions are perfectly learned (consistent with Eq. (1.10)).

3. It is assumed that the action space, $\mathbb{A}$, is such that it ensures each component i of the action vector $\vec{u}$ is either unbound (i.e. $(\vec{u})^i \in \mathbb{R}$) or is bound to $(\vec{u})^i \in [-1, 1]$. It would be fairly straightforward to extend the analysis to include $\mathbb{A}$ being any arbitrary cuboid or spheroid; but for clarity, this has been omitted.

By the first assumption (determinism), three simplifications to the ADPRL formulation can be made:

1. We can omit the $\vec{e}$ argument from the environment functions and policy function, and hence just use $f(\vec{x},\vec{u})$, $U(\vec{x},\vec{u})$, $\Phi(\vec{x})$ and $A(\vec{x},\vec{z})$ instead.

2. We can omit the expectation operator from the expression, $\mathbb{E}\left(G'_t\right)$, since the environment is not stochastic. Hence the VGL training objective simplifies down to getting as close as possible to $\widetilde{G}_t = G'_t$ for all t along a greedy trajectory.

3. Similarly, since the environment is deterministic, and since we are also assuming the environment functions are perfectly learned (i.e. we assume Eq. (1.10) holds), we have,

$$\begin{aligned} f(\vec{x},\vec{u}) &\equiv \bar{f}(\vec{x},\vec{u}) \\ U(\vec{x},\vec{u}) &\equiv \bar{U}(\vec{x},\vec{u}) \\ \Phi(\vec{x}) &\equiv \bar{\Phi}(\vec{x}). \end{aligned} \tag{7.1}$$

In rest of this chapter, in Section 7.1, “greedy actions” and “saturated actions” are defined. Saturated actions are actions which lie on the boundary of $\mathbb{A}$, and the possibility of their existence makes a significant complication to the proof in this chapter. In Section 7.2, a more basic version of the cost-to-go function is defined, and this is used to define exactly what is meant by an “optimal trajectory”. In Section 7.3 the main optimality result of this chapter is proven. Several lemmas lead up to this result, and the main result itself comes in Theorem 7.1. The close connection of this theorem to Pontryagin’s Minimum Principle (PMP) is described in Section 7.3.3. The theorem proven in this chapter gives an extra corollary which makes the optimality result stronger than that provided by PMP alone. Finally, chapter conclusions are given in Section 7.4.

7.1 Greedy and Saturated Actions

A greedy policy chooses actions dependent on $\widetilde{J}(\vec{x},\vec{w})$ as defined by Eq. (1.7), which ensures for any $\vec{x}\in\mathbb{S}$:

$$\vec{u} := \arg\min_{\vec{u}\in\mathbb{A}} \left[U(\vec{x},\vec{u}) + \gamma\widetilde{J}(f(\vec{x},\vec{u}),\vec{w}) \right]. \tag{7.2}$$

where $\mathbb{A}$ is the space of all possible action vectors $\vec{u}$, and where the assumption of a deterministic environment has been used (so that the expectation operator and noise argument, $\vec{e}$, have been omitted here, compared to Eq. (1.7)).

The model-based approximate $\widetilde{Q}$ function definition was defined in Eq. (3.9) as,

$$\widetilde{Q}(\vec{x},\vec{u},\vec{w}) := \bar{U}(\vec{x},\vec{u}) + \gamma \widetilde{J}(\bar{f}(\vec{x},\vec{u}),\vec{w}). \tag{7.3}$$

By using Eq. (7.1) to interchange the learned environment functions with the actual environment functions, the greedy policy (Eq. (7.2)) can be rewritten as,

$$\vec{u} = \arg\min_{\vec{u}\in\mathbb{A}}(\widetilde{Q}(\vec{x},\vec{u},\vec{w})), \quad \text{for any } \vec{x}\in\mathbb{S}. \tag{7.4}$$

The $\widetilde{Q}$ function used here is defined to be smooth, because all of its constituent functions are smooth. For example, the functions $\bar{f}(\vec{x},\vec{u})$ and $\bar{U}(\vec{x},\vec{u})$ were defined in Section 3.2 to be smooth, as a prerequisite to using VGL methods. Also, the function $\widetilde{J}(\vec{x},\vec{w})$ is defined to be the output of a critic-network, i.e. a smooth function approximator. Therefore the function $\widetilde{Q}(\vec{x},\vec{u},\vec{w})$ is differentiable, and its derivative with respect to $\vec{u}$ is,

$$\left(\frac{\partial \widetilde{Q}}{\partial \vec{u}}\right)_t = \left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \widetilde{G}_{t+1}. \tag{7.5}$$

This derivative was found by differentiating Eq. (7.3) using the chain rule, and substituting $\left(\frac{\partial \widetilde{J}}{\partial \vec{x}}\right)_{t+1} \equiv \widetilde{G}_{t+1}$, by Eq. (3.3).

The action space $\mathbb{A}$ can be used to impose bounds on some actions or action components. For example in some problems, some action components $\vec{u}^i$ may be bound to $\vec{u}^i \in [-1,1]$, or any other given range (but for simplification, only the range $[-1,1]$ will be considered in this chapter).[11]

A *greedy action* is any action $\vec{u}$ that satisfies Eq. (7.4). Therefore all actions chosen by a greedy policy will be greedy actions, and any action chosen by an action network which happens to satisfy Eq. (7.4) will also be classified as a greedy action.

A *greedy trajectory* is a trajectory in which all of the actions that parameterise it are greedy actions. The optimality condition in this chapter is only relevant to greedy trajectories.

[11]Throughout this chapter upper indices indicate vector component.

In preparation for the optimality proof of Section 7.3, the effects of bound actions need to be fully accounted for, i.e. if the constraints $\vec{u}^i \in [-1,1]$ are present for some action components. Bound actions lead to what will be called *saturated actions*, which are described in the following subsection.

7.1.1 Saturated Actions

When the constraints $\vec{u}^i \in [-1,1]$ are present for some action component $\vec{u}^i$, the action component is *saturated* if $\left|\vec{u}^i\right| = 1$ and $\frac{\partial \widetilde{Q}}{\partial \vec{u}^i} \neq 0$. If either of these two conditions is not met, or the constraints are not present, then the action component $\vec{u}^i$ is not saturated. For example if an action component represents the steering wheel of a car, then that action component is saturated when the steering wheel is rotated to its full limit in either direction, with pressure being applied against that limit.

Some useful lemmas about saturated greedy actions are derived in this section. These are closely related to the Karush-Kuhn-Tucker (KKT) conditions of constrained-optimization theory. However in some cases the lemmas here derive strict inequalities, whereas the KKT conditions would yield non-strict inequalities. These lemmas now follow:

Lemma 7.1. *For a greedy action $\vec{u}$, if a component $\vec{u}^i$ is saturated with $\vec{u}^i = 1$ then $\frac{\partial \widetilde{Q}}{\partial \vec{u}^i} < 0$; and if a component $\vec{u}^i$ is saturated with $\vec{u}^i = -1$ then $\frac{\partial \widetilde{Q}}{\partial \vec{u}^i} > 0$.*

Proof. The first of these two conditional statements has to be true since for a saturated action $\frac{\partial \widetilde{Q}}{\partial \vec{u}^i} \neq 0$ by definition, and if $\frac{\partial \widetilde{Q}}{\partial \vec{u}^i} > 0$ at $\vec{u}^i = 1$ then the minimum of $\widetilde{Q}$ would not be at $\vec{u}^i = 1$, which contradicts the greedy-action condition. The second conditional statement is true for the same reason with the situation reversed. □

Lemma 7.2. *For a greedy action $\vec{u}$, if action component $\vec{u}^i$ is saturated, then, whenever it exists, $\frac{\partial A^i}{\partial \vec{x}} = \vec{0}$.*

Proof. Supposing, $\vec{u}^i = 1$, then $\frac{\partial \widetilde{Q}}{\partial \vec{u}^i} = k$, for some *finite* $k < 0$, by Lemma 7.1. Therefore after any *infinitesimal* change to $\vec{x}$ of the smooth function $\widetilde{Q}(\vec{x}, \vec{u}, \vec{w})$, it will still be true that $\frac{\partial \widetilde{Q}}{\partial \vec{u}^i} < 0$ at $\vec{u}^i = 1$, so we still have a minimum of $\widetilde{Q}$ at $\vec{u}^i = 1$. Therefore, for a greedy policy, which is designed to choose a minimum of $\widetilde{Q}$, we must have $\frac{\partial A^i}{\partial \vec{x}} = \vec{0}$, when it exists.

The proof so far has only dealt with the situation where the greedy action under consideration was generated by a greedy policy. However the greedy action may also have come from an action network, so this possibility also needs dealing with.

If we are using an action network instead of the greedy policy, then the function $A^i(\vec{x},\vec{w})$ is the output of a smooth neural network. Since the action component is saturated, the bounds $(\vec{u})^i \in [-1,1]$ must be present, and therefore the function $A(\vec{x},\vec{z})$ must have been designed with an output range $A^i(\vec{x},\vec{w}) \in [-1,1]$. Then, since $A^i(\vec{x},\vec{w}) = 1$, this is the highest value that $A^i(\vec{x},\vec{w})$ can attain, and therefore we are at a maximum. Also, the action-network $A(\vec{x},\vec{z})$ was designed to be smooth. Therefore $\frac{\partial A^i}{\partial \vec{x}} = \vec{0}$, which is a property of all differentiable local maxima, from basic calculus.

This completes the proof for the case of $\vec{u}^i = 1$. A similar argument can be made for the case of $\vec{u}^i = -1$. □

Lemma 7.3. *For a greedy action $\vec{u}$, if $\vec{u}^i$ is unsaturated, then $\frac{\partial \widetilde{Q}}{\partial \vec{u}^i} = 0$.*

Proof. This is true because the greedy-action condition means $\vec{u}^i$ is at a local minimum of the $\widetilde{Q}$ function with respect to $\vec{u}$. □

Lemma 7.4. *For any greedy action $\vec{u}$, regardless of whether any components are saturated or not, whenever $\frac{\partial A}{\partial \vec{x}}$ exists, we have $\left(\frac{\partial A}{\partial \vec{x}}\right)\left(\frac{\partial \widetilde{Q}}{\partial \vec{u}}\right) = \vec{0}$.*

Proof. The inner product is defined by $\left(\frac{\partial A}{\partial \vec{x}}\right)\left(\frac{\partial \widetilde{Q}}{\partial \vec{u}}\right) = \sum_i \frac{\partial A^i}{\partial \vec{x}} \frac{\partial \widetilde{Q}}{\partial \vec{u}^i}$. For each term of this sum, the greedy-action condition implies either $\frac{\partial A^i}{\partial \vec{x}} = \vec{0}$ (in the case that action component $\vec{u}^i$ is saturated and $\frac{\partial A^i}{\partial \vec{x}}$ exists, by Lemma 7.2), or $\frac{\partial \widetilde{Q}}{\partial \vec{u}^i} = 0$ (in the case that $\vec{u}^i$ is not saturated, by Lemma 7.3). Hence each term of the sum is zero, hence the sum is zero. □

7.2 Definitions Relating to Locally-Optimality Trajectories

In this section, locally optimal trajectories and their related functions are defined.

7.2.1 Total Trajectory-Cost Function, $\bar{J}$

First the total cost for a given trajectory is defined, in a way that is irrespective of the policy that was used to find it. For any trajectory starting at state $\vec{x}_0$ and following actions $(\vec{u}_0, \vec{u}_1, \ldots)$ under the given model, the total trajectory-cost encountered is given by the function:

$$\bar{J}(\vec{x}_0, \vec{u}_0, \vec{u}_1, \ldots) := \sum_{t=0}^{T-1} \gamma^t U(\vec{x}_t, \vec{u}_t) + \gamma^T \Phi(\vec{x}_T) \tag{7.6}$$

where we require $\gamma < 1$, or the problem to be episodic, for this sum to converge; and where T is the time step at which the first terminal state is reached (which in general will be dependent on $\vec{x}_0, \vec{u}_0, \vec{u}_1, \ldots$). If the trajectory never reached a terminal state then the $\Phi(\vec{x}_T)$ term would be omitted from the above definition, and T replaced by ∞.

The above definition of $\bar{J}$ is very similar looking to the definition of $\widehat{J}(\vec{x}, \vec{\mathbf{e}}, \vec{z})$ in Eq. (1.4), but the key difference (apart from the fact that in this chapter we are assuming determinism, but the $\widehat{J}(\vec{x}, \vec{\mathbf{e}}, \vec{z})$ function is for stochastic trajectories) is that $\bar{J}$ depends upon a full list of actions $(\vec{u}_0, \vec{u}_1, \ldots)$, whereas $\widehat{J}(\vec{x}, \vec{\mathbf{e}}, \vec{z})$ depends upon the action network's weight vector $\vec{z}$. The reason for making the separate definition of $\bar{J}$ is that whether a trajectory is optimal or not is irrespective of the action network used to find it, but is dependent on which actions parameterise that trajectory; hence $\bar{J}$ is a useful function for defining optimality, and making the trajectory-optimality proof of this chapter.

The above definition of $\bar{J}$ can be defined recursively as

$$\bar{J}(\vec{x}_t, \vec{u}_t, \vec{u}_{t+1}, \ldots) := \begin{cases} U(\vec{x}_t, \vec{u}_t) + \gamma \bar{J}(f(\vec{x}_t, \vec{u}_t), \vec{u}_{t+1}, \vec{u}_{t+2}, \ldots) & \text{if } \vec{x}_t \notin \mathbb{T} \\ \Phi(\vec{x}_t) & \text{if } \vec{x}_t \in \mathbb{T}, \end{cases} \tag{7.7}$$

which is valid for both episodic and continuing (non-episodic) trajectories. Thus $\bar{J}$ is a function of an arbitrary state $\vec{x}_t$ and all of the actions from that time step onwards.

7.2.2 Trajectory-Shorthand Notation for $\bar{J}$

The trajectory-shorthand notation defined in Section 2.1.2 is extended to include the function $\bar{J}$. For this extended trajectory-shorthand notation, for any given trajectory, we define $\bar{J}_t := \bar{J}(\vec{x}_t, \vec{u}_t, \vec{u}_{t+1}, \ldots)$. This enables the partial derivatives to be defined as $\left(\frac{\partial \bar{J}}{\partial \vec{x}}\right)_t := \frac{\partial \bar{J}_t}{\partial \vec{x}_t}$ and $\left(\frac{\partial \bar{J}}{\partial \vec{u}}\right)_t := \frac{\partial \bar{J}_t}{\partial \vec{u}_t}$.

7.2.3 Locally Optimal Trajectories

A trajectory parameterised by values $(\vec{x}_0, \vec{u}_0, \vec{u}_1, \vec{u}_2, \ldots)$ is defined to be locally optimal if $\bar{J}(\vec{x}_0, \vec{u}_0, \vec{u}_1, \vec{u}_2, \ldots)$ is at a local minimum with respect to the parameters $(\vec{u}_0, \vec{u}_1, \vec{u}_2, \ldots)$, subject to the constraints (if present) that $(\vec{u}_t)^i \in [-1, 1]$ for each action component i.

7.2.4 Locally Extremal Trajectories

Now locally extremal trajectories (LETs) are defined. Extremal means it could be at a local maximum of $\bar{J}$, or a local minimum of $\bar{J}$, or a point of inflection or saddle point of $\bar{J}$. All of these three possibilities are stationary points, which means we can use calculus to identify a LET. However the possibility of bound actions means that it is not necessarily true that $\left(\frac{\partial \bar{J}}{\partial \vec{u}}\right)_t = 0$ for all t; if an action is pushed right up against its limit (i.e. when the action is saturated) then usually $\left(\frac{\partial \bar{J}}{\partial \vec{u}}\right)_t \neq 0$. Considering this complication, we can define a LET as follows:

We define a trajectory parameterised by values $(\vec{x}_0, \vec{u}_0, \vec{u}_1, \vec{u}_2, \ldots)$ to be locally extremal if, for all t and all action components i,

$$\begin{cases} \left(\frac{\partial \bar{J}}{\partial \vec{u}^i}\right)_t = 0 & \text{if } (\vec{u}_t)^i \text{ is not saturated} \\ \left(\frac{\partial \bar{J}}{\partial \vec{u}^i}\right)_t < 0 & \text{if } (\vec{u}_t)^i \text{ is saturated and } (\vec{u}_t)^i = 1 \\ \left(\frac{\partial \bar{J}}{\partial \vec{u}^i}\right)_t > 0 & \text{if } (\vec{u}_t)^i \text{ is saturated and } (\vec{u}_t)^i = -1. \end{cases} \tag{7.8}$$

In the case that all the actions are unbound, this criterion for a LET simplifies to that of just requiring $\left(\frac{\partial \bar{J}}{\partial \vec{u}}\right)_t = \vec{0}$ for all t, which is a standard sufficient condition for a stationary point. This justifies the first condition in the above definition of a LET.

The second and third conditions of Eq. (7.8) address the extra complication of saturated actions. These two conditions are justified since they imply that $\bar{J}$ is locally optimal with respect to any saturated action components $(\vec{u}_t)^i$, since any small change of $(\vec{u}_t)^i$ inwards from the boundary $\left|(\vec{u}_t)^i\right| = 1$ will cause an immediate increment in $\bar{J}$, by the second and third conditions of Eq. (7.8).

A further consequence of this definition of a LET is that if all of the actions are fully saturated (a situation known as *bang-bang control*), then this definition of a LET provides a sufficient condition for a locally optimal trajectory. This can be formally stated as the following Lemma:

Lemma 7.5. *If a locally extremal trajectory is such that all of the actions are saturated in every component, then the LET is locally optimal.*

Proof. The definition of a LET by Eq. (7.8), when all the actions are saturated, simplifies down to:

$$\begin{cases} \left(\frac{\partial \bar{J}}{\partial \vec{u}^i}\right)_t < 0 & \text{if } (\vec{u}_t)^i = 1 \\ \left(\frac{\partial \bar{J}}{\partial \vec{u}^i}\right)_t > 0 & \text{if } (\vec{u}_t)^i = -1. \end{cases} \tag{7.9}$$

Then, because the actions are saturated in every component, we have $(\vec{u}_t)^i = \pm 1$, for all components i and every time step t. If $(\vec{u}_t)^i = 1$ and if that action component is changed by any amount, then that change to $(\vec{u}_t)^i$ can only be a reduction (since $-1 \leq (\vec{u}_t)^i \leq 1$). When this reduction happens, $\bar{J}$ will initially increase a bit, since $\left(\frac{\partial \bar{J}}{\partial \vec{u}^i}\right)_t < 0$ by Eq. (7.9). The increase of $\bar{J}$ corresponds to an increase of total trajectory-cost. Alternatively, if $(\vec{u}_t)^i = -1$, then a similar argument can be applied; any modification to $(\vec{u}_t)^i$ must be an increase, which again will lead to an increase in the total trajectory cost, since $\left(\frac{\partial \bar{J}}{\partial \vec{u}^i}\right)_t > 0$ by Eq. (7.9).

Therefore any change to any action component $(\vec{u}_t)^i$ at any time step t will lead to an increase in total trajectory cost, and therefore the current trajectory must have been at a local minimum of total trajectory cost with respect to all action components $(\vec{u}_t)^i$, at all time steps t; i.e. the trajectory was locally optimal. □

7.3 The Local Optimality of the Value-Gradient Learning Objective

In this section, it is proven that if the VGL objective is achieved, which for deterministic trajectories is $\widetilde{G}_t = G'_t$ for all t along a greedy trajectory, then that trajectory is locally extremal, and in certain situations, locally optimal. Only problems that are deterministic are considered, and which are either episodic or have $\gamma < 1$.

First a lemma is proven, and then the main result of this section and chapter comes in Theorem 7.1.

Lemma 7.6. *For a deterministic greedy trajectory and any fixed $\lambda \in [0, 1]$, if the VGL objective is met exactly in a deterministic environment, i.e. if $\widetilde{G}_t = G'_t$ for all t, then $\widetilde{G}_t = \left(\frac{\partial \bar{J}}{\partial \vec{x}}\right)_t$ for all t.*

Proof. First note that since $\widetilde{G}_t$ is defined to exist, then G'_t must also exist (since $\widetilde{G}_t = G'_t$ for all t).

The target value gradient was defined for non-terminal states by Eq. (3.7) to be,

$$G'_t := \left(\frac{D\bar{U}}{D\vec{x}}\right)_t + \gamma \left(\frac{D\bar{f}}{D\vec{x}}\right)_t \left(\lambda G'_{t+1} + (1-\lambda)\widetilde{G}_{t+1}\right),$$

where the $\frac{D}{D\vec{x}}$ notation was defined by Eq. (3.5).

Substituting $G'_t = \widetilde{G}_t$ into the above definition for G'_t gives,

$$\begin{aligned}
\widetilde{G}_t &= \left(\frac{D\bar{U}}{D\vec{x}}\right)_t + \gamma \left(\frac{D\bar{f}}{D\vec{x}}\right)_t \widetilde{G}_{t+1} \\
&= \left(\frac{\partial\bar{U}}{\partial\vec{x}}\right)_t + \gamma \left(\frac{\partial\bar{f}}{\partial\vec{x}}\right)_t \widetilde{G}_{t+1} + \left(\frac{\partial A}{\partial\vec{x}}\right)_t \left(\frac{\partial\widetilde{Q}}{\partial\vec{u}}\right)_t && \text{(by Eqs. (3.5) \& (7.5))} \\
&= \left(\frac{\partial\bar{U}}{\partial\vec{x}}\right)_t + \gamma \left(\frac{\partial\bar{f}}{\partial\vec{x}}\right)_t \widetilde{G}_{t+1} && \text{(by Lemma 7.4)} \\
&= \left(\frac{\partial U}{\partial\vec{x}}\right)_t + \gamma \left(\frac{\partial f}{\partial\vec{x}}\right)_t \widetilde{G}_{t+1} && \text{(by Eq. (7.1))} \qquad (7.10)
\end{aligned}$$

where in the application of Lemma 7.4, it was known that $\left(\frac{\partial A}{\partial\vec{x}}\right)_t$ exists since G'_t exists.

Also, differentiating Eq. (7.7), for non-terminal states, with respect to $\vec{x}$ gives

$$\left(\frac{\partial\bar{J}}{\partial\vec{x}}\right)_t = \left(\frac{\partial U}{\partial\vec{x}}\right)_t + \gamma \left(\frac{\partial f}{\partial\vec{x}}\right)_t \left(\frac{\partial\bar{J}}{\partial\vec{x}}\right)_{t+1}. \qquad (7.11)$$

So by comparing Eqs. (7.10) and (7.11), we see that $\left(\frac{\partial\bar{J}}{\partial\vec{x}}\right)_t$ and $\widetilde{G}_t$ have the same recursive definition for non-terminal states. For terminal states, $\vec{x}_T \in \mathbb{T}$, we have $\left(\frac{\partial\bar{J}}{\partial\vec{x}}\right)_T = \widetilde{G}_T = \left(\frac{\partial\Phi}{\partial\vec{x}}\right)_T$, by Eqs. (3.7) and (7.7). Therefore, for episodic trajectories, $\widetilde{G}_t = \left(\frac{\partial\bar{J}}{\partial\vec{x}}\right)_t$ for all t, since they have the same recursive formula at both terminal and non-terminal states.

Alternatively, if the problem is non-episodic then the recursive formula for $\widetilde{G}_t$ can be expanded as follows:

$$\begin{aligned}
\widetilde{G}_t &= \left(\frac{\partial\bar{U}}{\partial\vec{x}}\right)_t + \gamma \left(\frac{\partial\bar{f}}{\partial\vec{x}}\right)_t \widetilde{G}_{t+1} && \text{(by Eq. (7.10))} \\
&= \left(\frac{\partial U}{\partial\vec{x}}\right)_t + \gamma \left(\frac{\partial f}{\partial\vec{x}}\right)_t \widetilde{G}_{t+1} && \text{(by Eq. (7.1))} \\
&= \left(\frac{\partial U}{\partial\vec{x}}\right)_t + \gamma \left(\frac{\partial f}{\partial\vec{x}}\right)_t \left(\frac{\partial U}{\partial\vec{x}}\right)_{t+1} + \gamma^2 \left(\frac{\partial f}{\partial\vec{x}}\right)_t \left(\frac{\partial f}{\partial\vec{x}}\right)_{t+1} \left(\frac{\partial U}{\partial\vec{x}}\right)_{t+2} \\
&\quad + \gamma^3 \left(\frac{\partial f}{\partial\vec{x}}\right)_t \left(\frac{\partial f}{\partial\vec{x}}\right)_{t+1} \left(\frac{\partial f}{\partial\vec{x}}\right)_{t+2} \left(\frac{\partial U}{\partial\vec{x}}\right)_{t+3} + \dots && \text{(expanding recursion)}
\end{aligned}$$

This infinite series should converge because $\gamma < 1$. The identical recursion for $\left(\frac{\partial \bar{J}}{\partial \vec{x}}\right)_t$, given by Eq. (7.11), can be expanded into the same infinite series. Therefore $\left(\frac{\partial \bar{J}}{\partial \vec{x}}\right)_t = \widetilde{G}_t$ for all t.

This proves that in either the case of episodic or non-episodic trajectories, $\widetilde{G}_t = \left(\frac{\partial \bar{J}}{\partial \vec{x}}\right)_t$ for all t. □

The main theorem of this chapter is now given:

Theorem 7.1. *Any deterministic greedy trajectory satisfying the deterministic VGL objective exactly, i.e. satisfying $\widetilde{G}_t = {G'}_t$ for all t, must be locally extremal.*

Proof. Since a greedy trajectory minimises $\widetilde{Q}(\vec{x}_t, \vec{u}_t, \vec{w})$ with respect to $\vec{u}_t$ at each time-step t, we know at each t and for each action component i,

$$\begin{cases} \left(\frac{\partial \widetilde{Q}}{\partial \vec{u}^i}\right)_t = 0 & \text{if } (\vec{u}_t)^i \text{ is not saturated} \\ \left(\frac{\partial \widetilde{Q}}{\partial \vec{u}^i}\right)_t < 0 & \text{if } (\vec{u}_t)^i \text{ is saturated and } (\vec{u}_t)^i = 1 \\ \left(\frac{\partial \widetilde{Q}}{\partial \vec{u}^i}\right)_t > 0 & \text{if } (\vec{u}_t)^i \text{ is saturated and } (\vec{u}_t)^i = -1. \end{cases} \tag{7.12}$$

These follow from Lemmas 7.1 and 7.3. Therefore since,

$$\begin{aligned} \left(\frac{\partial \bar{J}}{\partial \vec{u}}\right)_t &= \left(\frac{\partial U}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial f}{\partial \vec{u}}\right)_t \left(\frac{\partial \bar{J}}{\partial \vec{x}}\right)_{t+1} && \text{(by differentiating Eq. (7.7))} \\ &= \left(\frac{\partial U}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial f}{\partial \vec{u}}\right)_t \widetilde{G}_{t+1} && \text{(by Lemma 7.6)} \\ &= \left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \widetilde{G}_{t+1} && \text{(by Eq. (7.1))} \\ &= \left(\frac{\partial \widetilde{Q}}{\partial \vec{u}}\right)_t && \text{(by Eq. (7.5))} \end{aligned}$$

we have $\left(\frac{\partial \bar{J}}{\partial \vec{u}}\right)_t = \left(\frac{\partial \widetilde{Q}}{\partial \vec{u}}\right)_t$ for all t. Therefore the consequences of a greedy trajectory (Eq. (7.12)) become equivalent to the sufficient conditions for a LET (Eq. (7.8)), which implies the trajectory is a LET. □

7.3.1 Bang-Bang Controls

When the greedy-trajectory is generated entirely from saturated actions, then some stronger optimality results apply, as demonstrated by the following corollary:

Corollary 7.1. *If, in addition to the conditions of Theorem 7.1, all of the actions are saturated (bang-bang control), then the trajectory is locally optimal.*

Proof. This follows from Lemma 7.5. □

According to the Bang-Bang Principle (Sonneborn and Vleck, 1965), bang-bang control often arises in situations where the model functions are linear with respect to bounded action vectors, or when the problem being solved is a time minimisation problem. Hence it is often the case that all LETs found by this method are locally optimal.

7.3.2 Discussion of Theorem 7.1

The optimality result proven in Theorem 7.1 is valid for any $\lambda \in [0, 1]$. It only needs satisfying over a single trajectory, whereas for VL the corresponding optimality condition (i.e. Bellman's) needs satisfying over the whole of state space. This implies that VGL methods have a much lesser requirement for value exploration than VL methods do, and can lead to greater efficiency.

A separate efficiency issue is the algorithmic complexity of VL and VGL, and these are both the same ($O(\dim(\vec{w}))$ per time step), provided the batch-mode version of VGL(λ) is used (Alg. 3.2). If the on-line implementation of VGL(λ) is used then the algorithmic complexity of VGL(λ) is higher than that for VL methods, as described in Section 3.5.1.

7.3.3 The Relationship to Pontryagin's Minimum Principle

The proof of Theorem 7.1 is highly related to Pontryagin's minimum principle (PMP; Section 4.3), since Eq. (7.10) is identical to the PMP equation for a costate vector defined by Eq. (4.2b). Therefore $\widetilde{G}_t$ is acting as a costate vector of PMP. Similarly, the greedy policy Eq. (7.4) forms the minimum condition of PMP (Eq. (4.2c)). And the state transition function $\vec{x}_{t+1} = f(\vec{x}_t, \vec{u}_t)$ is common to both PMP and ADPRL (in Eqs. (1.1) and (4.2a)). This completes Pontryagin's conditions to be a LET.

However PMP still needs supplementing with Lemma 7.6 for it to be applicable for any $\lambda \in [0, 1]$. Hence the proof of this chapter does add something beyond PMP. Also a major bonus of the proof of this chapter is the corollary providing the extra conclusion for bang-bang control producing locally optimal trajectories (in Corollary 7.1). This corollary came as a consequence of carefully considering saturated actions, and what it means to define a locally optimal or extremal trajectory.

7.4 Chapter Conclusions

The result that learning the value gradients exactly in a deterministic environment, i.e. achieving $\widetilde{G}_t = G'_t$ for all t, under a greedy policy, ensures a locally extremal, and often locally optimal, trajectory has been proven. This result provides a key motivation for value-gradient methods, i.e. it is what provides the automatic local value exploration and high speed of learning for VGL methods. This theorem is what sets VGL methods apart from VL methods, which do not have an equivalent trajectory optimality result; and counterexamples to the VL equivalent of perfectly learning values along a trajectory have been given in this thesis (for example, in Fig. 3.1).

The result proven in this chapter shows that provided the VGL algorithms make progress in moving the gradients $\widetilde{G}_t$ towards their targets G'_t, the trajectories will make progress in automatically bending into locally optimal shapes.

The result of Theorem 7.1 is also relevant to VL methods, because it shows what they need to learn to achieve optimality in continued-valued state space. Even if VL methods go about learning the value-gradients by different means (i.e. stochastic value exploration), the value-gradients will still need learning, eventually.

The chapter has also shown the relationship to Pontryagin's minimum principle, and how this theorem extends PMP to apply to VGL(λ) for any $\lambda \in [0, 1]$. Also the corollary showing that saturated actions are sufficient to imply local optimality, as opposed to local extremity, has been given, which seems to be an original extension of PMP.

In further work it would be good to be able to extend the optimality proof of this problem to stochastic environments.

Chapter 8

Convergence Proof of VGLΩ(1) and its Equivalence to BPTT

In this chapter it is shown that the VGLΩ(1) weight update, i.e. VGL(1) with an Ω_t matrix given by Eq. (3.8), when combined with the greedy-on-$\widetilde{Q}$ policy, is identical to the application of backpropagation through time (BPTT) to a greedy policy. This means that VGLΩ(1) is identical to gradient descent on the total trajectory cost with respect to the weight vector $\vec{w}$ of the critic function used by the greedy policy. It also makes a theoretical connection between two seemingly different learning paradigms of ADPRL, i.e. the paradigm of using a critic function, and the PGL/BPTT paradigm described in Chapter 6. This proof of equivalence acts as a convergence proof for VGLΩ(1), since it means that VGLΩ(1) will adopt the same convergence properties that BPTT has. A summary of the convergence proof of this chapter is shown in Fig. 8.1.

$$\boxed{\begin{array}{c}\text{Gradient descent on } \widehat{J} \text{ with respect to } \vec{w} \\ \text{for a greedy-on-}\widetilde{Q} \text{ policy function, } \pi(\vec{x},\vec{w}) \\ \text{on some critic-network, } \widetilde{J}(\vec{x},\vec{w})\end{array}} \equiv \boxed{\text{VGL}\Omega(1)\text{, with that greedy policy}}$$

$\therefore$ VGLΩ(1) with a greedy policy, will converge, under smoothness assumptions.

Figure 8.1: The Equivalence of VGLΩ(1) to BPTT, and the Implied Convergence Proof.

The convergence properties for BPTT were described previously in Section 6.1.2, and are limited to continuous-valued action and state spaces. Also the BPTT algorithm is best suited to the noise in the environment being additive, otherwise the BPTT algorithm may only find an approximation to the true gradient descent direction (for reasons previously described in Section 6.1.1). These limitations of the BPTT convergence assurance will therefore carry over to the VGLΩ(1) algorithm.

The equivalence proof of this chapter is valid for both stochastic and deterministic environments. The convergence implication is valid with a general smooth function approximator (i.e. a neural network) for the critic, and with the greedy-on-$\widetilde{Q}$ policy defined in Chapter 5. The greedy-on-$\widetilde{Q}$ policy was defined in Chapter 5 as a model-based deterministic version of the greedy policy, that is often more practical to work with and is a sometimes-exact approximation to the true ADPRL greedy policy, as described in Chapter 5's introduction.

In the rest of this chapter, in Section 8.1, a short summary of key convergence proofs from the ADP literature is made, and how they differ from the proof of this chapter is described. Section 8.2 contains the main equivalence-proof of this chapter. Chapter conclusions are given in Section 8.3.

Empirical results demonstrating the advantages of using the algorithm VGLΩ(1) compared to VGL(λ) have already been given, in Chapter 5.

8.1 Convergence Proofs in the ADP Literature

Proving convergence of ADPRL algorithms is a challenging problem. Convergence proofs from the RL literature were described previously in Section 2.7. This section concentrates on similar results from the ADP literature.

Ferrari and Stengel (2004), Howard (1960b) and Al-Tamimi et al. (2008) show the ADP process will converge to optimal behaviour if the critic could be perfectly learned all over the state space at each iteration. However in reality it is necessary to work with a function approximator for the critic with finite capabilities, so this assumption is not valid.

A variant of DHP is proven to converge by Heydari and Balakrishnan (2011) for a critic which is linear in $\vec{w}$. This DHP-variant assumes the critic can update all the $\widetilde{G}_t$ values towards their targets, ${G'}_t$, for all t, in a single critic weight update. This large critic weight update is possible using a least-squares solution from linear algebra, involving a matrix inversion operation to achieve this goal in one weight update, so it is not appropriate for the general non-linear neural critics considered in this thesis. The process still works iteratively though, because every time the critic is updated, the target gradients ${G'}_t$ will change (since the targets are functions of $\widetilde{G}_t$; they are moving targets). The significant achievement of Heydari and Balakrishnan (2011) is to prove that this process converges, while operating under the limitations of linear function approximation, and under a greedy policy (which is a major challenge for ADPRL critic-convergence algorithms). It is a convergence proof for a $\lambda = 0$ algorithm, whereas the proof in this chapter is for $\lambda = 1$ and for a non-linear critic, so their method complements the proof in this chapter nicely.

Another variant of DHP is proven to converge by Prokhorov and Wunsch (1997b). This proof is for the Galerkin-based form of DHP (Werbos, 1998), which is a "residual gradients" version of DHP (Baird, 1995), i.e. true gradient descent on an error function of the form $E = \sum_t ({G'}_t - \widetilde{G}_t)^2$. Unfortunately, as it is well known, Galerkin-based methods almost always converge to suboptimal solutions in stochastic environments (as described in Section 2.7.3). Furthermore, and this fact is less well known than the previous one, Galerkin versions of DHP and VGL will often converge to suboptimal solutions in deterministic environments too, when a greedy policy is used (Fairbank, 2008, sec 2.3). Therefore Galerkin-based methods are not considered in this thesis.

One reason that it is difficult in general to make convergence proofs for ADPRL methods is that in the Bellman condition, there is an interdependence between $J(\vec{x}, \vec{\mathbf{e}}, \vec{z})$, $A(\vec{x}, \vec{e}, \vec{z})$ and $\widetilde{J}(\vec{x}, \vec{w})$. This chapter makes an insight into this difficulty by showing (in Lemma 8.4) that the dependency of a greedy policy on the critic is primarily through the value-gradient.

8.2 The Relationship of VGL to BPTT

This section contains the main proof that the VGL$\Omega(1)$ weight update, when combined with a greedy-on-$\widetilde{Q}$ policy, is equivalent to backpropagation through time (BPTT) on that greedy policy. First a reminder of the equations for BPTT are given (in Section 8.2.1), then some important lemmas about a greedy policy are derived (in Section 8.2.2). Then it is demonstrated in Section 8.2.3 that when BPTT is applied to a greedy-on-$\widetilde{Q}$

policy, the weight update obtained is identical to VGLΩ(1). Finally, in Section 8.2.4, the consequences of the results and the convergence properties are discussed.

8.2.1 Backpropagation Through Time for Control Problems

As described in Chapter 6, BPTT can be used to perform gradient descent on the sampled cost-to-go function $\widehat{J}(\vec{x}, \vec{\mathbf{e}}, \vec{z})$, and hence optimise the ADPRL problem, as was summarized in Fig. 6.1. Gradient descent of this kind will naturally find local minima of $\widehat{J}(\vec{x}_0, \vec{\mathbf{e}}_0, \vec{z})$, and has good convergence properties when the surface $\widehat{J}(\vec{x}_0, \vec{\mathbf{e}}_0, \vec{z})$ is smooth with respect to $\vec{z}$.

The BPTT derivation was given in Section 6.1. The main BPTT weight update was given by Eq. (6.3), which was:

$$\Delta\vec{z} = -\,\beta \sum_{t\geq 0} \gamma^t \left(\frac{\partial A}{\partial \vec{z}}\right)_t \left(\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \left(\frac{\partial \widehat{J}}{\partial \vec{x}}\right)_{t+1}\right). \tag{8.1}$$

Full pseudocode for the BPTT algorithm was given in Alg. 6.1.

In the BPTT derivation of Section 6.1, it is useful to note that, by comparing Eq. (6.4), i.e.

$$\left(\frac{\partial \widehat{J}}{\partial \vec{x}}\right)_t = \begin{cases} \left(\frac{D\bar{U}}{D\vec{x}}\right)_t + \gamma \left(\frac{D\bar{f}}{D\vec{x}}\right)_t \left(\frac{\partial J}{\partial \vec{x}}\right)_{t+1} & \text{if } \vec{x}_t \notin \mathbb{T} \\ \left(\frac{\partial \Phi}{\partial \vec{x}}\right)_t & \text{if } \vec{x}_t \in \mathbb{T}, \end{cases} \tag{8.2}$$

with Eq. (3.7), i.e.,

$$G'_t := \begin{cases} \left(\frac{D\bar{U}}{D\vec{x}}\right)_t + \gamma \left(\frac{D\bar{f}}{D\vec{x}}\right)_t \left(\lambda G'_{t+1} + (1-\lambda)\widetilde{G}_{t+1}\right), & \text{for } \vec{x}_t \notin \mathbb{T} \\ \left(\frac{\partial \Phi}{\partial \vec{x}}\right)_t, & \text{for } \vec{x}_t \in \mathbb{T} \end{cases}$$

it can be seen that

$$G'_t \equiv \left(\frac{\partial \widehat{J}}{\partial \vec{x}}\right)_t, \text{ for all } t, \qquad \text{when } \lambda = 1. \tag{8.3}$$

Note that Eq. (8.2) uses an exact equality as opposed to the approximation seen in Eq. (6.4), because in this chapter we are considering the BPTT algorithm exactly as implemented by Alg. 6.1. However as noted previously in Section 6.1.1, if the conditions for $\frac{\partial \bar{f}}{\partial \vec{x}} \equiv \frac{\partial f}{\partial \vec{x}}$ (which were stated in Section 3.2.1) do not hold, then BPTT will only generate approximations to the quantity $\frac{\partial \widehat{J}}{\partial \vec{z}}$.

8.2.2 Lemmas about the Greedy-on-$\widetilde{Q}$ Policy and Greedy Actions

To prepare for the later analysis of BPTT applied to the greedy-on-$\widetilde{Q}$ policy, first we prove some lemmas about this greedy policy. These lemmas apply only when the action space, $\mathbb{A}$, is equal to $\mathbb{R}^{\dim(\vec{u})}$, which we will denote as $\mathbb{A}^*$.

A greedy-on-$\widetilde{Q}$ policy $\pi(\vec{x}, \vec{w})$ is a policy which always selects actions $\vec{u}$ that are the minimum of the smooth function $\widetilde{Q}(\vec{x}, \vec{u}, \vec{w})$ defined by Eq. (3.9):

$$\widetilde{Q}(\vec{x}, \vec{u}, \vec{w}) := \bar{U}(\vec{x}, \vec{u}) + \gamma \widetilde{J}(\bar{f}(\vec{x}, \vec{u}), \vec{w}). \tag{8.4}$$

These minimising actions are what we call *greedy actions*. In this case, since the minimum of a smooth function is found from an unbound domain, $\mathbb{A}^*$, the following two consequences hold:

Lemma 8.1. *For a greedy action $\vec{u}$ chosen from $\mathbb{A}^*$, we have $\frac{\partial \widetilde{Q}}{\partial \vec{u}} = \vec{0}$.*

Lemma 8.2. *For a greedy action $\vec{u}$ chosen from $\mathbb{A}^*$, $\frac{\partial^2 \widetilde{Q}}{\partial \vec{u} \partial \vec{u}}$ is a positive semi-definite matrix.*

Note that the above two lemmas are multi-dimensional analogues of the familiar minimum conditions for a one-dimensional function $q(u) : \mathbb{R} \to \mathbb{R}$ with an unbound domain, which are $q'(u) = 0$ and $q''(u) \geq 0$, respectively.

We now prove too less obvious lemmas about the greedy-on-$\widetilde{Q}$ policy:

Lemma 8.3. *The greedy-on-$\widetilde{Q}$ policy on $\mathbb{A}^*$ implies $\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t = -\gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \widetilde{G}_{t+1}$.*

Proof. First, note that differentiating the definition of $\widetilde{Q}(\vec{x}, \vec{u}, \vec{w})$ given in Eq. (8.4) gives,

$$\left(\frac{\partial \widetilde{Q}}{\partial \vec{u}}\right)_t = \left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \widetilde{G}_{t+1}. \tag{8.5}$$

Substituting this into Lemma 8.1 and solving for $\frac{\partial \bar{U}}{\partial \vec{u}}$ completes the proof. □

Lemma 8.4. *When* $\left(\frac{\partial \pi}{\partial \vec{w}}\right)_t$ *and* $\left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{u} \partial \vec{u}}\right)_t^{-1}$ *exist, the greedy-on-*$\widetilde{Q}$ *policy on* $\mathbb{A}^*$ *implies*

$$\left(\frac{\partial \pi}{\partial \vec{w}}\right)_t = -\gamma \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_{t+1} \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t^T \left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{u} \partial \vec{u}}\right)_t^{-1}.$$

Proof. We use implicit differentiation. The dependency of $\vec{u}_t = \pi(\vec{x}_t, \vec{w})$ on $\vec{w}$ must be such that Lemma 8.1 is always satisfied, since the policy is greedy on $\widetilde{Q}$. This means that $\left(\frac{\partial \widetilde{Q}}{\partial \vec{u}}\right)_t \equiv \vec{0}$, both before and after any infinitesimal change to $\vec{w}$. Therefore the greedy-policy function $\pi(\vec{x}_t, \vec{w})$ must be such that,

$$\begin{aligned}
\vec{0} =& \frac{\partial}{\partial \vec{w}} \left(\frac{\partial \widetilde{Q}(\vec{x}_t, \pi(\vec{x}_t, \vec{w}), \vec{w})}{\partial \vec{u}_t} \right) \\
=& \frac{\partial}{\partial \vec{w}} \left(\frac{\partial \widetilde{Q}(\vec{x}_t, \vec{u}_t, \vec{w})}{\partial \vec{u}_t} \right) + \left(\frac{\partial \pi}{\partial \vec{w}}\right)_t \frac{\partial}{\partial \vec{u}_t} \left(\frac{\partial \widetilde{Q}(\vec{x}_t, \vec{u}_t, \vec{w})}{\partial \vec{u}_t} \right) \\
=& \frac{\partial}{\partial \vec{w}} \left(\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \widetilde{G}_{t+1} \right) + \left(\frac{\partial \pi}{\partial \vec{w}}\right)_t \left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{u} \partial \vec{u}}\right)_t \\
=& \frac{\partial}{\partial \vec{w}} \left(\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \sum_i \left(\frac{\partial (\bar{f})^i}{\partial \vec{u}}\right)_t (\widetilde{G}_{t+1})^i \right) + \left(\frac{\partial \pi}{\partial \vec{w}}\right)_t \left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{u} \partial \vec{u}}\right)_t \\
=& \gamma \sum_i \left(\frac{\partial (\bar{f})^i}{\partial \vec{u}}\right)_t \frac{\partial (\widetilde{G}_{t+1})^i}{\partial \vec{w}} + \left(\frac{\partial \pi}{\partial \vec{w}}\right)_t \left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{u} \partial \vec{u}}\right)_t \\
=& \gamma \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_{t+1} \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t^T + \left(\frac{\partial \pi}{\partial \vec{w}}\right)_t \left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{u} \partial \vec{u}}\right)_t.
\end{aligned}$$

In the above six lines of algebra, the sum of the two partial derivatives in line 2 follows by the chain rule from the total derivative in line 1, since $\vec{w}$ appears twice in line 1. This step has also made use of $\vec{u}_t = \pi(\vec{x}_t, \vec{w})$. Also note that the first term in line 2 is not zero, despite the greedy-on-$\widetilde{Q}$ policy's requirement for $\frac{\partial \widetilde{Q}}{\partial \vec{u}} \equiv 0$, since in this term the $\vec{u}$ and $\vec{w}$ are now treated as independent variables. Then in the remaining lines, line 3 is by Eq. (8.5); line 4 just expands an inner product; line 5 follows since $\frac{\partial \bar{U}}{\partial \vec{u}}$ and $\frac{\partial \bar{f}}{\partial \vec{u}}$ are not functions of $\vec{w}$; and line 6 just forms an inner product.

Then solving the final line for $\left(\frac{\partial \pi}{\partial \vec{w}}\right)_t$ proves the lemma. □

8.2.3 The Equivalence of VGLΩ(1) to BPTT

In Section 8.2.1, the equation for gradient descent on $\widehat{J}(\vec{x}, \vec{\mathbf{e}}, \vec{z})$ was described for a general action network, $A(\vec{x}, \vec{e}, \vec{z})$, using BPTT. But BPTT can be applied to *any* differentiable policy function, and so we now consider what would happen if BPTT is applied to the *greedy-on-$\widetilde{Q}$ policy* $\pi(\vec{x},\vec{w})$, with actions chosen from $\mathbb{A}^*$. The parameter vector for the greedy policy is $\vec{w}$. Hence we can do gradient descent with respect to $\vec{w}$ instead of $\vec{z}$ (assuming the derivatives $\frac{\partial \pi}{\partial \vec{w}}$ and $\frac{\partial \widehat{J}}{\partial \vec{w}}$ exist). It is first worth emphasising that with the greedy policy, it is the same weight vector that appears in the critic, $\vec{w}$, as appears in the greedy policy $\pi(\vec{x},\vec{w})$.

Consequently, for the gradient-descent equation in BPTT for control (Eq. (8.1)), we now change all instances of A and $\vec{z}$ to π and $\vec{w}$, respectively, giving the new weight update:

$$\begin{aligned}
\Delta\vec{w} &= -\alpha\sum_{t\geq 0}\gamma^t\left(\frac{\partial \pi}{\partial \vec{w}}\right)_t\left(\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t+\gamma\left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t\left(\frac{\partial \widehat{J}}{\partial \vec{x}}\right)_{t+1}\right) && \text{(By Eq. (8.1))}\\
&= -\alpha\sum_{t\geq 0}\gamma^{t+1}\left(\frac{\partial \pi}{\partial \vec{w}}\right)_t\left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t\left(\left(\frac{\partial \widehat{J}}{\partial \vec{x}}\right)_{t+1}-\widetilde{G}_{t+1}\right) && \text{(By Lemma 8.3)}\\
&= -\alpha\sum_{t\geq 0}\gamma^{t+1}\left(\frac{\partial \pi}{\partial \vec{w}}\right)_t\left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t\left({G'}_{t+1}-\widetilde{G}_{t+1}\right) && \text{(by Eq. (8.3), with } \lambda=1)\\
&= \alpha\sum_{t\geq 0}\gamma^{t+2}\left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_{t+1}\left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t^T\left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{u}\partial \vec{u}}\right)_t^{-1}\left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t({G'}_{t+1}-\widetilde{G}_{t+1}) && \text{(By Lemma 8.4)}\\
&= \alpha\sum_{t\geq 0}\gamma^{t+1}\left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t\Omega_t({G'}_t-\widetilde{G}_t),
\end{aligned} \tag{8.6}$$

where Ω_t is the special form given previously by Eq. (3.8), i.e.,

$$\Omega_t := \begin{cases}\left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_{t-1}^T\left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{u}\partial \vec{u}}\right)_{t-1}^{-1}\left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_{t-1} & \text{for } t>0\\ 0 & \text{for } t=0,\end{cases}$$

and is positive semi-definite, by the greedy policy (Lemma 8.2).

Equation (8.6) is identical to the VGLΩ(1) weight update equation (Eq. (3.6) with Eq. (3.8) and $\lambda=1$), with $\gamma=1$, provided $\left(\frac{\partial \pi}{\partial \vec{w}}\right)_t$ and $\left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{u}\partial \vec{u}}\right)_t^{-1}$ exist for all t. If $\left(\frac{\partial \pi}{\partial \vec{w}}\right)_t$ does not exist, then $\frac{\partial \widehat{J}}{\partial \vec{w}}$ is not defined either.

This completes the demonstration of the equivalence of a critic-learning algorithm (VGLΩ(1)) to BPTT (with a greedy-on-$\widetilde{Q}$ policy with actions chosen from $\mathbb{A}^*$, and when $\frac{\partial \widehat{J}}{\partial \vec{w}}$ exists).

Furthermore, the presence of the γ^t factor in Eq. (8.6) could be removed if we changed the BPTT gradient-descent equation by removing the γ^t factor from Eq. (6.3). This would make the BPTT weight update more accurately follow the spirit and intention of the on-line critic weight update; and then the equivalence of VGLΩ(1) to BPTT would hold for any γ too.

For a fuller discussion of the Ω_t matrix defined by Eq. (3.8), including methods for its computation and a discussion of its purpose and effectiveness, see reference (Fairbank et al., 2012b, Sections 7.3.4 and 7.4.3).

8.2.4 Discussion

If the ADP problem is such that $\frac{\partial \pi}{\partial \vec{w}}$ always exists for a greedy policy (specifically, the greedy-on-$\widetilde{Q}$ policy), then the relatively-good convergence guarantees of BPTT will apply to VGLΩ(1). In this case, this particular VGL algorithm will achieve monotonic progress with respect to $\widehat{J}$, and so will have guaranteed convergence, provided it is operating within a smooth region of the surface of the function $\widehat{J}$, and the step size is sufficiently small. Also the conditions for assured convergence of BPTT, which were discussed in Section 6.1.2, must be satisfied.

A key possible source of lack of smoothness could come from the min operator in the greedy policy, as this could cause $\frac{\partial \pi}{\partial \vec{w}}$ to not exist. But significantly, when a sigmoidal greedy-policy of the type defined in Chapter 5 is used, the greedy-policy function is always smooth and differentiable.

The equivalence result of this chapter was surprising to discover, because it was thought that the VGL weight updates (Eq. (3.6), and DHP and GDHP) were approximations to gradient descent on an error function $E = \sum_t (G'_t - \widetilde{G}_t)^T \Omega_t (G'_t - \widetilde{G}_t)$. But the VGL($\lambda$) weight update is not true gradient descent on E (unless both the policy is fixed and $\lambda = 1$). The equivalence-proof of this chapter shows that when a greedy policy is used, VGL(1) is an approximation to true gradient descent on J rather than on E; and this approximation becomes exact if VGLΩ(1) is used.

It was also surprising to learn that BPTT and critic weight updates, i.e. the two opposite paradigms of ADPRL, are not as fundamentally different to each other as first thought. In fact, this equivalence proof makes one question whether there really is a big advantage of using a value function at all, in contradiction to the importance placed on the value-function in the introduction of this thesis. A fairly speculative discussion of this issue follows:

- If a value-function method is going to be used, and if a VGL method is chosen (which would be done for example if the arguments given in Sec. 3.1 are considered convincing), then it does seem like a good idea to use VGLΩ(1), because this variant has proven convergence with a greedy policy and non-linear function approximation. Divergence examples exist for other VGL variants, such as those described in Chapter 9.

- If someone is using the VGLΩ(1) algorithm with a greedy policy, then they may as well use the equivalent algorithm, BPTT. However to implement BPTT with a greedy policy, it is necessary to use the expression for $\frac{\partial \pi}{\partial \vec{w}}$ given by Lemma 8.4. Therefore the implementation of BPTT will be identical to VGLΩ(1). Before this equivalence proof was made, it was not realised that the two methods were the same.

- Arguments can be given to motivate the use of value-function methods because they learn something useful (i.e. the values) about every intermediate state that the trajectory passes through, whereas PGL methods and BPTT do not as obviously have this benefit. This equivalence proof shows that BPTT with a greedy policy is just as good in this respect.

- Value-function methods still introduce the possibilities of using $\lambda < 1$ and model-free learning (through VL methods), both of which have benefits (as discussed in Sections 2.3.1 and 1.4, respectively). Although the benefits of noise-tolerance by $\lambda < 1$ may not be as important for VGL methods as they are for VL methods (as discussed in Section 3.5.3).

- A value-function allows more computationally efficient on-line learning than is done by critic-free methods such as the RTRL algorithm.

- Even despite the equivalence of BPTT and VGLΩ(1) proven in this chapter, it might be beneficial to use a greedy policy as opposed to an action network, because a greedy policy might just happen to be more stable for training by gradient descent. For example, using a greedy policy may go some way towards solving the problem of "exploding gradients" (Fairbank et al., 2014a) compared to an action network; but this is a speculative and separate issue to explore.

8.3 Chapter Conclusions

A strong theoretical equivalence has been given between BPTT and VGLΩ(1), which are two algorithms that on first sight appeared to be operating totally differently. This provides a convergence proof for this VGL algorithm under the conditions stated in section 8.2.4. This analysis has been successful for a VGL learning system where a greedy policy is used (specifically the greedy-on-$\widetilde{Q}$ policy), and using non-linear function approximation.

The convergence proof is only valid for $\lambda = 1$, but it is for the situation of a concurrently changing greedy-policy function, which usually makes convergence much harder to assure. For example, in the next chapter, various other algorithms will all be made to diverge with $\lambda = 1$.

In the experiments in Chapter 5, the effectiveness of using the Ω_t matrix was shown at stabilising learning and producing approximate monotonic learning progress for a neural-network based critic with a greedy policy, even when combined with an aggressive learning accelerator such as RPROP.

Chapter 9

Divergence Examples

This chapter investigates which of the major ADPRL algorithms considered in this thesis can be made to diverge with a greedy policy. Divergence results are derived for VGL(1), VGL(0), VGLΩ(0), TD(0) and TD(1). Consequently the divergence results also apply to DHP and HDP, which are instances of VGL(0) and TD(0), respectively. All divergences are derived for a smooth and differentiable critic function, and smooth and differentiable model and cost functions.

The purpose of this chapter is to prove that although TD(1) and VGL(1) are true gradient descent when the policy function is fixed, and therefore will converge, they are not true gradient descent, and convergence is not assured, when a changing policy function is used. The "changing policy function" considered in this chapter is the greedy policy, but any form of value-iteration with an actor-critic architecture could also be used. It was previously an open question of whether these $\lambda = 1$ algorithms can ever be made to diverge, and this chapter shows the answer to be that they can.

The results may be surprising, because TD(1) is considered to be one of the most reliably convergent ADPRL algorithms. TD(λ) is proven to converge when $\lambda = 1$ since it is then (and only then) true gradient descent on an error function (Sutton, 1988). Also for $0 \leq \lambda \leq 1$, TD(λ) is proven to converge by Tsitsiklis and Van Roy (1996a) when the approximate value function is linear in its weight vector and learning is on-policy. Recent advancements in the RL literature have extended convergence conditions of variants of TD(λ) to an off-policy setting (Sutton et al., 2009), and with non-linear function approximation of the value function (Maei et al., 2009). However, all these proofs apply when the agent is following a fixed policy instead of the greedy-policy situation we consider here.

Ferrari and Stengel (2004) show that ADP processes will converge to optimal behaviour if the value function could be perfectly learned all over the state space at each iteration. However in reality, the critic will be implemented by a function approximator with limited flexibility, so this assumption is not valid. Working with a general quadratic function approximator, (Werbos, 1998, sec.7.7-7.8) proves the general instability of DHP and GDHP. This analysis was for a fixed policy, so with a greedy policy convergence would presumably seem even less likely. This chapter confirms this.

A key insight into the difficulty of understanding convergence with a greedy policy is shown in Lemma 8.4, i.e. the expression for $\frac{\partial \pi}{\partial \vec{w}}$ derived in the previous chapter, which proves that the dependency of a greedy action on the approximated value function is primarily through the value-gradient. Hence a value-gradient analysis is used in this chapter to understand the divergence of all of the algorithms being tested, including the value-learning ones.

These divergence examples contrast to VGLΩ(1), which was proven to converge with a greedy policy, under certain smoothness conditions, in the previous chapter.

The approach of this chapter to achieve divergence is to define a problem which is simple enough to analyse algebraically, but flexible enough to provide divergence. It takes several sections to obtain an analytical expression for the VGL(λ) weight update purely in terms of the critic weights, i.e. to find a weight-update expression of the form $\Delta\vec{w} = h(\vec{w})$, for some function h. This can then be used to find which learning parameters could make this function h divergent. To achieve both of these things, it is necessary to perform the following steps:

1. Define an environment (i.e. model and cost functions, $f(\vec{x}, \vec{u})$ and $U(\vec{x}, \vec{u})$) which is suitably flexible to create a divergence example (Section 9.1.1).

2. Define a critic function, $\widetilde{J}(\vec{x}, \vec{w})$ which is suitably flexible to allow divergence in the defined environment (Section 9.1.2).

3. Find an analytical expression for the greedy policy, and expressions for the greedy-trajectory's states and actions (Section 9.1.3), purely in terms of $\vec{w}$.

4. Find analytical expressions for $\widetilde{G}_t$ and ${G'}_t$ (Sections 9.1.4 to 9.1.5).

5. Find the final analytical expression for the VGL(λ) weight update (Section 9.1.6).

6. Try to find a set of learning parameters that cause divergence for the VGL(λ) weight update (Section 9.2).

Going through this long chain of derivations allows the analytical expression for the whole weight update, and hence the divergence examples, to be found. This analysis was done for the VGL(λ) weight update because it is easier to analyse than the TD(λ) one, since as mentioned above the greedy policy depends on the value-gradient. However the divergences found for VGL(λ) also cause divergence for TD(λ), as demonstrated empirically in Section 9.3.

The divergence results of this chapter apply specifically to a greedy policy. But since a greedy policy can be understood as a form of value-iteration, it is also possible to use an actor-critic architecture with a value-iteration weight-update scheme to achieve the same results. A specific action-network that can also achieve these divergences under value iteration is described by Fairbank and Alonso (2012b). That reference also extends the divergence results of this chapter to include divergence for the Sarsa(λ) algorithm (Rummery and Niranjan, 1994).

Finally, in Section 9.4, chapter conclusions are given.

9.1 Example Analytical Problem

In this section, an ADPRL problem is defined which is simple enough to analyse algebraically. This problem is defined, and the VGL(λ) weight update is derived algebraically for it, in Sections 9.1.1 to 9.1.6. These results will then be used in the next main section (Section 9.2) to derive the main divergence results of this chapter.

9.1.1 Environment Definition

A deterministic environment is defined with state $x \in \mathbb{R}$ and action $u \in \mathbb{R}$, and with model and cost functions:

$$f(x_t, t, u_t) := \bar{f}(x_t, t, u_t) := x_t + u_t \qquad \text{for } t \in \{0, 1\} \tag{9.1a}$$

$$U(x_t, t, u_t) := \bar{U}(x_t, t, u_t) := k(u_t)^2 \qquad \text{for } t \in \{0, 1\} \tag{9.1b}$$

where $k > 0$ is a constant. Each trajectory is defined to terminate immediately on arriving at time step $t = 2$, when a final terminal cost of

$$\Phi(x_t) := \bar{\Phi}(x_t) := (x_t)^2 \tag{9.2}$$

is given, so that exactly three costs are received by the agent over the full trajectory duration. The termination condition is dependent on t, so strictly speaking t should be included in the state vector, but instead t is included as an extra argument to the functions $U(x_t, t, u_t)$, $f(x_t, t, u_t)$ and $\widetilde{J}(x_t, t, \vec{w})$.

A whole trajectory is completely parameterised by x_0, u_0 and u_1, and the total cost is

$$J = k(u_0)^2 + k(u_1)^2 + (x_0 + u_0 + u_1)^2. \tag{9.3}$$

The examples derived below consider a trajectory which starts at $x_0 = 0$. From this start point, the optimal actions are those that minimise J, i.e. $u_0 = u_1 = 0$.

9.1.2 Critic Definition

A critic function is defined using a weight vector with just two weights, $\vec{w} = (w_1, w_2)^T$:

$$\widetilde{J}(x_t, t, \vec{w}) = \begin{cases} 0 & \text{if } t = 0 \\ c_1(x_1)^2 - w_1 x_1 & \text{if } t = 1 \\ c_2(x_2)^2 - w_2 x_2 & \text{if } t = 2 \end{cases} \tag{9.4}$$

where c_1 and c_2 are positive constants. These two constants are not to be treated as weights. They were included so that a greater range of function approximators could be considered for the critic, to allow a fuller search for a divergence example, as described in Section 9.2.1. Also to ease the finding of that divergence example, this simplified critic structure was chosen (as opposed to a neural network) since it is linear in $\vec{w}$, and its weight vector has just two components.

Hence the critic-gradient function, $\widetilde{G}(x_t, t, \vec{w})$, is given by:

$$\begin{aligned} \widetilde{G}(x_t, t, \vec{w}) &:= \frac{\partial \widetilde{J}(x_t, t, \vec{w})}{\partial x_t} && \text{(by Eq. (3.3))} \\ &= \begin{cases} 0 & \text{if } t = 0 \\ 2c_t x_t - w_t & \text{if } t \in \{1, 2\}. \end{cases} && \text{(by Eq. (9.4))} \qquad (9.5) \end{aligned}$$

Note that this implies

$$\left(\frac{\partial \widetilde{G}}{\partial w_k}\right)_t = \begin{cases} -1 & \text{if } t \in \{1,2\} \text{ and } t = k \\ 0 & \text{otherwise.} \end{cases} \tag{9.6}$$

9.1.3 Unrolling a Greedy Trajectory

A *greedy trajectory* is a trajectory that is found by following greedy actions only. In a deterministic environment, greedy actions are $\vec{u}$ values that minimise $\widetilde{Q}(\vec{x}, \vec{u}, \vec{w})$ defined by Eq. (3.9).

Substituting the model functions (Eq. (9.1)) and the critic definition (Eq. (9.4)) into the $\widetilde{Q}$ function definition (Eq. (3.9)) gives, with $\gamma = 1$,

$$\begin{aligned}
\widetilde{Q}(x_t, t, u_t, \vec{w}) &:= \bar{U}(x_t, t, u_t) + \gamma \widetilde{J}(\bar{f}(x_t, t, u_t), t+1, \vec{w}) && \text{(by Eq. (3.9))} \\
&= k(u_t)^2 + \widetilde{J}(x_t + u_t, t+1, \vec{w}), \quad \text{for } t \in \{0,1\} && \text{(by Eq. (9.1))} \\
&= \begin{cases} k(u_0)^2 + c_1(x_0+u_0)^2 - w_1(x_0+u_0) & \text{if } t = 0 \\ k(u_1)^2 + c_2(x_1+u_1)^2 - w_2(x_1+u_1) & \text{if } t = 1 \end{cases} && \text{(by Eq. (9.4))} \\
&= k(u_t)^2 + c_{t+1}(x_t+u_t)^2 - w_{t+1}(x_t+u_t), \quad \text{for } t \in \{0,1\}.
\end{aligned}$$

In order to minimise this with respect to u_t and get greedy actions, we first differentiate to get,

$$\begin{aligned}
\left(\frac{\partial \widetilde{Q}}{\partial u}\right)_t &= 2ku_t + 2c_{t+1}(x_t+u_t) - w_{t+1} && \text{for } t \in \{0,1\} \\
&= 2u_t(c_{t+1}+k) - w_{t+1} + 2c_{t+1}x_t && \text{for } t \in \{0,1\} \qquad (9.7)
\end{aligned}$$

and then solve $\left(\frac{\partial \widetilde{Q}}{\partial u}\right)_t = 0$ to obtain,

$$u_0 \equiv \frac{w_1 - 2c_1x_0}{2(c_1+k)} \tag{9.8}$$

$$u_1 \equiv \frac{w_2 - 2c_2x_1}{2(c_2+k)}. \tag{9.9}$$

These two equations define the greedy-policy function, $\pi(\vec{x}, \vec{w})$, for this environment and critic function.

Since the optimal actions are $u_0 = u_1 = 0$ from a start state of $x_0 = 0$, the optimal weights are $w_1 = w_2 = 0$.

Following the greedy actions along a trajectory starting at $x_0 = 0$, and using the recursion $x_{t+1} = f(x_t, t, u_t)$ with the model functions (Eq. (9.1)) gives

$$\begin{aligned} x_1 &= x_0 + u_0 && \text{(by Eq. (9.1a))} \\ &= \frac{w_1}{2(c_1+k)} && \text{(by Eq. (9.8) \& } x_0 = 0\text{)} \end{aligned} \tag{9.10}$$

and

$$\begin{aligned} x_2 &= x_1 + u_1 && \text{(by Eq. (9.1a))} \\ &= \frac{w_2(c_1+k) + kw_1}{2(c_2+k)(c_1+k)}. && \text{(by Eqs. (9.9) \& (9.10))} \end{aligned} \tag{9.11}$$

Substituting x_1 (Eq. (9.10)) back into the equation for u_1 (Eq. (9.9)) gives u_1 purely in terms of the weights and constants:

$$u_1 \equiv \frac{w_2(c_1+k) - c_2 w_1}{2(c_2+k)(c_1+k)}. \tag{9.12}$$

9.1.4 Evaluation of Value-Gradients Along the Greedy Trajectory

We can now evaluate the $\widetilde{G}$ values by substituting the greedy trajectory's state vectors (Eqs. (9.10)-(9.11)) into Eq. (9.5), giving:

$$\begin{aligned} \widetilde{G}_1 &:= 2c_1x_1 - w_1 && \text{(by Eq. (9.5))} \\ &= \frac{c_1w_1}{(c_1+k)} - w_1 && \text{(by Eq. (9.10))} \\ &= \frac{-w_1k}{(c_1+k)} \end{aligned} \tag{9.13}$$

and

$$\begin{aligned} \widetilde{G}_2 &:= 2c_2x_2 - w_2 && \text{(by Eq. (9.5))} \\ &= \frac{w_2(c_1+k)c_2 + kw_1c_2}{(c_2+k)(c_1+k)} - w_2 && \text{(by Eq. (9.11))} \\ &= \frac{kw_1c_2 - w_2k(c_1+k)}{(c_2+k)(c_1+k)}. \end{aligned} \tag{9.14}$$

The greedy actions of Eqs. (9.8) and (9.9) both satisfy

$$\left(\frac{\partial \pi}{\partial x}\right)_t = \frac{-c_{t+1}}{c_{t+1}+k}, \qquad \text{for } t \in \{0,1\}. \tag{9.15}$$

Substituting Eq. (9.15) and Eq. (9.1) into the definition for $\left(\frac{D\bar{f}}{Dx}\right)_t$ given by Eq. (3.5), gives,

$$\begin{aligned}
\left(\frac{D\bar{f}}{Dx}\right)_t &:= \left(\frac{\partial \bar{f}}{\partial x}\right)_t + \left(\frac{\partial \pi}{\partial x}\right)_t \left(\frac{\partial \bar{f}}{\partial u}\right)_t && \text{(by Eq. (3.5))} \\
&= \left(\frac{\partial x+u}{\partial x}\right)_t + \left(\frac{\partial \pi}{\partial x}\right)_t \left(\frac{\partial x+u}{\partial u}\right)_t, \quad \text{for } t \in \{0,1\} && \text{(by Eq. (9.1a))} \\
&= 1 - \frac{c_{t+1}}{c_{t+1}+k}, \quad \text{for } t \in \{0,1\} && \text{(by Eq. (9.15))} \\
&= \frac{k}{c_{t+1}+k}, \quad \text{for } t \in \{0,1\}. && (9.16)
\end{aligned}$$

Similarly, the expression for $\left(\frac{D\bar{U}}{Dx}\right)_t$ is found by:

$$\begin{aligned}
\left(\frac{D\bar{U}}{Dx}\right)_t &:= \left(\frac{\partial \bar{U}}{\partial x}\right)_t + \left(\frac{\partial \pi}{\partial x}\right)_t \left(\frac{\partial \bar{U}}{\partial u}\right)_t && \text{(by Eq. (3.5))} \\
&= \left(\frac{\partial k(u)^2}{\partial x}\right)_t + \left(\frac{\partial \pi}{\partial x}\right)_t \left(\frac{\partial k(u)^2}{\partial u}\right)_t, \quad \text{for } t \in \{0,1\} && \text{(by Eq. (9.1b))} \\
&= 0 - \frac{c_{t+1}}{c_{t+1}+k}(2ku_t), \quad \text{for } t \in \{0,1\} && \text{(by Eq. (9.15))} \\
&= \frac{-2kc_{t+1}u_t}{c_{t+1}+k}, \quad \text{for } t \in \{0,1\}. && (9.17)
\end{aligned}$$

9.1.5 Backwards Pass along Trajectory

We do a backwards pass along the trajectory calculating the target gradients using Eq. (3.7) with $\gamma = 1$:

$$\begin{aligned}
{G'}_2 :=& \left(\frac{\partial \bar{\Phi}}{\partial x}\right)_2 && \text{(by Eq. (3.7) with } x_2 \in \mathbb{T}) \\
=& 2x_2 && \text{(by Eq. (9.2))} \\
=& \frac{w_2(c_1+k)+kw_1}{(c_2+k)(c_1+k)}. && \text{(by Eq. (9.11))} \qquad (9.18)
\end{aligned}$$

Similarly,

$$
\begin{aligned}
{G'}_1 :=& \left(\frac{D\bar{U}}{Dx}\right)_1 + \left(\frac{D\bar{f}}{Dx}\right)_1 \left(\lambda {G'}_2 + (1-\lambda)\widetilde{G}_2\right) && \text{(by Eq. (3.7) with } \gamma = 1\text{)} \\
=& \frac{-2kc_2u_1}{c_2+k} + \frac{k}{c_2+k}\left(\lambda {G'}_2 + (1-\lambda)\widetilde{G}_2\right) && \text{(by Eqs. (9.17) \& (9.16))} \\
=& \frac{-kc_2(w_2(c_1+k)-c_2w_1)}{(c_1+k)(c_2+k)^2} && \text{(by Eq. (9.12))} \\
&+ \frac{k}{c_2+k}\left(\lambda\frac{w_2(c_1+k)+kw_1}{(c_2+k)(c_1+k)}\right. && \text{(by Eq. (9.18))} \\
&\left.+(1-\lambda)\frac{kw_1c_2-w_2k(c_1+k)}{(c_2+k)(c_1+k)}\right) && \text{(by Eq. (9.14))} \\
=& \frac{w_1k(k\lambda+(c_2)^2+k(1-\lambda)c_2)}{(c_1+k)(c_2+k)^2} - \frac{w_2k(c_2-\lambda+k(1-\lambda))}{(c_2+k)^2}. && (9.19)
\end{aligned}
$$

The VGL(λ) weight update relies upon the differences ${G'}_t - \widetilde{G}_t$. These differences can now be evaluated as follows:

$$
\begin{aligned}
{G'}_1 - \widetilde{G}_1 =& \frac{w_1k(k\lambda+(c_2)^2+k(1-\lambda)c_2)}{(c_1+k)(c_2+k)^2} - \frac{w_2k(c_2-\lambda+k(1-\lambda))}{(c_2+k)^2} - \frac{-w_1k}{(c_1+k)} \\
& \text{(by Eqs. (9.19) \& (9.13))} \\
=& \left(\frac{k(k\lambda+(c_2)^2+k(1-\lambda)c_2)}{(c_1+k)(c_2+k)^2} + \frac{k}{(c_1+k)}\right)w_1 - \frac{k(c_2+k-\lambda(k+1))}{(c_2+k)^2}w_2, \\
& (9.20)
\end{aligned}
$$

and

$$
\begin{aligned}
{G'}_2 - \widetilde{G}_2 =& \frac{w_2(c_1+k)+kw_1}{(c_2+k)(c_1+k)} - \frac{kw_1c_2-w_2k(c_1+k)}{(c_2+k)(c_1+k)} && \text{(by Eqs. (9.18) \& (9.14))} \\
=& \frac{k(1-c_2)}{(c_2+k)(c_1+k)}w_1 + \frac{1+k}{(c_2+k)}w_2. && (9.21)
\end{aligned}
$$

9.1.6 Analysis of Weight-Update Equation

We now have the whole trajectory, and the terms ${G'}_t - \widetilde{G}_t$ written algebraically, so that we can next analyse the VGL(λ) weight update algebraically.

The VGL(λ) weight update (Eq. (3.6)) is comprised of

$$
\sum_t \left(\frac{\partial \widetilde{G}}{\partial w_i}\right)_t \Omega_t({G'}_t - \widetilde{G}_t), \quad \text{for } i \in \{1,2\},
$$

$$= -\Omega_i({G'}_i - \widetilde{G}_i), \quad \text{for } i \in \{1,2\}, \qquad\qquad \text{(by Eq. (9.6))}.$$

Switching to vector notation for $\vec{w}$, this is

$$\begin{aligned}\sum_t \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t({G'}_t - \widetilde{G}_t) &= -\begin{pmatrix}\Omega_1({G'}_1 - \widetilde{G}_1)\\ \Omega_2({G'}_2 - \widetilde{G}_2)\end{pmatrix}\\ &= -\begin{pmatrix}\Omega_1 & 0\\ 0 & \Omega_2\end{pmatrix}\begin{pmatrix}{G'}_1 - \widetilde{G}_1\\ {G'}_2 - \widetilde{G}_2\end{pmatrix}\\ &= DB\vec{w} \qquad (9.22)\end{aligned}$$

where

$$D := \begin{pmatrix}\Omega_1 & 0\\ 0 & \Omega_2\end{pmatrix} \qquad (9.23)$$

and

$$B := -\begin{pmatrix}\frac{k(k\lambda+(c_2)^2+k(1-\lambda)c_2)}{(c_1+k)(c_2+k)^2} + \frac{k}{(c_1+k)} & \frac{-k(c_2+k-\lambda(k+1))}{(c_2+k)^2}\\ \frac{k(1-c_2)}{(c_2+k)(c_1+k)} & \frac{1+k}{(c_2+k)}\end{pmatrix}, \quad \text{(by Eqs. (9.20) \& (9.21))}. \qquad (9.24)$$

By Eqs. (3.6) and (9.22), $\Delta\vec{w} = \alpha DB\vec{w}$ is the VGL(λ) weight update written as a single dynamic system of $\vec{w}$.

To add further complexity to the system, in order to achieve the desired divergence, we next define $\vec{w}$ to be a linear function of two *other* weights, $\vec{p} = (p_1, p_2)^T$, such that $\vec{w} = F\vec{p}$, where F is a 2×2 constant real matrix. The VGL(λ) weight-update equation can now be recalculated for these new weights, as follows:

$$\begin{aligned}\Delta\vec{p} &= \alpha \sum_t \left(\frac{\partial \widetilde{G}}{\partial \vec{p}}\right)_t \Omega_t({G'}_t - \widetilde{G}_t) && \text{(by Eq. (3.6))}\\ &= \alpha \sum_t \frac{\partial \vec{w}}{\partial \vec{p}} \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t({G'}_t - \widetilde{G}_t) && \text{(by chain rule)}\\ &= \alpha \frac{\partial \vec{w}}{\partial \vec{p}} \sum_t \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t({G'}_t - \widetilde{G}_t) && \text{(since independent of } t)\\ &= \alpha \frac{\partial \vec{w}}{\partial \vec{p}} DB\vec{w} && \text{(by Eq. (9.22))}\\ &= \alpha(F^T DBF)\vec{p}. && \text{(by } \vec{w} = F\vec{p} \text{ and } \frac{\partial \vec{w}}{\partial \vec{p}} = \frac{\partial (F\vec{p})}{\partial \vec{p}} = F^T) \qquad (9.25)\end{aligned}$$

Eq. (9.25) represents the whole learning system, described as a dynamical system of the weight vector $\vec{p}$.

9.2 Divergence Results

Now that the difficult work of finding the analytical expression for the VGL(λ) weight update has been completed, it is relatively straightforward to find the divergence results sought. The following subsections give the divergence results for VGL(0), VGL(1), VGLΩ(0), TD(0) and TD(1). Plus, to demonstrate that the divergence results do not contradict the convergence proof of Chapter 8, the algorithm VGLΩ(1) is shown to converge under the same learning parameters which caused divergence for VGL(1).

9.2.1 Divergence Examples for DHP and VGL(1)

It is now shown that the algorithms VGL(0) (i.e. DHP) and VGL(1) can both be made to diverge with a greedy policy in this problem domain.

We consider the VGL(0) and VGL(1) algorithms with the Ω_t matrix equal to the identity matrix, which implies that the D matrix (defined by Eq. (9.23)) satisfies $D = I$, the 2×2 identity matrix, and hence we can ignore D from Eq. (9.25).

The optimal actions $u_0 = u_1 = 0$ would be achieved by $\vec{p} = \vec{0}$. To produce a divergence example, we want to ensure that $\vec{p}$ does *not* converge to $\vec{0}$.

Taking $\alpha > 0$ to be sufficiently small, then the weight vector $\vec{p}$ evolves according to a continuous-time linear dynamic system (Eq. (9.25), with D ignored), and this system is stable if and only if the matrix product $F^T BF$ is "stable" (i.e. if the real part of every eigenvalue of this matrix product is negative). The logic here is that if it is proven to diverge for a continuous-time system, i.e. in the limit of an infinitesimal learning rate, then it would also diverge for any small finite learning rate too.

Choosing $\lambda = 0$, with $c_1 = c_2 = k = 0.01$ gives $B = \left(\begin{smallmatrix} -0.75 & 0.5 \\ -24.75 & -50.5 \end{smallmatrix}\right)$ (by Eq. (9.24)). Choosing $F = \left(\begin{smallmatrix} 10 & 1 \\ -1 & -1 \end{smallmatrix}\right)$ makes $F^T BF = \left(\begin{smallmatrix} 117.0 & -38.25 \\ 189.0 & -27.0 \end{smallmatrix}\right)$ which has eigenvalues $45 \pm 45.22i$. Since the real parts of these eigenvalues are positive, Eq. (9.25) will diverge for VGL(0) (i.e. DHP).

Also, perhaps surprisingly, it is possible to get instability with VGL(1). Choosing $c_2 = k = 0.01$, $c_1 = 0.99$ gives $B = \left(\begin{smallmatrix} -0.2625 & -24.75 \\ -0.495 & -50.5 \end{smallmatrix}\right)$. Choosing $F = \left(\begin{smallmatrix} -1 & -1 \\ .2 & .02 \end{smallmatrix}\right)$ makes

$F^TBF = \left(\begin{smallmatrix} 2.7665 & 0.1295 \\ 4.4954 & 0.2222 \end{smallmatrix}\right)$ which has two positive real eigenvalues. Therefore this VGL(1) system diverges.

Fig. 9.1 shows the divergences obtained for VGL(0) and VGL(1) with a greedy policy.

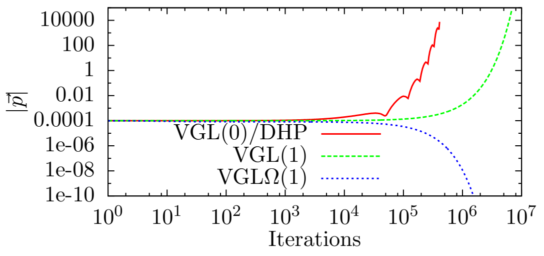


Figure 9.1: Diverging behaviour for VGL(0) (i.e. DHP) and VGL(1), using the learning parameters described in Section 9.2.1 and a learning rate of $\alpha = 10^{-6}$; and converging behaviour on the same problem for VGLΩ(1), as described in Section 9.2.2, with $\alpha = 10^{-3}$.

9.2.2 Convergence Results for VGLΩ(1)

To show that the above divergence result for VGL(1) does not contradict the convergence proof of Chapter 8, the algorithm VGLΩ(1) is now shown to converge under the same conditions that caused divergence for VGL(1).

VGLΩ(λ) is defined to be VGL(λ) with the Ω_t matrix defined by Eq. (3.8).

To construct the Ω_t matrix of Eq. (3.8), we differentiate Eq. (9.7) to get

$$\left(\frac{\partial^2 \widetilde{Q}}{\partial u \partial u}\right)_t = 2(c_{t+1} + k) \qquad \text{for } t \in \{0, 1\}, \tag{9.26}$$

and then substitute Eq. (9.26) and $\left(\frac{\partial \bar{f}}{\partial u}\right)_t = 1$ (by Eq. (9.1a)) into Eq. (3.8), to get

$$\Omega_t = \begin{cases} 1/(2(c_t + k)) & \text{for } t \in \{1, 2\} \\ 0 & \text{for } t = 0. \end{cases} \tag{9.27}$$

This Ω_t matrix can be used directly in the original VGL(λ) weight-update equation (Eq. (3.6)), or alternatively it can be used in the matrix for D given by Eq. (9.23) and the weight update given by Eq. (9.25). In either case, as predicted by the convergence proof of Chapter 8, it was not possible to make the VGLΩ(1) weight update diverge.

An example of VGLΩ(1) converging, under the same conditions that caused VGL(1) to diverge, is given in Fig. 9.1.

9.2.3 Divergence Result for VGLΩ(0)

Divergence for VGLΩ(0) was derived as follows: Substituting the above Ω_t matrix from Eq. (9.27) into the D matrix of Eq. (9.23) gives

$$D = \begin{pmatrix} \frac{1}{2(c_1+k)} & 0 \\ 0 & \frac{1}{2(c_2+k)} \end{pmatrix}. \tag{9.28}$$

This D matrix can be used in the analytical expression found for the VGL(λ) weight update applied to the environment defined in this chapter, by substituting D into Eq. (9.25). Then, substituting the same parameters that made VGL(0) diverge, i.e. $c_1 = c_2 = k = 0.01$, into Eq. (9.28) gives $D = \left(\begin{smallmatrix} 25 & 0 \\ 0 & 25 \end{smallmatrix}\right)$. Since D is a positive multiple of the identity matrix, its presence in Eq. (9.25) will not affect the stability of the product $F^T DBF$, so the system for $\vec{p}$ will still be unstable, and diverge, just as it did for VGL(0) (where D was taken to be the identity matrix). So unfortunately, using the Ω_t matrix of Eq. (3.8) does not force reliable convergence for VGL(0) (i.e. DHP) with a greedy policy.

9.3 Divergence Results for TD(λ) and HDP

To satisfy the requirement for value exploration in TD(λ)-based algorithms, it is necessary to supplement the greedy policies (Eqs. (9.8) and (9.9)) with a small amount of stochastic Gaussian noise with zero mean and variance 0.0001. This Gaussian noise is necessary, since these algorithms can converge to severely sub-optimal solutions without value exploration (e.g. see Fig. 3.1).

To achieve divergence of these algorithms with the noisy greedy policy, the same learning and environment constants as used for the VGL(0) and VGL(1) divergence experiments were used again. These choices of parameters, with the stochastic noise added to the greedy policy, made TD(0) and TD(1) diverge respectively, as shown in Fig. 9.2 and Fig. 9.3. Hence HDP diverges too, since this is equivalent to TD(0) with the given policy.

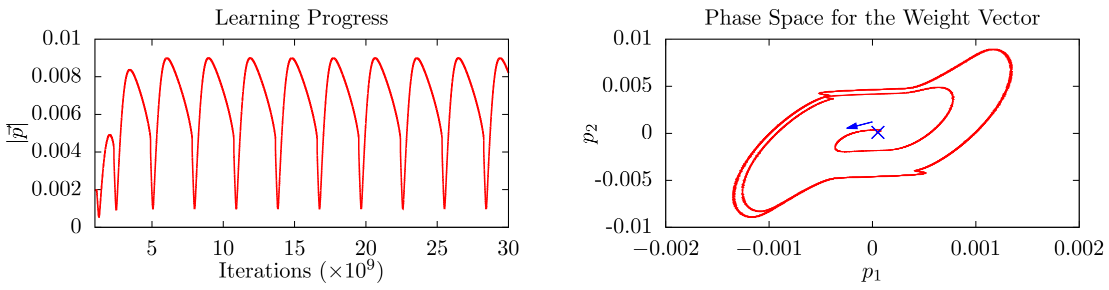


Figure 9.2: Divergence for TD(0) generated with the diverging parameters described in Section 9.3, and a learning rate of $\alpha = 10^{-6}$. The left-hand graph shows progress of $|\vec{p}|$ versus iterations. The oscillation shown in this left graph occurs because the weight vector goes into a limit cycle after divergence. This limit cycle is shown fully in the right-hand graph, which shows the evolution of the weight vector (p_1, p_2) in phase space. The phase curve starts close to the origin (at the 'X'), then moves in the direction of the arrow, and finally finishes off in a limit cycle. The weight vector initially moves in the opposite direction from the origin, i.e. away from the target fixed point; confirming divergence.

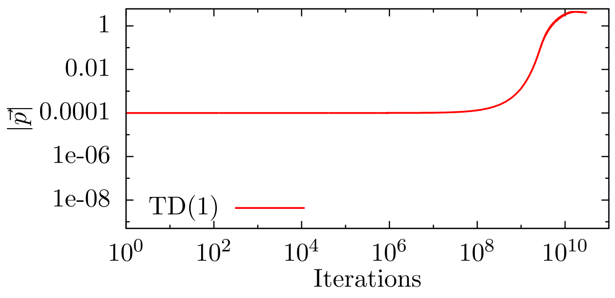


Figure 9.3: Divergence for TD(1) generated with the diverging parameters described in Section 9.3, and a learning rate of $\alpha = 10^{-6}$.

Although these divergence results for the TD(λ) based algorithms were only found empirically, as opposed to the results for the previous sections which were first found analytically, these results do still have value. Firstly, the empirical results are easily replicable (source code for the empirical experiments available by (Fairbank and Alonso, 2011a, in ancillary files)). Secondly, an insight into why the divergence parameters for VGL were sufficient to make the TD(λ) based algorithms diverge too is because TD(λ) with stochastic exploration can be understood to be an approximation to a stochastic version of VGL(λ). Since TD(λ) eventually has to learn the value gradients if it is ever to achieve an optimal trajectory, so it would be expected that a divergence example for VGL(λ) will cause divergence for TD(λ) too.

Without the stochastic noise added to the greedy policy, these examples would not diverge, but instead converge to a sub-optimal policy, which is also considered a failure. Fig. 9.4 shows results for this situation.

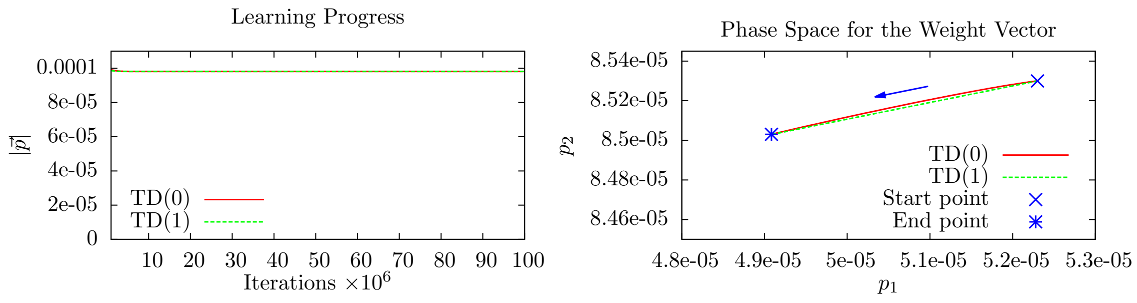


Figure 9.4: The performance of TD(0) and TD(1) in the divergence problem in the absence of stochastic value exploration. Under these circumstances, the weight vector $\vec{p}$ does not move much at all before hitting a fixed point, as shown in the left-hand figure. The right-hand figure shows in a phase-space view that both TD(0) and TD(1) reach the same fixed point, which is indicated by the asterisk in the figure. This fixed point is just 1.9% closer to the optimal weights ($\vec{p} = (0, 0)$) than the initial weights were. This is another example of VL failing in the absence of value exploration.

9.4 Chapter Conclusions

It has been shown that under a greedy policy, all but one of the ADPRL algorithms have been made to diverge. The algorithm that didn't diverge was VGLΩ(1), which was proven to converge in Chapter 8.

These are new divergence results for TD(λ), in that previous examples of divergence have only been for TD(0) and for non-greedy policies (Baird, 1995; Tsitsiklis and Van Roy, 1996a,b). The divergences achieved for TD(1) and VGL(1) were only possible because of the use of a greedy policy (or equivalently, value-iteration).

A conclusion of this chapter is that the diverging algorithms considered cannot currently be reliably used for a greedy policy, or equivalently, under value-iteration, and instead can only be used under the form of policy iteration described in Section 2.7.5 if provable convergence is required. However, as noted in Section 5.1.1, there are clear efficiency advantages of using value-iteration over this form of policy iteration. The algorithm VGLΩ(1) solves all of these problems by being proven to converge with a greedy policy.

The divergence results of this chapter were derived for linear-quadratic critic functions, as this was the situation that allowed for easiest analysis to derive concrete divergence examples. It is assumed that similar divergence results will exist for neural-network based critic functions, since neural networks are more complex structures that should allow for more possibilities for divergence situations similar to the simple example here. In practical experience, divergence often does occur when using a greedy policy with a neural-network critic, but these situations are harder to analyse and make replicable. In this neural-divergence situation, it seems reasonable to speculate that a second-order Taylor-series expansion of the neural network could be made about the fixed point of the learning process, and locally this approximation could be behaving very similarly to the quadratic functions used in this chapter.

This chapter also demonstrates how a value-gradient analysis can be useful to value-learning research. Since value-learning must eventually learn value-gradients in order to produce optimal trajectories, the value-gradient divergence example was sufficient to also cause divergence for the value-learning algorithms. The theoretical analysis is much easier coming from a value-gradient approach though, since a VGL weight update has a very predictable and direct effect on the greedy policy (as shown in Lemma 8.4).

Part III

Advanced Implementation Details

Chapter 10

Implementing Clipping correctly

In ADPRL, the agent moves along a trajectory which terminates only when (and if) a terminal state is reached. As shown in Fig. 10.1, clipping is the concept of calculating the exact fraction in the final time step at which a boundary of terminal states is reached, and stopping the agent exactly at this boundary. The name clipping is taken by analogy to the concept in computer graphics. Without clipping, the discretization of time would cause the agent to penetrate slightly beyond the terminal boundary, as shown in the figure.

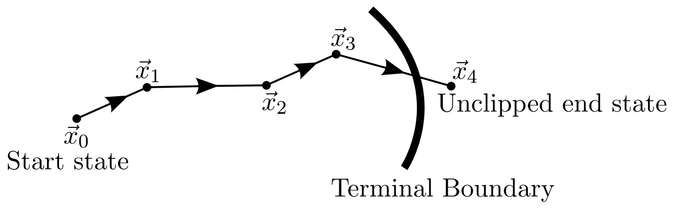


Figure 10.1: A Trajectory Reaching a Terminal State, without Clipping. The thick curved line indicates a boundary of terminal states. In this diagram, clipping does not take place, and the trajectory penetrates beyond the terminal boundary. When clipping is used correctly, the intention is to stop the agent exactly at the point of intersection between the trajectory and the terminal boundary.

In this chapter, it is shown that when a large final impulse of cost $\Phi(\vec{x}, \vec{e})$ is given at a terminal state $\vec{x} \in \mathbb{T}$, then failure to do clipping in the final time step of the trajectory can very significantly distort the direction of the learning gradient used by certain ADP algorithms, and thus prevent successful solution of the ADP problem. It is also shown that this problem is not lessened by sampling the time steps of the underlying continuous-time process at a higher rate. This problem affects commonly used ADP algorithms such as DHP (Alg. 3.1), BPTT (Alg. 6.1), and the main algorithm of this thesis, VGL(λ) (Algs. 3.2 and 3.3). These algorithms are all very closely related to each other, and for purposes of explaining clipping as clearly as possible, BPTT will be used as the example.

As described in Chapter 6, BPTT works by calculating the quantity $\frac{\partial J}{\partial \vec{z}}$ directly and very efficiently for each trajectory sampled, enabling gradient descent to be performed on J with respect to $\vec{z}$. However if clipping is omitted then the gradient that BPTT calculates can be distorted enough to prevent learning. Fig. 10.2 illustrates the problems that arise without clipping.

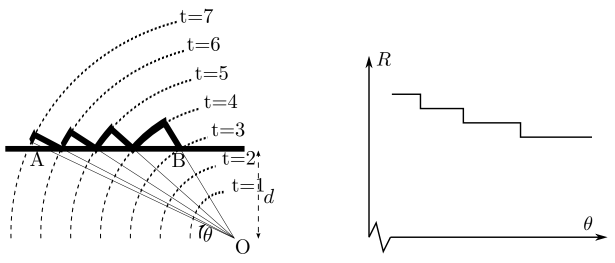


Figure 10.2: An Example of the Problems that Can Occur when Clipping is Not Used. The left figure shows that spurious zigzag boundary penetrations can occur when clipping is not used. The right figure shows a corresponding graph of R, the trajectory length, versus θ, and this yields no useful local gradient information. Hence minimising R with respect to θ using only $dR/d\theta$ would be impossible.

In Fig. 10.2 (left) the agent starts at O and travels in a straight line at a constant speed, along a fixed chosen initial angle, θ. The straight line AB is a terminal boundary (i.e. a continuous line of states in $\mathbb{T}$). The dotted arcs represent the integer time steps that the agent passes through. If clipping is not used then the agent will stop on the first integer time step (i.e. on the first dotted arc) after passing the terminal boundary. This means the agent will finally stop at a point somewhere on the bold zigzag path from A to B. Fig. 10.2 (right) shows how the distance the agent travelled before stopping (R) varies with θ. If the cost-to-go function J was defined to be the total distance travelled before termination (i.e. if $J := R$), and the parameter vector of J was defined to be θ, then the ADP objective would be to minimise R with respect to θ. But Fig. 10.2 (right) shows that there is no useful gradient information for learning, since $\frac{\partial J}{\partial \theta} = \frac{\partial R}{\partial \theta} = 0$, whenever it exists, and hence gradient descent on J with respect to θ would fail without clipping.

Situations can get even worse than this: In Fig. 10.3, a pathological example is shown, where the gradient of the graph is always in the opposite direction of the global minimum of R. This could occur for example if we were trying to minimise the function $J := R + y$ with respect to θ, for the situation in Fig. 10.2, where y is the final y-coordinate of the agent, and R is the distance travelled before stopping.

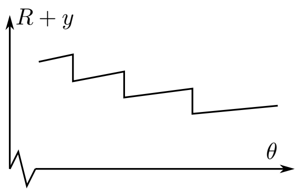


Figure 10.3: A Pathological Example: Local gradient is Opposite to Global Gradient.

In general, increasing the sampling rate of the discretization of time will not solve the problem, since that would simply make the dotted arcs in Fig. 10.2 squeeze closer together, and will make the teeth of the saw-tooth blade shape in Fig. 10.3 finer. The gradients in Figs. 10.2 and 10.3 would still not be helpful for learning.

This chapter shows how to solve the problem by incorporating clipping into the model and cost functions, $f(\vec{x}, \vec{u}, \vec{e})$ and $U(\vec{x}, \vec{u}, \vec{e})$, when terminal states are reached. BPTT, VGL(λ) and DHP make intensive use of the derivatives of these two functions, and hence it is necessary to differentiate carefully through the clipped versions of these functions. This is the important step derived in this chapter, and this step corrects the gradient $\frac{\partial J}{\partial \vec{z}}$ to make it suitable for learning, and solves the problems explained by Fig. 10.2 and Fig. 10.3.

The necessity for clipping affects any algorithm which calculates the derivatives of the model function, i.e. $\frac{\partial f}{\partial \vec{x}}$ directly, and when terminal states that deliver impulses of cost are present. For example the RL method of Munos (2006), which implements a continuous-time numerical differentiation to evaluate $\frac{\partial J}{\partial \vec{z}}$, will also be affected by this clipping problem. Likewise, the ADP methods of BPTT, DHP, GDHP and Value-Gradient Learning are also affected by the requirement for clipping.

Clipping is not necessary for any problem where the termination condition is simply when a fixed integer number of time steps is reached, as discussed further in Section 10.2.1. Also the experiments in this chapter show that TD(0)/HDP do not need clipping, since these algorithms do not make significant use of the derivatives of the model function. For the same reasons, the policy-gradient learning methods of Chapter 6 (Peters and Schaal, 2006; Williams, 1992) do not require clipping either. Policy-gradient methods are discussed further in Section 10.4.

In this chapter, only episodic environments will be considered; that is environments where all trajectories are guaranteed to meet a terminal state eventually. Hence the sampled cost-to-go function to be minimised is the episodic version defined by Eq. (1.4), which is:

$$\widehat{J}(\vec{x}_0, \vec{\mathbf{e}}_0, \vec{z}) := \mathbb{E}\left(\sum_{t=0}^{T-1} \gamma^t U_t + \gamma^T \Phi(\vec{x}_T, \vec{e}_T)\right) \tag{10.1}$$

subject to the usual equations for $\vec{x}_{t+1} = f(\vec{x}_t, \vec{u}_t, \vec{e}_t)$, $U_t = U(\vec{x}_t, \vec{u}_t, \vec{e}_t)$ and $\vec{u}_t = A(\vec{x}_t, \vec{e}_t, \vec{z})$ (Eqs. (1.1), (1.2) and (1.3), respectively).

In the rest of this chapter, in Section 10.1, formulae for the clipped model and cost functions are described, and it is shown how these can be used to evaluate a single trajectory with clipping. Then in Section 10.2, details of the solution to the clipping problem is described, by correctly differentiating through the clipped model functions, as is required for effective gradient descent. These details are used to generate pseudocode for BPTT and VGL(λ) with clipping, in Alg. 10.2 and Alg. 10.3 respectively. In Section 10.3 experimental details are given for neural-network control problems, both with and without clipping. One of these problems is the classic Cart-Pole benchmark problem which is formulated in a way which would be impossible for VGL methods to solve without clipping, and it is shown that the clipping methods enable this problem to be solved efficiently. In Section 10.4, policy-gradient learning methods are described, and details of why they don't require clipping are given, despite the methods' similarity to BPTT. Finally, in Section 10.5, chapter conclusions are given.

10.1 Using Clipping in Trajectory Evaluation

The formulae for the clipped model and cost functions are now derived. The clipped versions of the original functions will be denoted with a superscripted C, so that f^C, U^C and $\widehat{J}^C$ will be the function names for the clipped versions of the model, cost and sampled cost-to-go functions, respectively. The functions f^C and U^C are only defined for any state $\vec{x}_t$ that occurs immediately before a terminal state is reached, i.e. for which $\vec{x}_t \notin \mathbb{T}$ and for which $f(\vec{x}_t, \vec{u}_t, \vec{e}_t) \in \mathbb{T}$.

These three clipped functions, f^C, U^C and $\widehat{J}^C$, are key concepts in this chapter, because defining them clearly allows them to be differentiated carefully (in Section 10.2), and hence used to calculate the learning gradients correctly.

10.1.1 Calculation of the Clipped Model and Cost Functions

Suppose the agent is transitioning between states $\vec{x}_t$ and $f(\vec{x}_t, \vec{u}_t, \vec{e}_t)$, and the state $f(\vec{x}_t, \vec{u}_t, \vec{e}_t)$ would be beyond the terminal boundary unless clipping was applied. To calculate the clipping correctly, we imagine this state transition as occurring along the straight line segment from $\vec{x}_t$ to $f(\vec{x}_t, \vec{u}_t, \vec{e}_t)$, i.e. the straight line given parametrically by position vector

$$\vec{r} = \vec{x}_t + s\vec{v}, \tag{10.2}$$

where

$$\vec{v} = f(\vec{x}_t, \vec{u}_t, \vec{e}_t) - \vec{x}_t, \tag{10.3}$$

and $s \in [0, 1]$ is a real parameter. This is illustrated in Fig. 10.4.

This straight line must intersect a boundary of terminal states. At the point of intersection, the tangent plane of the terminal boundary is given by $(\vec{r} - \vec{P}) \cdot \vec{n} = 0$ (i.e. where $\vec{n}$ is the plane's normal vector, $\vec{P}$ is a fixed point on the plane, $\vec{r}$ is a general point on the plane, and where $\cdot$ denotes the inner product between two vectors), as illustrated in Fig. 10.4. The constants $\vec{P}$ and $\vec{n}$ should be available from either the physical environment or from the collision-detection routine of the simulated environment.

At the intersection of the line and the plane, we have

$$\begin{aligned} &(\vec{x}_t + s\vec{v} - \vec{P}) \cdot \vec{n} = 0 \\ \Rightarrow &s = \frac{(\vec{P} - \vec{x}_t) \cdot \vec{n}}{\vec{v} \cdot \vec{n}}. \end{aligned}$$

This value of s is a real number between 0 and 1 which indicates the fraction along the transition line from $\vec{x}_t$ to $f(\vec{x}_t, \vec{u}_t, \vec{e}_t)$ at which the terminal boundary was encountered. The value s will be referred to as the "clipping fraction", and since it depends on $\vec{x}_t$, $\vec{u}_t$, $\vec{P}$ and $\vec{n}$, it is defined by the function:

$$s := S(\vec{x}_t, \vec{u}_t, \vec{e}_t, \vec{P}, \vec{n}) := \frac{(\vec{P} - \vec{x}_t) \cdot \vec{n}}{(f(\vec{x}_t, \vec{u}_t, \vec{e}_t) - \vec{x}_t) \cdot \vec{n}}. \tag{10.4}$$

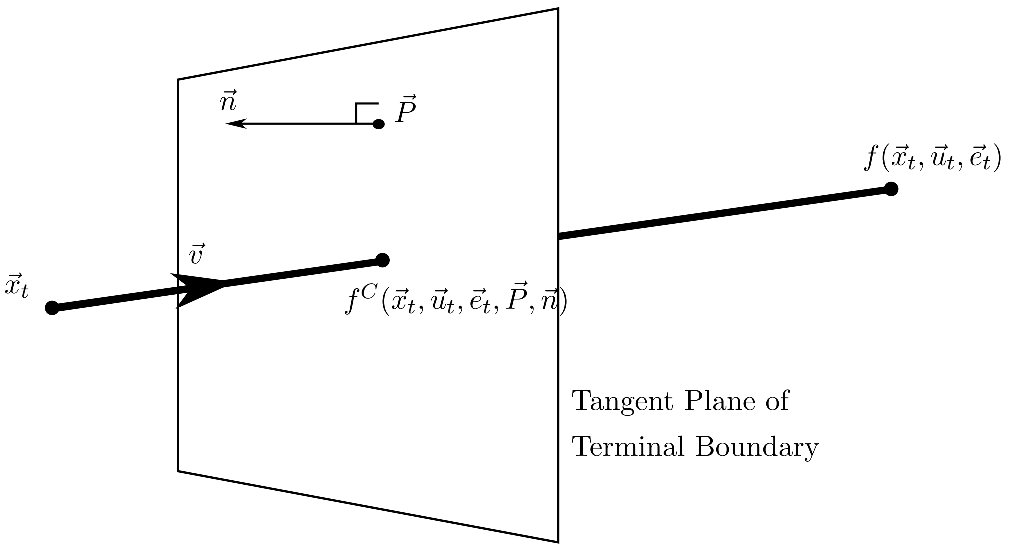

$\vec{n}$
$\vec{P}$
$f(\vec{x}_t, \vec{u}_t, \vec{e}_t)$
$\vec{v}$
$\vec{x}_t$
$f^C(\vec{x}_t, \vec{u}_t, \vec{e}_t, \vec{P}, \vec{n})$
Tangent Plane of
Terminal Boundary

Figure 10.4: The Final State Transition of a Trajectory Crossing the Tangent Plane of a Terminal Boundary. The unclipped line goes from $\vec{x}_t$ to $f(\vec{x}_t, \vec{u}_t, \vec{e}_t)$. The line intersects the plane at a point given by the new *clipped* model function $f^C(\vec{x}_t, \vec{u}_t, \vec{e}_t, \vec{P}, \vec{n})$.

Hence the clipped value of the final state is $\vec{x}_{t+1} = \vec{x}_t + S(\vec{x}_t, \vec{u}_t, \vec{e}_t, \vec{P}, \vec{n})(f(\vec{x}_t, \vec{u}_t, \vec{e}_t) - \vec{x}_t)$, which is found by combining Eqs. (10.2), (10.3) and (10.4). This gives the function for the clipped model function as

$$f^C(\vec{x}, \vec{u}, \vec{e}, \vec{P}, \vec{n}) := \vec{x} + S(\vec{x}, \vec{u}, \vec{e}, \vec{P}, \vec{n})(f(\vec{x}, \vec{u}, \vec{e}) - \vec{x}). \tag{10.5}$$

Assuming that "cost" is delivered at a uniform rate during the final state transition, the total clipped cost would be proportional to the clipping fraction, giving:

$$U^C(\vec{x}, \vec{u}, \vec{e}, \vec{P}, \vec{n}) := S(\vec{x}, \vec{u}, \vec{e}, \vec{P}, \vec{n})U(\vec{x}, \vec{u}, \vec{e}). \tag{10.6}$$

Since the final clipped time step has duration $s \in [0, 1]$, the terminal cost $\Phi(\vec{x}_T, \vec{e}_T)$ should only receive a discount of γ^s instead of the full discount γ. Hence, at the penultimate time step, $\vec{x}_{T-1}$, the total cost-to-go is

$$J^C(\vec{x}_{T-1}, \mathbf{\vec{e}}_{T-1}, \vec{z}) := U^C(\vec{x}_{T-1}, \vec{u}_{T-1}, \vec{e}_{T-1}, \vec{P}, \vec{n}) + \gamma^s \Phi(\vec{x}_T, \vec{e}_T). \tag{10.7}$$

Deciding to use γ^s in place of γ might seem like a trivial detail, but when differentiated, it provides useful information for the correct learning gradient, with clipping.

This detail allows solution of a version of the Cart-Pole benchmark problem, in Section 10.3.2, which would otherwise be impossible for VGL methods.

Alg. 10.1 illustrates how Eqs. (1.1)-(1.3) and Eqs. (10.4)-(10.7) would be used to evaluate a trajectory with clipping. This algorithm is an updated version of the original trajectory-unroll algorithm (Alg. 1.1) which did not include clipping.

Algorithm 10.1 Unrolling a Trajectory with Clipping.

```
1:  $t \leftarrow 0$, $\widehat{J}^C \leftarrow 0$
2:  while $\vec{x}_t \notin \mathbb{T}$ do
3:      $\vec{u}_t \leftarrow A(\vec{x}_t, \vec{e}_t, \vec{z})$
4:      $\vec{x}_{t+1} \leftarrow f(\vec{x}_t, \vec{u}_t, \vec{e}_t)$
5:      if $\vec{x}_{t+1} \in \mathbb{T}$ then
6:          Identify $\vec{P}$ and $\vec{n}$ by inspection of the intersection with the terminal boundary,
            $\mathbb{T}$.
7:          $s \leftarrow S(\vec{x}_t, \vec{u}_t, \vec{e}_t, \vec{P}, \vec{n})$ {using Eq. (10.4)}
8:          $T \leftarrow t+1$
9:          $\vec{x}_T \leftarrow \vec{x}_t + s\left(\vec{x}_T - \vec{x}_t\right)$
10:         $\widehat{J}^C \leftarrow \widehat{J}^C + \left(\gamma^t\right)\left(sU(\vec{x}_t, \vec{u}_t, \vec{e}_t) + \gamma^s \Phi(\vec{x}_T, \vec{e}_T)\right)$
11:     else
12:         $\widehat{J}^C \leftarrow \widehat{J}^C + \left(\gamma^t\right) U(\vec{x}_t, \vec{u}_t, \vec{e}_t)$
13:     end if
14:     $t \leftarrow t+1$
15: end while
16: $T \leftarrow t$
```

Note that $\vec{P}$ and $\vec{n}$ are required by Eqs. (10.4)-(10.6). These would be found during the collision-detection routine (i.e. line 6 of Alg. 10.1), from knowledge of the terminal-boundary orientation, together with knowledge of $\vec{x}_{T-1}$ and $f(\vec{x}_{T-1}, \vec{u}_{T-1}, \vec{e}_{T-1})$. Knowledge of the orientation of the terminal boundary could come from a model of the physical environment's boundary; or if this model was not available, then a physical inspection of the actual boundary would need to take place. Examples of how these two vectors were found in this chapter's experiments are given in Sections 10.3.1 and 10.3.2.

10.2 Calculation of the Derivatives of the Clipped Model and Cost Functions

The ADP algorithms BPTT, DHP and VGL(λ) require the derivatives of the model function, and hence they will require the derivatives of the clipped model function $f^C(\vec{x}, \vec{u}, \vec{e}, \vec{P}, \vec{n})$ too. Fig. 10.5 shows how different the derivative of f^C can be from the derivative of f, and hence how important it is to get this correct in ADPRL. This figure clarifies why algorithms that are dependent on $\frac{\partial f^C}{\partial \vec{x}}$ are critically affected by the need for clipping, and also that just reducing the duration of each time step tracking or simulating the motion will not solve the problem at all.

This section describes how to calculate the derivatives of $f^C(\vec{x}, \vec{u}, \vec{e}, \vec{P}, \vec{n})$ and $U^C(\vec{x}, \vec{u}, \vec{e}, \vec{P}, \vec{n})$. This is what allows the clipping problem to be solved. Hence this section is the key part of this chapter, in terms of providing implementation details for solving the clipping problem. Pseudocode is given for the modified version of the algorithms BPTT and VGL(λ).

Differentiating the formula for $S(\vec{x}, \vec{u}, \vec{e}, \vec{P}, \vec{n})$ in Eq. (10.4) gives:

$$\begin{aligned}
\frac{\partial S(\vec{x}, \vec{u}, \vec{e}, \vec{P}, \vec{n})}{\partial \vec{x}} &= \frac{\partial}{\partial \vec{x}}\left(\frac{(\vec{P}-\vec{x})\cdot\vec{n}}{(f(\vec{x},\vec{u},\vec{e})-\vec{x})\cdot\vec{n}}\right) && \text{(by Eq. (10.4))}\\
&= \frac{-\vec{n}}{\vec{v}\cdot\vec{n}} - \frac{(\vec{P}-\vec{x})\cdot\vec{n}}{(\vec{v}\cdot\vec{n})^2}\frac{\partial(f(\vec{x},\vec{u},\vec{e})-\vec{x})\cdot\vec{n}}{\partial\vec{x}} && \text{(using Eq. (10.3))}\\
&= \frac{-\vec{n}}{\vec{v}\cdot\vec{n}} - \frac{(\vec{P}-\vec{x})\cdot\vec{n}}{(\vec{v}\cdot\vec{n})^2}\left(\frac{\partial f}{\partial \vec{x}} - I\right)\vec{n}
\end{aligned}$$

where I is the identity matrix, and the matrix notation is as defined in Section 3.3. Similarly,

$$\begin{aligned}
\frac{\partial S(\vec{x}, \vec{u}, \vec{e}, \vec{P}, \vec{n})}{\partial \vec{u}} &= \frac{\partial}{\partial \vec{u}}\left(\frac{(\vec{P}-\vec{x})\cdot\vec{n}}{(f(\vec{x},\vec{u},\vec{e})-\vec{x})\cdot\vec{n}}\right) && \text{(by Eq. (10.4))}\\
&= -\frac{(\vec{P}-\vec{x})\cdot\vec{n}}{(\vec{v}\cdot\vec{n})^2}\frac{\partial(f(\vec{x},\vec{u},\vec{e})-\vec{x})\cdot\vec{n}}{\partial\vec{u}} && \text{(using Eq. (10.3))}\\
&= -\frac{(\vec{P}-\vec{x})\cdot\vec{n}}{(\vec{v}\cdot\vec{n})^2}\left(\frac{\partial f}{\partial \vec{u}}\right)\vec{n}
\end{aligned}$$

For the reasons highlighted previously in Section 3.2.1, it is often more practical to work with derivatives of the learned environment functions instead of the true envi-

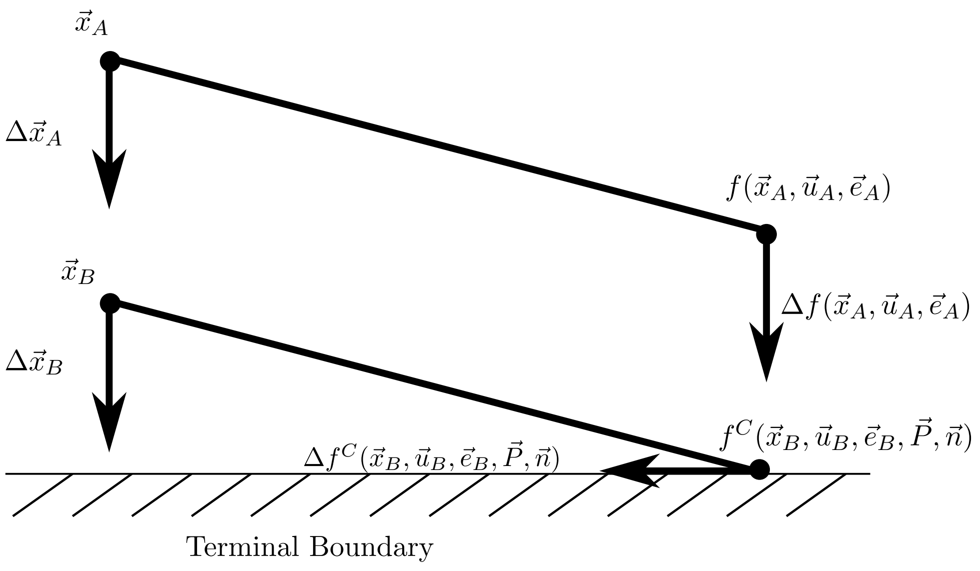

$\vec{x}_A$
$\Delta\vec{x}_A$
$f(\vec{x}_A,\vec{u}_A,\vec{e}_A)$
$\vec{x}_B$
$\Delta f(\vec{x}_A,\vec{u}_A,\vec{e}_A)$
$\Delta\vec{x}_B$
$f^C(\vec{x}_B,\vec{u}_B,\vec{e}_B,\vec{P},\vec{n})$
$\Delta f^C(\vec{x}_B,\vec{u}_B,\vec{e}_B,\vec{P},\vec{n})$
Terminal Boundary

Figure 10.5: The Discontinuous Change in Derivatives of the Model Function $f(\vec{x},\vec{u},\vec{e})$ at a Terminal Boundary. The straight line segment from $\vec{x}_A$ to $f(\vec{x}_A,\vec{u}_A,\vec{e}_A)$ represents a state transition that is not intersecting the terminal boundary. If the start of this line segment is perturbed in the direction of the arrow $\Delta\vec{x}_A$ then its other end will move in the direction indicated by the arrow $\Delta f(\vec{x}_A,\vec{u}_A,\vec{e}_A)$. The line segment below, however, which starts at $\vec{x}_B$, does reach the terminal boundary. If the start of this line segment is moved in the direction of $\Delta\vec{x}_B$, then its end will move in a perpendicular direction, as indicated by the arrow $\Delta f^C(\vec{x}_B,\vec{u}_B,\vec{e}_B,\vec{P},\vec{n})$. This indicates that $\left(\frac{\partial f^C}{\partial \vec{x}}\right)_A$ is very different from $\left(\frac{\partial f}{\partial \vec{x}}\right)_B$, and hence this needs treating carefully in the ADP algorithms.

ronment functions; and these are usually only approximations to the true derivatives (except in circumstances highlighted in Section 3.2.1). In this case, the above two equations would be modified into:

$$\frac{\partial S(\vec{x},\vec{u},\vec{e},\vec{P},\vec{n})}{\partial \vec{x}} \approx \frac{-\vec{n}}{\vec{v}\cdot\vec{n}} - \frac{(\vec{P}-\vec{x})\cdot\vec{n}}{(\vec{v}\cdot\vec{n})^2}\left(\frac{\partial \bar{f}}{\partial \vec{x}} - I\right)\vec{n} \qquad \text{(by Eq. (3.1))} \qquad (10.8)$$

$$\frac{\partial S(\vec{x},\vec{u},\vec{e},\vec{P},\vec{n})}{\partial \vec{u}} \approx -\frac{(\vec{P}-\vec{x})\cdot\vec{n}}{(\vec{v}\cdot\vec{n})^2}\left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)\vec{n} \qquad \text{(by Eq. (3.1))} \qquad (10.9)$$

Similarly, under this assumption, in all of the following equations (Eqs. (10.10)-(10.15)), derivatives of the functions $f(\vec{x},\vec{u},\vec{e})$, $U(\vec{x},\vec{u},\vec{e})$ or $\Phi(\vec{x},\vec{e})$, are replaced by approximate derivatives of the corresponding learned functions. However if this assumption is not required, then these approximations can be replaced by their corresponding exact equalities.

Using the derivatives of $S(\vec{x},\vec{u},\vec{e},\vec{P},\vec{n})$ in Eqs. (10.8)-(10.9), we can now differentiate the clipped model and cost functions, giving:

$$\begin{aligned}
\frac{\partial f^C(\vec{x},\vec{u},\vec{e},\vec{P},\vec{n})}{\partial \vec{x}} &= I + \frac{\partial S}{\partial \vec{x}}\vec{v}^T + s\left(\frac{\partial f}{\partial \vec{x}} - I\right) && \text{(by Eqs. (10.3)-(10.5))} \\
&\approx I + \frac{\partial S}{\partial \vec{x}}\vec{v}^T + s\left(\frac{\partial \bar{f}}{\partial \vec{x}} - I\right) && \text{(by Eq. (3.1))} \qquad (10.10) \\
\frac{\partial f^C(\vec{x},\vec{u},\vec{e},\vec{P},\vec{n})}{\partial \vec{u}} &= \frac{\partial S}{\partial \vec{u}}\vec{v}^T + s\frac{\partial f}{\partial \vec{u}} && \text{(by Eqs. (10.3)-(10.5))} \\
&\approx \frac{\partial S}{\partial \vec{u}}\vec{v}^T + s\frac{\partial \bar{f}}{\partial \vec{u}} && \text{(by Eq. (3.1))} \qquad (10.11) \\
\frac{\partial U^C(\vec{x},\vec{u},\vec{e},\vec{P},\vec{n})}{\partial \vec{x}} &= \frac{\partial S}{\partial \vec{x}}U(\vec{x},\vec{u},\vec{e}) + s\frac{\partial U}{\partial \vec{x}} && \text{(by Eq. (10.6))} \\
&\approx \frac{\partial S}{\partial \vec{x}}U(\vec{x},\vec{u},\vec{e}) + s\frac{\partial \bar{U}}{\partial \vec{x}} && \text{(by Eq. (3.1))} \qquad (10.12) \\
\frac{\partial U^C(\vec{x},\vec{u},\vec{e},\vec{P},\vec{n})}{\partial \vec{u}} &= \frac{\partial S}{\partial \vec{u}}U(\vec{x},\vec{u},\vec{e}) + s\frac{\partial U}{\partial \vec{u}} && \text{(by Eq. (10.6))} \\
&\approx \frac{\partial S}{\partial \vec{u}}U(\vec{x},\vec{u},\vec{e}) + s\frac{\partial \bar{U}}{\partial \vec{u}} && \text{(by Eq. (3.1))} \qquad (10.13)
\end{aligned}$$

The cost-to-go function for the penultimate time step, Eq. (10.7), can be rewritten as a Q-function of both $\vec{x}$ and $\vec{u}$, to give

$$Q(\vec{x}_{T-1},\vec{u}_{T-1}) := U^C(\vec{x}_{T-1},\vec{u}_{T-1},\vec{e}_{T-1},\vec{P},\vec{n}) + \gamma^s\Phi(f^C(\vec{x}_{T-1},\vec{u}_{T-1},\vec{e}_{T-1},\vec{P},\vec{n})). \qquad (10.14)$$

Differentiating this with respect to $\vec{u}_{T-1}$ or $\vec{x}_{T-1}$ gives:

$$\begin{aligned}
\left(\frac{\partial Q}{\partial \bullet}\right)_{T-1} &= \left(\frac{\partial U^C}{\partial \bullet}\right)_{T-1} + \gamma^s\left(\left(\frac{\partial f^C}{\partial \bullet}\right)_{T-1}\left(\frac{\partial \Phi}{\partial \vec{x}}\right)_T + (\ln\gamma)\left(\frac{\partial S}{\partial \bullet}\right)_{T-1}\Phi(\vec{x}_T,\vec{e}_T)\right) \\
&\approx \left(\frac{\partial U^C}{\partial \bullet}\right)_{T-1} + \gamma^s\left(\left(\frac{\partial f^C}{\partial \bullet}\right)_{T-1}\left(\frac{\partial \bar{\Phi}}{\partial \vec{x}}\right)_T + (\ln\gamma)\left(\frac{\partial S}{\partial \bullet}\right)_{T-1}\Phi(\vec{x}_T,\vec{e}_T)\right) \quad \text{(by Eq. (3.1))}
\end{aligned} \qquad (10.15)$$

where $\bullet$ represents either $\vec{u}$ or $\vec{x}$.

This equation, which relies upon the derivatives of $f^C(\vec{x},\vec{u},\vec{e},\vec{P},\vec{n})$ and $U^C(\vec{x},\vec{u},\vec{e},\vec{P},\vec{n})$ (as defined in Eqs. (10.8) to (10.13)), can be used to modify BPTT from Alg. 6.1 into its corresponding "with clipping" version given in Alg. 10.2. Eq. (10.15) appears in the algorithm directly in lines 9-10. This completes the description of the corrections to BPTT to include clipping. It would be possible (and advisable) to verify any implementation of BPTT to check that it is calculating $\frac{\partial \widehat{J}^C(\vec{x}_0,\vec{\mathbf{e}}_0,\vec{z})}{\partial \vec{z}}$ correctly by numerical differentiation (using Alg. 10.1).

Algorithm 10.2 Backpropagation Through Time for Control, with Clipping.

```
1: Unroll full trajectory from start state $\vec{x}_0$ using Alg. 10.1, and retain the variables $\vec{x}_t$, $\vec{u}_t$, $T$, $s$, $\vec{P}$ and $\vec{n}$.
2: $\frac{\partial \widehat{J}}{\partial \vec{z}} \leftarrow \vec{0}$
3: $\vec{p}_T \leftarrow \left(\frac{\partial \bar{\Phi}}{\partial \vec{x}}\right)_T$
4: for $t = T-1$ to 0 step $-1$ do
5:   if $\vec{x}_{t+1} \in \mathbb{T}$ then
6:     Calculate $\left(\frac{\partial S}{\partial \vec{x}}\right)_t$ and $\left(\frac{\partial S}{\partial \vec{u}}\right)_t$ by Eqs. (10.8) and (10.9).
7:     Calculate $\left(\frac{\partial f^C}{\partial \vec{x}}\right)_t$ and $\left(\frac{\partial f^C}{\partial \vec{u}}\right)_t$ by Eqs. (10.10) and (10.11).
8:     Calculate $\left(\frac{\partial U^C}{\partial \vec{x}}\right)_t$ and $\left(\frac{\partial U^C}{\partial \vec{u}}\right)_t$ by Eqs. (10.12) and (10.13).
9:     $Q_x \leftarrow \left(\frac{\partial U^C}{\partial \vec{x}}\right)_t + \gamma^s \left(\left(\frac{\partial f^C}{\partial \vec{x}}\right)_t \vec{p}_{t+1} + (\ln \gamma) \left(\frac{\partial S}{\partial \vec{x}}\right)_t \Phi(\vec{x}_T, \vec{e}_T)\right)$
10:    $Q_u \leftarrow \left(\frac{\partial U^C}{\partial \vec{u}}\right)_t + \gamma^s \left(\left(\frac{\partial f^C}{\partial \vec{u}}\right)_t \vec{p}_{t+1} + (\ln \gamma) \left(\frac{\partial S}{\partial \vec{u}}\right)_t \Phi(\vec{x}_T, \vec{e}_T)\right)$
11:  else
12:    $Q_x \leftarrow \left(\frac{\partial \bar{U}}{\partial \vec{x}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{x}}\right)_t \vec{p}_{t+1}$
13:    $Q_u \leftarrow \left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \vec{p}_{t+1}$
14:  end if
15:  $\frac{\partial \widehat{J}}{\partial \vec{z}} \leftarrow \frac{\partial \widehat{J}}{\partial \vec{z}} + \gamma^t \left(\frac{\partial A}{\partial \vec{z}}\right)_t Q_u$
16:  $\vec{p}_t \leftarrow Q_x + \left(\frac{\partial A}{\partial \vec{x}}\right)_t Q_u$
17: end for
18: $\vec{z} \leftarrow \vec{z} - \beta \frac{\partial \widehat{J}}{\partial \vec{z}}$
```

The VGL(λ) algorithm needs similar modifications to convert it to include clipping. Alg. 10.3 gives pseudocode for the "with clipping" version of VGL(λ). Since DHP is a special case of VGL(λ), i.e. DHP is the same as VGL(0), this pseudocode covers DHP too. The modifications used are essentially the same as were made for BPTT. However, one extra difference was applied in the clipped VGL(λ) algorithm compared to its unclipped version (Alg. 3.2). That is that the clipped version is modified so that it does not learn (or need to learn) the value-gradient at the terminal state. The reason for this change is because the value-gradient can change discontinuously at the terminal boundary, as shown in Fig. 10.5, and therefore it would not be easy for a smooth function approximator to have to learn a discontinuous change like this.

Even though this chapter argues that clipping is not really necessary for VL methods, for the purposes of empirical comparison, it is worth describing which modifications the VL algorithms would need, to incorporate clipping. For the HDP algorithm, clipping needs applying to the final time step of the trajectory unroll, which can be implemented by replacing the line "$\vec{x}_{t+1} \leftarrow f(\vec{x}_t, \vec{u}_t, \vec{e}_t)$" (i.e. line 4 of Alg. 2.1) by lines 4-13 of Alg. 10.1. Also, the expression for $\left(\frac{\partial \bar{U}}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial \bar{f}}{\partial \vec{u}}\right)_t \left(\frac{\partial \tilde{J}}{\partial \vec{x}}\right)_{t+1}$, which appears in line 7 of Alg. 2.1, should be replaced by Q_u, which needs calculating by lines 5-14 of Alg. 10.2. When these changes are made, clipping is fully included into HDP.

Algorithm 10.3 VGL(λ), with Clipping; Actor-Critic, Batch-Mode Implementation.

```
1: Unroll full trajectory from start state $\vec{x}_0$ using Alg. 10.1, and retain the variables
   $\vec{x}_t$, $\vec{u}_t$, $T$, $s$, $\vec{P}$ and $\vec{n}$.
2: $\vec{p} \leftarrow \left(\frac{\partial \Phi}{\partial \vec{x}}\right)_T$, $\Delta\vec{z} \leftarrow \vec{0}$, $\Delta\vec{w} \leftarrow \vec{0}$
3: {Backwards pass...}
4: for $t = T-1$ to 0 step $-1$ do
5:    Calculate $Q_x$ and $Q_u$ by lines 5-14 of Alg. 10.2
6:    $G'_t \leftarrow Q_x + \left(\frac{\partial A}{\partial \vec{x}}\right)_t Q_u$
7:    $\Delta\vec{w} \leftarrow \Delta\vec{w} + \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t \left(G'_t - \widetilde{G}_t\right)$
8:    $\Delta\vec{z} \leftarrow \Delta\vec{z} - \left(\frac{\partial A}{\partial \vec{z}}\right)_t Q_u$
9:    $\vec{p} \leftarrow \lambda G'_t + (1-\lambda)\widetilde{G}_t$
10: end for
11: $\vec{w} \leftarrow \vec{w} + \alpha\Delta\vec{w}$
12: $\vec{z} \leftarrow \vec{z} + \beta\Delta\vec{z}$
```

10.2.1 Clipping with Trajectories of Fixed or Variable Finite Length

In situations where trajectories are of fixed finite length (commonly referred to as a fixed-length finite-horizon problem), clipping is not necessary. This is in contrast to the problems considered in the chapter introduction, which were variable finite-length problems, since the trajectory lengths were determined by the environment (e.g. a trajectory terminates only when the agent crashes into a wall). In this section, these two situations will be distinguished from each other by referring to them as "fixed finite-length" and "variable finite-length" problems, respectively. Only in variable finite-length problems is clipping necessary.

In the fixed finite-length problem, the clipping fraction defined by Eq. (10.4) is always $s \equiv 1$, and therefore $\frac{\partial S}{\partial \vec{x}} = \vec{0}$, $\frac{\partial S}{\partial \vec{u}} = \vec{0}$ and $\gamma^s = \gamma$. Hence the clipped model and cost functions are identical to their unclipped counterparts, and therefore it is not necessary to implement any program code specifically to handle clipping. This might be one reason why the need for clipping has not previously been noted in the DHP/BPTT research literature, since most finite-horizon problems considered have been fixed-finite length.

However the fixed finite-length problem does have one minor different complication, in that it is often necessary to include the time step into the state vector. This is because the optimal actions and cost-to-go function will often be dependent upon the number of incomplete steps in a trajectory.

10.3 Experimental Results

This section describes two neural-network based ADPRL experiments which require clipping to be solved well.

In all experiments the action and critic networks used were MLPs. Each MLP had $\dim(\vec{x})$ input nodes, 2 hidden layers of 6 nodes each, and one output layer, with short-cut connections connecting all pairs of layers. The output layers were dimensioned as follows: Each action network had $\dim(\vec{u})$ output nodes; each HDP critic network had 1 output node; and each DHP critic had $\dim(\vec{x})$ output nodes. All network nodes had bias weights, as is usual in MLP architectures. The activation functions used were hyperbolic tangent functions, except for the critic network's output layer which was always a linear activation function (with linear slope as specified in the individual experiments, below). At the start of each experimental trial, neural weights were initialised randomly in the range $[-.1, .1]$, with uniform probability distribution.

10.3.1 Vertical-Lander Problem

This is a repeat of the Vertical-Lander problem defined in Section 2.8.1. Previously when this experiment was described, the description of clipping was omitted, but now the full details are given.

Trajectories terminate as soon as the spacecraft hits the ground ($x_h = 0$) or runs out of fuel ($x_f = 0$). These two conditions define $\mathbb{T}$. On termination, the algorithms need to choose values for $\vec{P}$ and $\vec{n}$, which describe the orientation of the terminal-boundary tangent plane. These choices are given for this experiment in Table 10.1. In the case that the final unclipped state transition crosses both terminal planes, then the one that is crossed first (i.e the one that produces a smaller clipping fraction by Eq. (10.4)) is to be used.

Termination Condition Breached	Position vector of Plane, $\vec{P}^T$	Normal Vector to Plane, $\vec{n}^T$
$x_h \leq 0$ (hits ground)	(0,0,0)	(1,0,0)
$x_f \leq 0$ (no fuel)	(0,0,0)	(0,0,1)

Table 10.1: Terminal Boundary Planes used in Vertical-Lander Experiment. The state vector used here is $\vec{x} = (x_h, x_v, x_f)^T$.

Physical environment constants used were the same as before ($k_g = 0.2$; $k_f = 4$; $k_u = 1$; $\Delta\tau = 1$; $\gamma = 1$). The input vector to the action and critic networks was $\vec{x}' = (x_h/100, x_v/10, x_f/50)^T$, and the model and cost functions were redefined to act on this rescaled input vector directly. The action network's output y was rescaled to give the action by $A(\vec{x}, \vec{e}, \vec{z}) := (y + 1)/2$ directly. Each algorithm was tested in batch mode, operating on five trajectories simultaneously. Those five trajectories had fixed start points, which had been randomly chosen in the region $x_h \in (0, 100)$, $x_v \in (-10, 10)$ and $x_f = 30$.

Fig. 10.6 shows learning performance of the BPTT, VGL(0), VGL(1) and HDP algorithms, both with and without clipping. Each graph shows five curves, and each curve shows the learning performance from a different random weight initialisation. The learning rates for the four algorithms were: BPTT ($\beta = 0.01$); VGL(0) ($\beta = 0.001$, $\alpha = 0.00001$); VGL(1) ($\beta = 0.001$, $\alpha = 0.000001$); and HDP ($\beta = 0.00001$, $\alpha = 0.00001$). The critic-network's output layer's activation function had a linear slope of 20 in the VGL(λ) experiments and 10 in the HDP experiment.

To supply stochastic exploration for the HDP experiment, random noise was added according to Eq. (2.16), for HDP only, with noise standard-deviation equal to $\sigma = 0.1$.

These graphs show the clear stability and performance advantages of using clipping correctly for the BPTT and DHP algorithms. The graphs also confirm that the HDP algorithm is not significantly affected by the need for clipping.

Fig. 10.7 shows that the need for clipping is not made arbitrarily small just by using a smaller $\Delta\tau$ value.

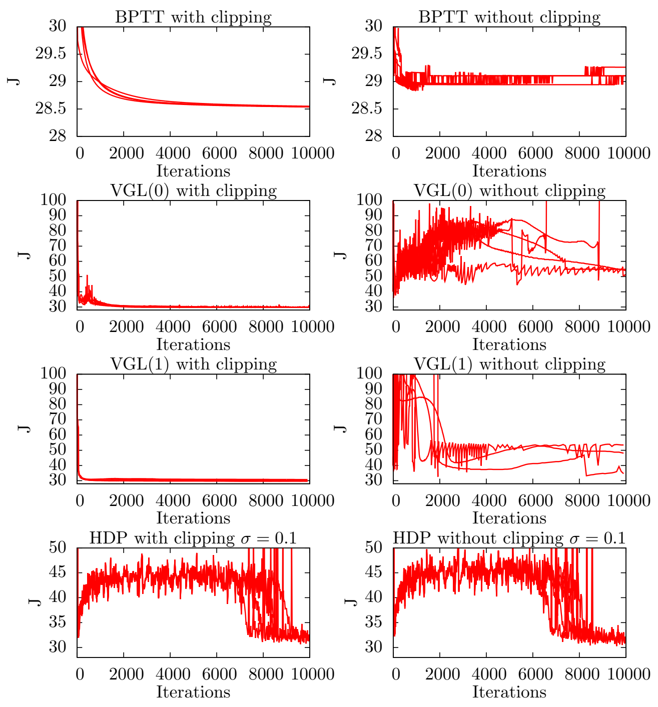


Figure 10.6: Vertical-Lander Solutions by BPTT, VGL(0), VGL(1) and HDP using $\Delta\tau = 1$.

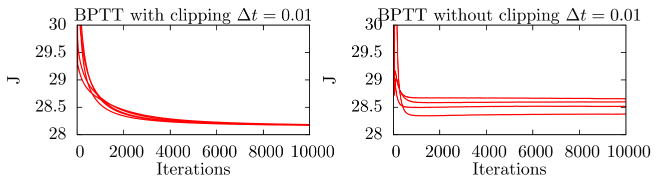


Figure 10.7: Vertical Lander with $\Delta\tau = 0.01$.

10.3.2 Cart-Pole Experiment

The Cart-Pole benchmark problem was described in Section 5.3.2. This experiment is now repeated but with a different trajectory termination condition, and a modified cost function. This version of the cost function is the one used most commonly in the

RL community (e.g. Barto et al., 1983), and would not work well with VGL methods, without clipping.

In this version of the problem, terminal states ($\mathbb{T}$) are defined to be any state with $|x| \geq 2.4$, or $|\theta| \geq \frac{\pi}{15}$ (i.e. 12 degrees), or $t \geq 300$. Hence the pole motion continues until it reaches a terminal state or until the pole is successfully balanced for 300 time steps, i.e. 6 seconds of real time. Termination plane constants are given in Table 10.2.

Termination Condition Breached	Position vector of Plane, $\vec{P}^T$	Normal Vector to Plane, $\vec{n}^T$
$\theta \geq \pi/15$	$(0,0,\pi/15,0,0)$	$(0,0,-1,0,0)$
$\theta \leq -\pi/15$	$(0,0,-\pi/15,0,0)$	$(0,0,1,0,0)$
$x \geq 2.4$	$(2.4,0,0,0,0)$	$(-1,0,0,0,0)$
$x \leq -2.4$	$(-2.4,0,0,0,0)$	$(1,0,0,0,0)$
$t \geq 300$	$(0,0,0,0,300)$	$(0,0,0,0,1)$

Table 10.2: Terminal Boundary Planes used in Cart-Pole Experiment. The state vector used here is $\vec{x} = (x, \dot{x}, \theta, \dot{\theta}, t)^T$.

The duration-based cost function of Barto et al. (1983) is equivalent to

$$U(\vec{x}, u) := 0, \tag{10.16}$$

for non-terminal states, and,

$$\Phi(\vec{x}, \vec{e}) := \begin{cases} 1 & \text{if } T < 300, \\ 0 & \text{otherwise,} \end{cases} \tag{10.17}$$

for terminal states $\vec{x} \in \mathbb{T}$. When the above two cost functions are used in conjunction with a discount factor $\gamma < 1$, and when the pole eventually falls over (i.e. when $T < 300$), the total trajectory cost is $J(\vec{x}_0, \vec{\mathbf{e}}_0, \vec{z}) \equiv \gamma^T$, where T is the time at which the trajectory terminated. Since this function decreases with T, minimising it will increase T, i.e. lead to successful pole balancing. However, unless clipping is used properly, the duration will be an integer number of time steps, and since this is not smooth and differentiable, it will cause problems (become impossible) for VGL(λ), DHP and BPTT. Hence traditionally when DHP or BPTT are used for the Cart-Pole problem, the differentiable cost function used in the experiment of Section 5.3.2 would be used. In this section clipping is used to solve the problem with the cost functions of Eqs. (10.16)-(10.17).

Four algorithms, BPTT, VGL(0), VGL(1) and HDP, were applied to this problem, with a discount factor $\gamma = 0.97$. To ensure the state vector was suitably scaled for input to the MLPs, rescaled state vectors $\vec{x}'$ were used, defined by $\vec{x}' := (0.16x, 15\theta/\pi, \dot{x}, 4\dot{\theta}, t/300)^T$, with θ in radians, throughout the implementation. Note that time was an input to the neural network, as the environment's behaviour does depend on t through the termination condition.

The output of the action network, y, was multiplied by 10 to give the control force $F = A(\vec{x}, \vec{e}, \vec{z}) := 10y$. The learning rates for the algorithms were: BPTT ($\beta = 0.1$); VGL(0) ($\alpha = 0.01$, $\beta = 0.0001$); VGL(1) ($\alpha = 0.001$, $\beta = 0.0001$); HDP ($\beta = 0.001$, $\alpha = 0.01$). The VGL(λ) and HDP critics used a final-layer activation-function slope of 0.1. HDP used a policy exploration rate of $\sigma = 0.15$, and the other algorithms used $\sigma = 0$.

Learning took place on five trajectories simultaneously, with fixed start points randomly chosen from the region $|x| < 2.4$, $|\theta| < \frac{\pi}{15}$, $\dot{x} = 0$, $\dot{\theta} = 0$. The exact derivatives of the model and cost functions were made available to the algorithms.

The performance of the four tested algorithms, both with and without clipping, are shown in Fig. 10.8. Each graph shows the average balancing duration over all five trajectories, versus the training iteration. Each graph shows an ensemble of five different curves, with each curve representing a training run from a different random weight initialisation.

The results show that using clipping correctly enables both the VGL(λ) and BPTT algorithms to solve this problem consistently, and without clipping it is impossible for both algorithms. Also the results show that HDP is largely unaffected by the need for clipping.

10.4 A Note on Policy-Gradient Methods

As described in Chapter 6, both BPTT and policy-gradient learning (PGL) methods work by doing gradient descent on the total cost-to-go function $\widehat{J}(\vec{x}, \vec{\mathbf{e}}, \vec{z})$ with respect to $\vec{z}$. PGL methods are stochastic algorithms which accumulate a mean weight update

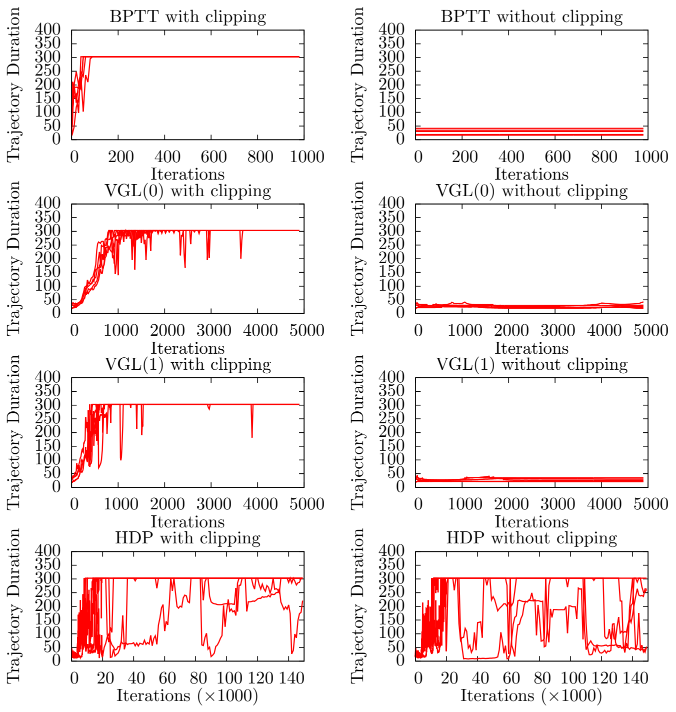


Figure 10.8: Cart-Pole Solutions by BPTT, VGL(0), VGL(1) and HDP

of the form

$$\mathbb{E}\left(\Delta\vec{z}\right) = -\beta\frac{\partial\mathbb{E}\left(\widehat{J}\right)}{\partial\vec{w}}. \tag{10.18}$$

Although these weight updates superficially look similar to the BPTT design, they do not use any explicit derivatives of the model or cost functions, and thus are not affected by the need for clipping. For example, the REINFORCE weight update, defined in Chapter 6, does not include any derivatives of $f(\vec{x},\vec{u},\vec{e})$ in its weight update.

The implementation of the BPTT algorithm, in Alg. 6.1 or Alg. 10.2, does require derivatives of $f(\vec{x},\vec{u},\vec{e})$, and hence does require clipping. In stochastic environments, the BPTT weight update would average to

$$\mathbb{E}\left(\Delta\vec{z}\right) = -\beta\mathbb{E}\left(\frac{\partial\widehat{J}}{\partial\vec{w}}\right). \tag{10.19}$$

So how can it be reconciled that for two such similar algorithms, BPTT requires clipping, but PGL does not? The answer lies in the subtle difference between Eqs.

(10.18) and (10.19), i.e. the fact that in general, the derivative of a mean is different from the mean of a derivative. In the PGL case, the $\mathbb{E}\left(\widehat{J}\right)$ term has a blurring effect which first smoothes out all of the jagged bumps in the $\widehat{J}$ versus $\vec{z}$ graph (for example as shown in Fig. 10.2), and then PGL performs gradient descent on this blurred-out graph. In contrast, BPTT first calculates the gradient of various randomly chosen points of this graph, and then averages out the answer, and clearly in the case of Fig. 10.2, this approach will not work (unless clipping is done).

This shows that PGL methods have an advantage over BPTT methods in avoiding the need for clipping. This advantage of PGL methods over BPTT methods is part of quite a long list of pros and cons between the two methods, which was given in Section 6.4.

10.5 Chapter Conclusions

The problem of clipping for ADPRL algorithms has been demonstrated and motivated. Without clipping, algorithms which rely on the derivatives of the model and cost functions can fail to work. The solution is to apply clipping, and then to correctly differentiate the model and cost functions in the final time step. This solution has been given in the form of the equations, plus in the form of clear pseudocode for the major affected ADP algorithms: VGL(λ) and BPTT.

Two neural-network experiments have confirmed the importance of applying clipping correctly. These included a Cart-Pole experiment, where clipping was found to be essential, and in the Vertical-Lander experiment, where clipping produced a significant improvement of performance.

The situations in which clipping is needed have been made clear, and those situations where it can be ignored have also been specified.

Chapter 11

Second-Order Gradient Calculations in Neural Networks

As described in Chapter 3, VGL algorithms such as dual heuristic programming (DHP) and VGL(λ) can use either a scalar critic or a vector critic. A scalar critic would be represented by a neural-network with one output node, and a vector-critic would be represented by a neural network with $\dim(\vec{x})$ output nodes. If the vector critic was chosen, then its output would form the critic-gradient function, $\widetilde{G}(\vec{x},\vec{w})$ directly. If the scalar critic was used, $\widetilde{J}(\vec{x},\vec{w})$, say, then the gradient of its output with respect to $\vec{x}$ would form the critic-gradient function, i.e. $\widetilde{G}(\vec{x},\vec{w}) := \frac{\partial \widetilde{J}(\vec{x},\vec{w})}{\partial \vec{x}}$. The differences between vector and scalar critics are described further in Section 3.4.1.

If a scalar critic is used, then the usual VGL(λ) weight update (Eq. (3.6)),

$$\Delta\vec{w} = \alpha \sum_t \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t \left({G'}_t - \widetilde{G}_t\right),$$

would implicitly transform into,

$$\Delta\vec{w} = \alpha \sum_t \left(\frac{\partial^2 \widetilde{J}}{\partial \vec{w}\partial \vec{x}}\right)_t \Omega_t \left({G'}_t - \left(\frac{\partial \widetilde{J}}{\partial \vec{x}}\right)_t\right), \tag{11.1}$$

since $\widetilde{G} \equiv \frac{\partial \widetilde{J}}{\partial \vec{x}}$.

This chapter describes how to efficiently program the second-order backpropagation necessary to implement the above weight update (Eq. (11.1)), particularly with regards to the second-derivative term $\frac{\partial^2 \widetilde{J}}{\partial \vec{w}\partial \vec{x}}$. Here, and throughout this chapter, the matrix $\frac{\partial^2 \widetilde{J}}{\partial \vec{w}\partial \vec{x}}$

is defined to have i, j component equal to $\frac{\partial^2 \widetilde{J}}{\partial \vec{w}^i \partial \vec{x}^j}$.

Globalized DHP (GDHP) is a variant on DHP that is a hybrid between DHP and HDP, i.e. it is a linear combination of a value-learning weight update plus a value-gradient learning weight update, as defined in Section 3.4.4. Hence for GDHP, it is mandatory to use a scalar critic, since there is a value-learning component.

In the field of ADP, the algorithm DHP is used far more commonly than GDHP. For example DHP has been used in control problems (Lendaris and Paintz, 1997; Prokhorov and Wunsch, 1997a), power grid control (Venayagamoorthy and Wunsch, 2003; Venayagamoorthy et al., 2002), and many other applications (Wang et al., 2009). One of the reasons for the comparatively low take-up of GDHP is that GDHP is more challenging to implement than DHP. The difficult step of implementation is to correctly and efficiently calculate the above described second-order derivative, $\frac{\partial^2 \widetilde{J}}{\partial \vec{w} \partial \vec{x}}$. Hence this chapter seeks to address this problem.

This second-order derivative matrix is related to, but slightly different to, the usual Hessian matrix, $\frac{\partial^2 y}{\partial \vec{w} \partial \vec{w}}$, for a neural network with output y, described in the neural-network literature (e.g. sec. 4.10 of Bishop, 1995). The difference is that the second-order derivative matrix required by Eq. (11.1) is a mixed derivative matrix, i.e. it is a derivative with respect to both $\vec{w}$ and $\vec{x}$; but the Hessian matrix is a derivative with respect to $\vec{w}$ twice.

In VGL algorithms, the mixed second-order derivative matrix, $\frac{\partial^2 \widetilde{J}}{\partial \vec{w} \partial \vec{x}}$, is only ever required as an inner product $\frac{\partial^2 \widetilde{J}}{\partial \vec{w} \partial \vec{x}} \vec{k}$, where $\vec{k}$ is a column-vector constant of dimension $\dim(\vec{x})$. For example, in the weight update given by Eq. (11.1), $\vec{k}$ would be given by

$$\vec{k} = \Omega_t \left({G'}_t - \left(\frac{\partial \widetilde{J}}{\partial \vec{x}} \right)_t \right). \tag{11.2}$$

To form this inner product by matrix-vector multiplication would take time $O(\dim(\vec{w}) \dim(\vec{x})^2)$. This chapter describes a very clear and straightforward algorithm to calculate this inner product directly and exactly in an asymptotically faster time of $O(\dim(\vec{w}))$.

Existing literature does briefly outline an equally efficient method to calculate the required inner product for GDHP, but this outline is only in the form of schematic diagrams (Prokhorov and Wunsch, 1997a), or a high level description of generic back-propagation (Werbos, 1987; Werbos et al., 1992). Simple pseudocode applicable for a generic feed-forward neural network is not available there.

These existing descriptions calculate the required second derivatives by applying the mathematical technique of backpropagation *twice* to the neural network's feed-forward equations (Werbos, 1974). Using the terminology of "automatic differentiation" (Rall, 1981; Werbos, 2005), this is called a *reverse accumulation* of the derivatives. But it is not a trivial task to create a correct implementation of this for the given error function. In fact this difficulty is thought to be one of the reasons that the equations for the required second-order derivatives have not been published before for a generic neural network.

The method presented here is to do a *forward accumulation* of the derivatives. This is much easier to derive and implement, and equally efficient. The method follows the technique and terminology of Pearlmutter (1994), which is used to calculate an inner product of the Hessian matrix of a neural network in a fast and exact manner, without explicitly finding the full Hessian matrix itself.

The difference in simplicity in derivation between the forwards and backwards accumulation methods is very significant, as is illustrated by the way the two techniques were used to find fast products with the Hessian matrix, in the neural-network literature. Here, the backward accumulation is described by Møller (1993), and the derivation takes several pages (dense with lemmas and equations). The forward accumulation is described by Pearlmutter (1994), and the derivation takes just one page to define a differential operator ("R"), which then is used to produce the five lines of the algorithm instantly. Despite the technically demanding accomplishment of Møller (1993), it is the vastly simpler technique of Pearlmutter (1994) that receives nearly all of the citations in the literature.

After making some necessary modifications to the technique of Pearlmutter (1994), an algorithm is obtained that is almost trivial to derive, and easy to state in pseudocode. By stating this forward accumulation method for GDHP/VGL(λ) clearly, and giving pseudocode for the resulting algorithm, it is hoped that using a scalar critic for VGL would be just as simple to implement as a vector critic.

Besides being useful for VGL, the quantity $\frac{\partial^2 \tilde{J}}{\partial \vec{w} \partial \vec{x}} \vec{k}$ is also useful in the general circumstance of trying to adjust the weights of a neural network with output y so as to force the gradient $\frac{\partial y}{\partial \vec{x}}$ to equal a given target value at a given $\vec{x}$. This could be achieved by doing gradient descent on an error function $E = \frac{1}{2}|\frac{\partial y}{\partial \vec{x}} - \vec{t}|^2$, where $\vec{t}$ is the "target" gradient. In this case we would choose the constant vector $\vec{k}$ by

$$\vec{k} = \frac{\partial E}{\partial(\partial y / \partial \vec{x})} = \frac{\partial y}{\partial \vec{x}} - \vec{t}. \tag{11.3}$$

An example of an application like this is given in Section 11.2.2.

A further requirement by VGL algorithms is an expression for $\frac{\partial A(\vec{x},\vec{e},\vec{z})}{\partial \vec{x}}$, where $A(\vec{x},\vec{e},\vec{z})$ is the output of the action-network. When a greedy-policy function is used, this derivative can require inner-products of the form $\frac{\partial \widetilde{G}}{\partial \vec{x}}\vec{k}$, for example as is required by Eq. (5.13). When a scalar critic is used, this translates into $\frac{\partial^2 \widetilde{J}}{\partial \vec{x} \partial \vec{x}}\vec{k}$, which is another second derivative of the scalar-critic network. This inner product is also calculated efficiently by the algorithm presented in this chapter.

In the rest of this chapter, in Section 11.1, the neural network and gradient finding algorithms are defined. In Section 11.2, experimental results are given. Finally, in Section 11.3, chapter conclusions are given.

11.1 The Algorithms

In this section, an algorithm is derived to find the required second-order gradients. A general feed-forward neural-network architecture is defined in Section 11.1.1, and the second derivatives are derived for it in Section 11.1.2. Details of how the method can be extended to find the full second-derivative matrices are given in Section 11.1.3.

The method for finding the second-derivative matrices for this neural network is a general technique that could be applied to any existing feed-forward network structure (or even a recurrent neural network that has been unrolled using backpropagation through time). Verification techniques for the algorithm's correctness are described in Section 11.2.1.

11.1.1 Feed-Forward Neural-Network Architecture and Backpropagation

Lines 2 to 8 of Alg. 11.1 implement a general neural network $y(\vec{x},\vec{w})$, which has a single input layer, and is fully connected with all short-cut connections (Prechelt et al., 1994, sec; 2.7). The algorithm takes an input vector $\vec{x}$ and weight vector $\vec{w}$, and, for the purposes of this chapter, produces a scalar output y. Pruned or layered network architectures are possible by fixing specific weights to zero. Fig. 11.1 illustrates an example network attainable by the algorithm.

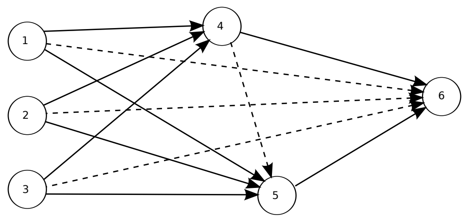


Figure 11.1: An example neural-network architecture obtainable by Alg. 11.1, with three input nodes and one output node. When all graph edges are included, we have a fully connected feed-forward neural network, with all short-cut connections. If the dotted edges are removed (for example by clamping those weights to zero) then a more traditional layered network is obtained, containing a single hidden layer of two nodes. For the purpose of this chapter, only a single output node is required, which in this example is node 6.

Algorithm 11.1 Feed-Forward Dynamics of a Neural Network, Followed by First-Order Error Backpropagation to Calculate $\frac{\partial y}{\partial \vec{x}}$.

```
1: {Feed-forward input vector $\vec{x}$ through network...}
2: $a^0 \leftarrow 1$ {Bias node}
3: $\forall j : 1 \leq j \leq \dim(\vec{x}),\ a^j \leftarrow x^j$ {Input vector, $\vec{x}$}
4: for $j = \dim(\vec{x}) + 1$ to $n$ do
5:     $s^j \leftarrow \sum_{m=0}^{j-1} w^{m,j} a^m$
6:     $a^j \leftarrow g(s^j)$
7: end for
8: $y \leftarrow a^n$ {Network Output}
9: {Backpropagation loop...}
10: for $j = n$ to 1 step $-1$ do
11:     $\delta a^j \leftarrow \begin{cases} 1 & \text{if } j = n \\ \sum_{m=j+1}^{n} (w^{j,m})(\delta s^m) & \text{otherwise} \end{cases}$
12:     $\delta s^j \leftarrow (\delta a^j) g'(s^j)$
13:     $\forall m : 0 \leq m < j,\ \frac{\partial y}{\partial w^{m,j}} \leftarrow (\delta s^j) a^m$
14: end for
15: $\forall j : 1 \leq j \leq \dim(\vec{x}),\ \frac{\partial y}{\partial x^j} \leftarrow \delta a^j$ {Output Vector}
```

In the algorithm, superscripts on variable names indicate the node number, for $j = 0, \ldots, n$, where n is the final node in the network. The variables a^j represent the firing activations of node j, and the final node's activation, a^n, gives the network's output, y. Node 0 is a dedicated "bias" node, with a fixed value of $a^0 = 1$. $\vec{x} = \begin{pmatrix} x^1 & x^2 & \ldots & x^m \end{pmatrix}$ is the external input vector to the whole network, with dimension $m = \dim(\vec{x})$. The notation $w^{k,j}$ is used to indicate the weight within $\vec{w}$ that connects node k to node j. s^j, δa^j and δs^j are workspace scalars for each node.

$g(x) : \mathbb{R} \to \mathbb{R}$ is the "activation function", which is often taken to be $g(x) = \tanh(x)$, or the logistic function, $g(x) = \frac{1}{1+e^{-x}}$. $g'(x)$ and $g''(x)$ are its first and second derivatives.

Lines 10 to 15 of Alg. 11.1 do a backpropagation calculation, which calculates the gradients $\frac{\partial y}{\partial \vec{x}}$ and $\frac{\partial y}{\partial \vec{w}}$. This is the standard backpropagation algorithm for neural networks (Rumelhart et al., 1986; Werbos, 1974) but modified, following Kindermann and Linden (1990), so that the "error function" used is actually the network output y, and also so that the backward pass continues until it has fully generated the quantity $\frac{\partial y}{\partial \vec{x}}$, as required.

11.1.2 Finding Second Derivatives using the R Operator

To find the second-order derivatives that this chapter is aiming to produce, first we note that we can swap the order of differentiation. Hence our desired second derivatives can be written as $\vec{k}^T \frac{\partial}{\partial \vec{x}} \left(\frac{\partial y}{\partial \vec{w}} \right)$ and $\vec{k}^T \frac{\partial}{\partial \vec{x}} \left(\frac{\partial y}{\partial \vec{x}} \right)$, respectively. This trick makes it easy to define a differential operator, R, analogous to that used by Pearlmutter (1994).

Hence we define $R(.)$ as the differential operator

$$R = \sum_i k^i \frac{\partial}{\partial x^i}, \tag{11.4}$$

where k^i is the ith component of $\vec{k}$.

Using this operator, the two second derivatives that we seek can now be written as $R(\frac{\partial y}{\partial \vec{w}})$ and $R(\frac{\partial y}{\partial \vec{x}})$, respectively.

Also, applying this R operator to $\vec{x}$ gives:

$$\begin{aligned} R(\vec{x}) &= \sum_i k^i \frac{\partial}{\partial x^i} \vec{x} \\ &= \vec{k}. \end{aligned} \tag{11.5}$$

As detailed by Pearlmutter (1994), R obeys the usual rules for a differential operator, i.e. it obeys the product rule, the chain rule, the sum rule for derivatives, and differentiating a constant gives zero. For example, the weight vector $\vec{w}$ is a constant with respect to the R operator, thus $R(\vec{w}) = \vec{0}$. Using these rules, and Eq. (11.5), the R operator is applied to each side of each line of Alg. 11.1, to obtain each line of Alg. 11.2, respectively. It should be emphasised that applying the R operator to each line of the algorithm, while obeying the correct rules for differential operators, is calculating the *exact* derivatives that we are seeking. See section 3 of Pearlmutter (1994) for further explanation of the exactness of the R method.

Hence Alg. 11.2 exactly calculates the quantities $R(\frac{\partial y}{\partial \vec{w}})$ and $R(\frac{\partial y}{\partial \vec{x}})$, which are the two second-derivative inner-products that were sought.

Algorithm 11.2 Calculation of the Second-Order Derivatives.

Require: s^j, a^j, δs^j and δa^j values calculated according to Alg. 11.1 for all nodes j, and vector $\vec{k}$ calculated by an appropriate equation (e.g. Eq. (11.2) or Eq. (11.3)).

```
1:  {Forward pass...}
2:  $R(a^0) \leftarrow 0$
3:  $\forall j : 1 \leq j \leq \dim(\vec{x}),\ R(a^j) \leftarrow k^j$
4:  for $j = \dim(\vec{x}) + 1$ to $n$ do
5:      $R(s^j) \leftarrow \sum_{m=0}^{j-1} w^{m,j} R(a^m)$
6:      $R(a^j) \leftarrow g'(s^j) R(s^j)$
7:  end for
8:  {Backward pass...}
9:  for $j = n$ to 1 step $-1$ do
10:     $R(\delta a^j) \leftarrow \begin{cases} 0 & \text{if } j = n \\ \sum_{m=j+1}^{n} (w^{j,m}) R(\delta s^m) & \text{otherwise} \end{cases}$
11:     $R(\delta s^j) \leftarrow R(\delta a^j) g'(s^j) + (\delta a^j) g''(s^j) R(s^j)$
12:     $\forall m : 0 \leq m < j,\ R(\frac{\partial y}{\partial w^{m,j}}) \leftarrow R(\delta s^j)(a^m) + (\delta s^j) R(a^m)$ {Output vector 1: $\vec{k}^T \frac{\partial^2 y}{\partial \vec{x} \partial \vec{w}}$}
13: end for
14: $\forall j : 1 \leq j \leq \dim(\vec{x}),\ R(\frac{\partial y}{\partial x^j}) \leftarrow R(\delta a^j)$ {Output vector 2: $\vec{k}^T \frac{\partial^2 y}{\partial \vec{x} \partial \vec{x}}$}
```

When implementing the algorithm, $R(a^j)$, $R(s^j)$, $R(\delta a^j)$ and $R(\delta s^j)$ are workspace scalars for each node j.

Since both algorithms involve two nested loops that count up to the value of n, the number of nodes in the network, the asymptotic complexity of each algorithm is $O(n^2)$. Since for a fully connected network, we have $\dim(\vec{w}) \approx n^2$, therefore the asymptotic complexity of each algorithm is $O(\dim(\vec{w}))$.

11.1.3 Generating the Full Second-Derivative Matrices

The above algorithm generates the inner products of two second-order derivative matrices with a constant vector. If instead of this inner product, the full second-order derivative matrices are required, then these can be constructed one column at a time. Execution of the above algorithms with $\vec{k}$ equal to the jth Euclidean standard basis vector of dimension $\dim(\vec{x})$, will calculate the jth column of each matrix. Thus accumulating the full second-order matrices, column by column, would take $\mathrm{O}(\dim(\vec{w})\dim(\vec{x}))$ time.

An alternative algorithm to find the full matrix $\frac{\partial^2 y}{\partial \vec{x} \partial \vec{w}}$ is given for a neural network with just one hidden layer by equations 14 and 15 of Prokhorov and Wunsch (1997a), and for a general network by appendix A.4 of Coulom (2002). These also have asymptotic timings of $\mathrm{O}(\dim(\vec{w})\dim(\vec{x}))$.

11.2 Experimental Results

In Section 11.2.1, details are given of how to verify the algorithm's correctness, and in Section 11.2.2, a simple experiment is described that shows how a neural network can be forced to learn a target quantity $\frac{\partial y}{\partial \vec{x}}$. In Section 11.2.3, a VGL experiment is described, comparing the use of a scalar critic to a vector critic.

11.2.1 Numerical Verification of the Algorithm

Numerical differentiation was used to validate the algorithm was calculating the correct second-order gradients. This method provides a flexible and powerful check of the correctness of the program. Since the R method could be applied to other neural-network architectures, it would be advisable to verify any other implementation in a similar manner.

To do the numerical confirmation, first the formation of $\frac{\partial y}{\partial \vec{x}}$ and $\frac{\partial y}{\partial \vec{w}}$ taking place in lines 10 to 15 of Alg. 11.1 was verified, by differentiating y numerically with respect to both $\vec{x}$ and $\vec{w}$, respectively. For example, to verify the first of these, a central-differences numerical derivative was used:

$$\frac{\partial y}{\partial x^i} = \frac{y(\vec{x} + \epsilon \vec{e}_i, \vec{w}) - y(\vec{x} - \epsilon \vec{e}_i, \vec{w})}{2\epsilon} + O(\epsilon^2),$$

where ϵ is a small positive constant, and $\vec{e}_i$ is the ith Euclidean standard basis vector.

Next the second derivatives found by Alg. 11.2 were checked against a first-order numerical differentiation of the (already tested) first-order analytical derivatives found by Alg. 11.1. For example,

$$\vec{k}^T \frac{\partial}{\partial \vec{x}} \left(\frac{\partial y}{\partial \vec{w}} \right) = \frac{\left.\frac{\partial y}{\partial \vec{w}}\right|_{(\vec{x}+\epsilon\vec{k},\vec{w})} - \left.\frac{\partial y}{\partial \vec{w}}\right|_{(\vec{x}-\epsilon\vec{k},\vec{w})}}{2\epsilon} + O(\epsilon^2), \tag{11.6}$$

where $\frac{\partial y}{\partial \vec{w}}$ is calculated by Alg. 11.1 each time it is required in the right hand side of this equation. This check was used to successfully confirm the correctness of Alg. 11.2. For example, using a layered neural network with 4 inputs, three hidden layers of 4 units each, one output layer with 1 unit, hyperbolic tangent activation functions at all nodes, and all components of $\vec{w}$, $\vec{k}$ and $\vec{x}$ randomised uniformly in $[-1, 1]$, the average magnitude of the vector $\vec{k}^T \frac{\partial^2 y}{\partial \vec{x} \partial \vec{w}}$ was found to be 1.4, and the average magnitude of the error in this vector (between its value calculated by Alg. 11.2 and its value calculated by Eq. (11.6)) was 6.4×10^{-10}, when $\epsilon = 10^{-5}$.

11.2.2 Wave-Learning Experiment

This experiment provides an example of how the algorithms of this chapter can be used to adjust the weights of a neural network so as to force the gradient $\frac{\partial y}{\partial \vec{x}}$ to equal a given target value at a given $\vec{x}$.

The objective here is to make a neural network with one input and one output learn the training data given in Table 11.1. In this training data, each row of the table is a different "pattern", p. Each pattern consists of an input value for the neural network (x_p), and target output value (s_p) and a target for the gradient $\frac{\partial y}{\partial \vec{x}}$ (t_p). This experiment is aiming to make a neural network learn two complete periods of a sine wave from just 5 training patterns positioned along the x-axis.

Learning took place by minimising the error function given in Eq. (11.7).

$$E = \frac{1}{2} \sum_{p=1}^{5} \left[\eta_1 \left(y(x_p, w) - s_p \right)^2 + \eta_2 \left(\left.\frac{\partial y}{\partial \vec{x}}\right|_{(x_p,w)} - t_p \right)^2 \right] \tag{11.7}$$

η_1 and η_2 are real constants to weight the relative significance of the two terms in this equation. The first term is an ordinary sum-of-squares error for making the network

x_p (Network input)	s_p (Target for y)	t_p (Target for $\frac{\partial y}{\partial \vec{x}}$)
0.0	0	20
0.25	0	-20
0.5	0	20
0.75	0	-20
1.0	0	20

Table 11.1: Training Data for Wave-Learning Experiment.

output $y(x, \vec{w})$ match the target output for each pattern. The gradient of this part of the error function (with respect to $\vec{w}$) would be found by ordinary backpropagation. The second term of E is a sum-of-squares error term for making the gradient $\frac{\partial y}{\partial \vec{x}}$ match the target gradient t_p for each pattern p. Hence this second term would form the vector $\vec{k}$ described in Eq. (11.3) by

$$\vec{k} = \left.\frac{\partial y}{\partial \vec{x}}\right|_{(x_p,w)} - t_p,$$

for each pattern p, and then the required gradient for learning can be found by Alg. 11.2.

The neural network used was a multilayer perceptron, with two hidden layers of 8 nodes each, and all short-cut connections present between all pairs of non-adjacent layers. The activation function used for all nodes was $g(x) = 1/(1 + e^{-x})$, except for the output node which used $g(x) = x$. Training used gradient descent on Eq. (11.7), with $\eta_1 = \eta_2 = 1$, i.e. with the same significance attached to each component of the error function. Initial network weights were randomised uniformly from [-0.1,0.1].

When training was accelerated through conjugate gradients, with a full line search, learning produced a 100% convergence rate over 100 trials, where convergence was defined as E going below 10^{-5} within 20000 iterations. The output of five typical neural networks trained in this way are shown in Fig. 11.2.

The results show that the problem has been solved correctly, and that it has been possible to make a neural network learn specified target gradients $\frac{\partial y}{\partial \vec{x}}$ at given values of $\vec{x}$. However when experimental parameters were changed, it was observed that the success rate could drop significantly. For example if RPROP was used, and the activation functions used were swapped to $g(x) = \tanh(x)$, then the success rate dropped from 100% to 65%. It seems that it is harder to train a neural network to learn target gradients, $\frac{\partial y}{\partial \vec{x}}$, than target values. By analogy, we can imagine that in trying to make the flexible curves of Fig. 11.2 bend into the shape of a sine wave, it is harder to do so by just twisting the curve at certain points along the x-axis than it is by stretching the curve to pass through given target points.

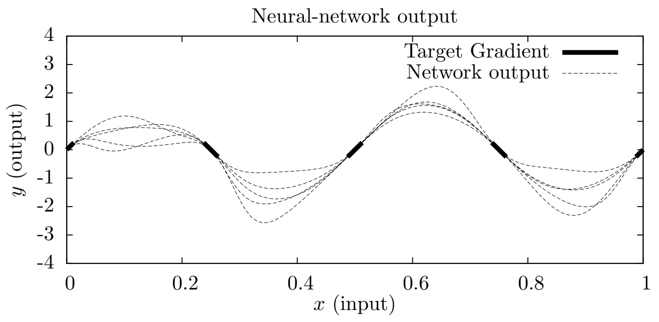


Figure 11.2: Outputs from a sample of five neural networks created to learn the shape of a sinusoidal wave, as detailed in Section 11.2.2. Each dotted curve shows the output from a neural network produced in a different trial. The solid thick lines on the x-axis are designed to give a visual indication of the training point and target gradient. The objective of training is to make the dotted lines run parallel and through the thick lines, which has been achieved well in all of the five network outputs in this figure.

11.2.3 Scalar-Critic VGL Experiment

A VGL experiment, using a scalar critic, is now described, for a variation on the simple quadratic-optimisation experiment previously described in Section 3.7.

The environment is defined with $\vec{x} \equiv x \in \mathbb{R}$ and $\vec{u} \equiv u \in \mathbb{R}$, and model and cost functions:

$$f(x,t,u) := x+u$$
$$U(x,t,u) := 0.$$

Each trajectory is defined to terminate after exactly one time step. A terminal cost defined by $\Phi(x) := (x)^2$ is received at the end of the trajectory. The whole trajectory is parameterised by just x_0 and u_0. The total cost for this trajectory is $(x_0+u_0)^2$, so the optimal action to choose is $u_0 = -x_0$.

The action network $A(\vec{x},\vec{z})$ was a layered neural network with two inputs, one output and one hidden layer of 4 nodes, short-cut connections from the input layer to the output layer, and with activation function $g(x) = \tanh(x)$ on all nodes. The weights $\vec{z}$ were initially randomly chosen from $[-0.1, 0.1]$ uniformly. The critic network $\widetilde{J}(\vec{x}, \vec{w})$ was identically dimensioned to the action network, with a weight vector $\vec{w}$ randomised initially in the same way. The activation function used for the critic was $g(x) = \tanh(x)$ on all nodes except for the output node, which used $g(x) = x$. The input vector to each neural network was (x, t), since in this problem the optimal cost-to-go function depends on both of these inputs.

The VGL(λ) algorithm was used (Alg. 3.2), using learning rate constants $\alpha = 0.1$ and $\beta = 0.1$, and discount factor $\gamma = 1$. Each trajectory started from $x = 0.8$. Experimental results, for both a vector-critic and a scalar critic, averaged over 10 trials are shown in Fig. 11.3. The graph shows that both variants can solve this problem in similar time. The vector-critic's results are slightly faster, but with this being such a simple test problem the slight difference between the two critic-types is not significant.

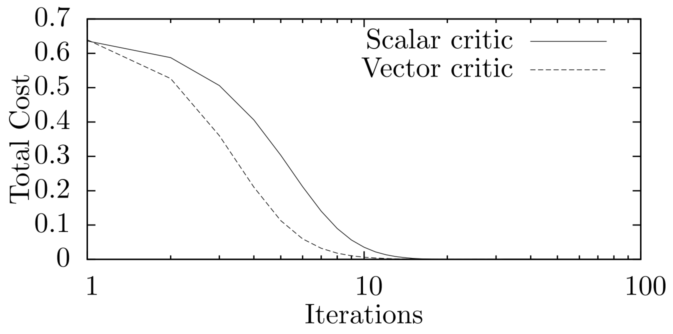


Figure 11.3: Results for Using a Scalar Critic and a Vector Critic to Solve the Problem Described in Section 11.2.3. Both critic types manage to reduce the total cost to almost zero within 100 iterations.

11.3 Chapter Conclusions

A very clear and straightforward algorithm to find the product $\frac{\partial^2 y}{\partial \vec{w} \partial \vec{x}} \vec{k}$ exactly, in time $O(\dim(\vec{w}))$, has been presented. It has been shown that using a forward accumulation of the derivatives leads to an extremely easy way to derive the algorithm, in comparison to a backward accumulation that the GDHP literature generally advocates.

Appropriate modifications to the R method of Pearlmutter (1994) have been made, which enable the algorithm to be derived quickly. Empirical demonstrations have been provided for the problem of learning target gradients in a neural network, and in a simple scalar-critic versus vector-critic VGL comparison. It is the intention of this chapter that implementing VGL with a scalar-critic should be straightforward, and in an efficient ($O(\dim(\vec{w}))$) running time too.

Part IV

Conclusions

Chapter 12

Thesis Conclusions

This thesis has presented a body of work on value-gradient learning. Motivations for using value-gradient learning have been given, in terms of demonstrations of the deficiencies in value-learning. VGL algorithms have been designed to overcome these limitations, and the VGL(λ) and DHP algorithms have been stated in both equation form, and in clear pseudocode form, and empirical results have been provided, in Chapters 3, 5 and 10.

A number of technical issues have been addressed relating to making a complete VGL implementation. These included the efficient computation of a differentiable greedy policy, described in Chapter 5. Another is the problem of clipping, which was described and solved in Chapter 10. And finally, for VGL methods, it is useful to be able to calculate the neural-network second-derivative matrix term, $\frac{\partial^2 \widetilde{J}(\vec{x},\vec{w})}{\partial \vec{w} \partial \vec{x}}$, easily and efficiently. This has been addressed in Chapter 11.

A reflection on the purpose and motivations for this thesis are given in the following subsection (Section 12.1), followed by a high-level description of what the theoretical contributions of this thesis have achieved, in Section 12.2. Finally, pointers for possible future work are given in Section 12.3.

12.1 Purpose and Motivation for Thesis

The purpose of this thesis has been to investigate and develop solutions that effectively solve the ADPRL problem, while following the paradigm of using approximations to Bellman's value function, i.e. approximate dynamic programming (ADP).

The value-function methodology provides a satisfying mathematical structure to the ADPRL problem which other methods such as policy-gradient learning, or genetic algorithms, or exhaustive search in the policy space, do not seem to offer. The intuitive motivation for value-functions is that one can just observe the total or partial trajectory-cost that arrives, and simply by observing (and remembering) those values, one can deduce optimal future behaviour; assuming the learning algorithm eventually converges.

The desire to develop value-gradient learning came from attempting to use TD(λ) to solve an extremely simple control problem, the Vertical Lander, and finding that it could not solve that problem without what seemed like the unreasonable requirement to introduce randomness into the algorithm, to satisfy the need for value exploration. Even when value exploration was successfully introduced, the learning process' convergence was very unreliable and slow, when using a neural-critic.

The thesis gives concrete examples of VL methods converging to suboptimal trajectories in the absence of value exploration (e.g. see Fig. 3.1), and of VL methods diverging (e.g. see Fig. 9.2).

After these experiences, it seemed worthwhile to rethink the approach. The cause of the problem was identified to be that Bellman's principle requires the policy to become greedy, and the greedy policy requires value gradients to be learned. Value-learning methods, without value exploration, were simply not learning the value-gradients; and when value exploration was used to address this problem, value-learning suffered greatly from inefficiency and from stochastic noise corrupting the learning signal.

The most obvious and fastest way to fix this problem was to directly learn the value gradients instead of the values themselves. This produced immediate successful results, by obviating the need for stochastic value exploration (e.g. as shown in Section 3.6), and producing large speed-ups of learning (e.g. as shown in Section 3.7).

It must be acknowledged, again, that Werbos has pioneered all of the main ideas for VGL methods way back since the late 1970s. He understood the motivations for using value-gradients (in the form of DHP/GDHP) to avoid the need for value exploration, and for speeding up learning. Interestingly, he had the foresight to include an "Omega" matrix into the GDHP weight update, although it was slightly troubling as it was previously unknown how to decide what value to assign to it. Happily this has been largely resolved, at least when $\lambda = 1$, by Eq. (3.8) of this thesis. Like Werbos' pioneering work of backpropagation (Werbos, 1974) preceding the neural-network community's general discovery of backpropagation (Rumelhart et al., 1986), from the point of view of myself, his dual-heuristic programming algorithms were very much ahead of their time in ADPRL.

Learning value-gradients requires obtaining an equation for the "target" value gradient, and this comes with the expense of needing to know the model functions, thus losing the model-free aspects of value-learning methodologies. But the requirement to know or learn the model functions does seem like a worthwhile price, as value-gradient methods deliver several potential benefits, including much faster convergence, automatic local value exploration, analytical understanding on what the greedy policy depends upon; and they have ultimately led to a robust, proven-convergent, critic-learning algorithm (VGL$\Omega(1)$), for control problems.

12.2 Theoretical Accomplishments

This discussion gives a very high-level overview of the theoretical achievements of this thesis. Questions are answered relating to "what" a critic needs to learn for optimality, "how" it can do it and "whether" it is guaranteed to work.

The question of "what" a critic needs to know for trajectory optimality has been addressed in Chapter 7, for deterministic environments. This showed that in continuous-valued state spaces, the value-gradient is a necessary artefact that must be learned for trajectory optimality. This proof affects all ADPRL methods that work with a critic in continuous-valued state spaces. In continuous-valued state spaces, it affects VL just as much as it affects VGL. It also shows how VGL methods differ from VL methods. VGL methods are trying to learn a quantity along the sampled trajectory which will make the sampled trajectory locally optimal or extremal. This quantity is the value-gradient. VL methods do not attempt to learn the value-gradient, directly. VL methods will only achieve trajectory optimality by learning the values along many adjacent trajectories, as illustrated in Fig. 3.2. This is required since learning values along multiple adjacent trajectories implicitly learns the value gradients which are necessary for trajectory optimality.

The next question addressed was "how" can a critic function be trained reliably, when function approximation creates difficulties in that bending the critic in one place will inadvertently bend it in other places too; and that learning the critic changes the greedy policy, while changing the policy changes the critic; so how can this process ever be assured to converge? This thesis has given a convergent algorithm for this problem, in Chapter 8. It was quite a shock to discover that, a) the progress measure that displayed monotonic progress was not some sum-of-squares error function on the value-function estimation error, but instead it was J itself; and b) the best convergence result of this thesis comes by proving equivalence to BPTT, a non-critic based ADPRL solution method. The implications of this were discussed further in Section 8.2.4.

The issue of how the critic can be trained reliably also raised the question of "whether" existing algorithms can also do it reliably. The answer to this question was found to be "no" for the algorithms examined, when used in conjunction with a greedy policy. This was proven in Chapter 9, where divergence examples were given for TD(0), TD(1), DHP and VGL(1). Convergence proofs do exist for some critic learning algorithms in the ADPRL literature, but these are limited in that most do not apply to a greedy policy and/or non-linear function approximation.

12.3 Further Future Work

For further work to extend this thesis, a convergent critic-learning algorithm that works with a greedy policy and $\lambda < 1$ is sought. It is likely that the methods of Heydari and Balakrishnan (2011) or Maei et al. (2009) could provide pointers for this.

Another area would be to investigate further how stochastic environments affect VGL(λ) compared to TD(λ). It was interesting that VGL(1) could learn the stochastic task shown in Fig. 3.7, but TD(1) could not learn the corresponding stochastic task shown in Fig. 2.4. Since one of the key motivations for using $\lambda < 1$ in VL methods is to cope with stochastic variance in the learning signal better, it would be worth investigating whether this benefit of low λ carries over to VGL(λ) methods at all. If so, then it motivates the need to find a convergent algorithm for VGL(λ) for $\lambda < 1$; if not, then it strengthens the value of the existing convergence proof for VGLΩ(1).

There was a missing stochastic analysis for the optimality-trajectory proof given Chapter 7. The empirical results in this thesis do show that VGL(λ) works well in stochastic environments, in the absence of value-exploration, but the theoretical case for automatic local value-exploration in stochastic environments needs making properly.

Equivalences have been proven between PGL and BPTT (in Chapter 6,) and also between VGL$\Omega(1)$ and BPTT (in Chapter 8). It would be worth investigating if any model-free stochastic algorithms could form a mean weight update equal to any VGL algorithm; and then the convergence results of VGL$\Omega(1)$ might be able to be transferred over to that model-free algorithm.

Finally, the formula for the Ω_t matrix given in Eq. (3.8) is proven to ensure convergence in the algorithm VGL$\Omega(1)$. It would be interesting to investigate empirically whether this choice of Ω_t can make learning more stable for VGL$\Omega(\lambda)$ for other values of $\lambda < 1$. Preliminary experiments have produced promising results in this direction, but only in the situation where the Ω_t matrix is full rank.

Afterword

Thank you for reading this thesis. The thesis is based upon research that was started in the year 2001 as a solitary research project, and finally completed in 2014, as a Ph.D. thesis. I would be grateful to receive any email to say that the thesis has found some readers other than those related to the examination process.

Michael Fairbank.

February 2014.

References

Asma Al-Tamimi, Frank L. Lewis, and Murad Abu-Khalaf. Discrete-time nonlinear HJB solution using approximate dynamic programming: Convergence proof. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 38(4):943–949, 2008. 18, 160

P.J. Antsaklis. Neural networks for control systems. *IEEE Transactions on Neural Networks*, 1(2):242–244, 1990. 24

Leemon C. Baird. Residual algorithms: Reinforcement learning with function approximation. In *International Conference on Machine Learning*, pages 30–37, 1995. 18, 40, 161, 182

S. N. Balakrishnan and Victor Biega. Adaptive critic based neural networks for control (low order systems applications). *Proceedings 1995 American Control Conference, 335-339, Seattle, WA*, 1995. 15

S. N. Balakrishnan, J. Ding, and F. L. Lewis. Issues on stability of ADP feedback controllers for dynamical systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 38(4):913–917, August 2008. 15

SN Balakrishnan and RD Weil. Neurocontrol: A literature survey. *Mathematical and computer modelling*, 23(1):101–117, 1996. 24

Etienne Barnard. Temporal-difference methods and markov models. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(2):357–365, 1993. 40

A Barto and RH Crites. Improving elevator performance using reinforcement learning. *Advances in neural information processing systems*, 8:1017–1023, 1996. 16

A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:834–846, 1983. 15, 17, 121, 122, 123, 202

Richard E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, USA, 1957. 3, 13, 14, 36, 86, 88

Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1995. xxvii, 49, 208

Rémi Coulom. *Reinforcement Learning Using Neural Networks, with Applications to Motor Control.* PhD thesis, Institut National Polytechnique de Grenoble, 2002. 214

P. Dayan. The convergence of td (λ) for general λ. *Machine learning*, 8(3):341–362, 1992. 18, 39, 41

Kenji Doya. Reinforcement learning in continuous time and space. *Neural Computation*, 12(1):219–245, 2000. 8, 102, 104, 106, 107, 122

Russell Enns and Jennie Si. Helicopter flight-control reconfiguration for main rotor actuator failures. *Journal of guidance, control, and dynamics*, 26(4):572–584, 2003. 15

Michael Fairbank. Neuropilot project, 2001. URL `http://freespace.virgin.net/michael.fairbank/neuropilot`. See sub-page “Neuropilot 1”. This URL was correct during 2013. 106

Michael Fairbank. Reinforcement learning of value gradients. Presented at the Gatsby Computational Neuroscience Unit, London, April 2002. 26, 53

Michael Fairbank. Reinforcement learning by value gradients. *CoRR*, abs/0803.3539, 2008. URL `http://arxiv.org/abs/0803.3539`. 17, 25, 26, 47, 53, 68, 112, 161

Michael Fairbank. The importance of clipping in neurocontrol by direct gradient descent on the cost-to-go function and in adaptive dynamic programming. *CoRR*, abs/1302.5565, 2013. URL `http://arxiv.org/abs/1302.5565`. 25

Michael Fairbank and Eduardo Alonso. The divergence of reinforcement learning algorithms with value-iteration and function approximation. *eprint arXiv:1107.4606*, 2011a. 181

Michael Fairbank and Eduardo Alonso. The local optimality of reinforcement learning by value gradients, and its relationship to policy gradient learning. *CoRR*, abs/1101.0428, 2011b. URL `http://arxiv.org/abs/1101.0428`. 25

Michael Fairbank and Eduardo Alonso. A comparison of learning speed and ability to cope without exploration between DHP and TD(0). In *Proceedings of the IEEE International Joint Conference on Neural Networks 2012 (IJCNN'12)*, pages 1478–1485. IEEE Press, June 2012a. 25

Michael Fairbank and Eduardo Alonso. The divergence of reinforcement learning algorithms with value-iteration and function approximation. In *Proceedings of the IEEE International Joint Conference on Neural Networks 2012 (IJCNN'12)*, pages 3070–3077. IEEE Press, June 2012b. 25, 171

Michael Fairbank and Eduardo Alonso. Value-gradient learning. In *Proceedings of the IEEE International Joint Conference on Neural Networks 2012 (IJCNN'12)*, pages 3062–3069. IEEE Press, June 2012c. 25, 62

Michael Fairbank, Eduardo Alonso, and Danil Prokhorov. Simple and fast calculation of the second-order gradients for globalized dual heuristic dynamic programming in neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 23 (10):1671–1678, October 2012a. 25

Michael Fairbank, Danil Prokhorov, and Eduardo Alonso. Approximating optimal control with value gradient learning. In Frank Lewis and Derong Liu, editors, *Reinforcement Learning and Approximate Dynamic Programming for Feedback Control*, pages 142–161. Wiley-IEEE Press, New York, 2012b. 25, 166

Michael Fairbank, Eduardo Alonso, and Danil Prokhorov. An equivalence between adaptive dynamic programming with a critic and backpropagation through time. *IEEE Transactions on Neural Networks and Learning Systems*, 24(12):2088–2100, December 2013. 25

Michael Fairbank, Shuhui Li, Xingang Fu, Eduardo Alonso, and Donald Wunsch. An adaptive recurrent neural-network controller using a stabilization matrix and predictive inputs to solve a tracking problem under disturbances. *Neural Networks*, 49(0):74 – 86, 2014a. ISSN 0893-6080. doi: http://dx.doi.org/10.1016/j.neunet.2013.09.010. URL `http://www.sciencedirect.com/science/article/pii/S0893608013002438`. 167

Michael Fairbank, Danil Prokhorov, and Eduardo Alonso. Clipping in neurocontrol by adaptive dynamic programming. *IEEE Transactions on Neural Networks and Learning Systems*, 2014b. doi: 10.1109/TNNLS.2014.2297991. In Press. 25

Silvia Ferrari and Robert F Stengel. An adaptive critic global controller. In *Proceedings of the 2002 American Control Conference*, volume 4, pages 2665–2670. IEEE, 2002. 15

Silvia Ferrari and Robert F. Stengel. Model-based adaptive critic designs. In J. Si, A.G. Barto, W.B. Powell, and D.C. Wunsch, editors, *Handbook of learning and approximate dynamic programming*, pages 65–96. Wiley-IEEE Press, New York, 2004. 18, 102, 160, 170

Razvan V. Florian. Correct equations for the dynamics of the cart-pole system. Technical report, Center for Cognitive and Neural Studies (Coneural), Str. Saturn 24, 400504 Cluj-Napoca, Romania, 2007. 121

V. Gullapalli. A stochastic reinforcement learning algorithm for learning real-valued functions. *Neural Networks*, 3:671–692, 1990. 102

M.T. Hagan and H.B. Demuth. Neural networks for control. In *American Control Conference, 1999. Proceedings of the 1999*, volume 3, pages 1642–1656. IEEE, 1999. 24

D. Han and S. N. Balakrishnan. Adaptive critic based neural networks for control-constrained agile missile control. In *Proceedings of the 1999 American Control Conference.*, volume 4, pages 2600–2604. IEEE, June 1999. 15

Dongchen Han and SN Balakrishnan. State-constrained agile missile control with adaptive-critic-based neural networks. *Control Systems Technology, IEEE Transactions on*, 10(4):481–489, 2002. 15

Ali Heydari and S N Balakrishnan. Finite-horizon input-constrained nonlinear optimal control using single network adaptive critics. *American Control Conference ACC*, pages 3047–3052, 2011. 18, 104, 114, 160, 161, 226

Ronald A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA, 1960a. 8, 14

Ronald A. Howard. *Dynamic Programming and Markov Processes*, chapter 4, pages 42–43. MIT Press, Cambridge, MA, 1960b. 18, 160

D. H. Jacobson and D. Q. Mayne. *Differential Dynamic Programming*. Elsevier, New York, NY, 1970. 13, 24, 88

L.P. Kaelbling, M.L. Littman, and A.W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996. 15, 19

Sham Kakade. A natural policy gradient. In *NIPS*, volume 14, pages 1531–1538, 2001. 8, 44

J. Kindermann and A. Linden. Inversion of neural networks by gradient descent. *Parallel Computing*, 14:277–286, 1990. 212

D. E. Kirk. *Optimal control theory: an introduction*. Dover Publications, 2004. 23

G. G. Lendaris, L. Schultz, and T. Shannon. Adaptive critic design for intelligent steering and speed control of a 2-axle vehicle. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks, 2000.*, volume 3, pages 73–78. IEEE, July 2000. 16

George G. Lendaris and James C. Neidhoefer. Guidance in the use of adaptive critics for control. In J. Si, A.G. Barto, W.B. Powell, and D.C. Wunsch, editors, *Handbook of learning and approximate dynamic programming*, pages 97–124. Wiley-IEEE Press, New York, 2004. 21

George G. Lendaris and Christian Paintz. Training strategies for critic and action neural networks in dual heuristic programming method. In *Proceedings of International Conference on Neural Networks, Houston*, 1997. 121, 122, 123, 208

G.G. Lendaris. A retrospective on adaptive dynamic programming for control. In *International Joint Conference on Neural Networks, 2009. IJCNN 2009.*, pages 1750–1757. IEEE, 2009. 15

F.L. Lewis and D. Vrabie. Reinforcement learning and adaptive dynamic programming for feedback control. *Circuits and Systems Magazine, IEEE*, 9(3):32–50, 2009. 15

Frank L Lewis, Draguna Vrabie, and Kyriakos G Vamvoudakis. Reinforcement learning and feedback control: Using natural decision methods to design optimal adaptive controllers. *Control Systems, IEEE*, 32(6):76–105, 2012. 15

Shuhui Li, Michael Fairbank, Donald Wunsch, and Eduardo Alonso. Vector control of a grid-connected rectifier inverter using an artificial neural network. In *Proceedings of the IEEE International Joint Conference on Neural Networks 2012 (IJCNN'12)*, pages 1783–1789. IEEE Press, June 2012. 16

Chao Lu, Jennie Si, Xiaorong Xie, and Lei Yang. Direct neural dynamic programming method for power system stability enhancement. In *Intelligent Systems Application to Power Systems, 2005. Proceedings of the 13th International Conference on*, pages 128–135. IEEE, 2005. 16

Hamid Maei, Csaba Szepesvári, Shalabh Bhatnager, Doina Precup, David Silver, and Richard Sutton. Convergent temporal-difference learning with arbitrary smooth function approximation. In *Advances in Neural Information Processing Systems (NIPS'09)*. MIT Press, 2009. 18, 41, 169, 226

Hamid Reza Maei and Richard S Sutton. GQ(λ): A general gradient algorithm for temporal-difference prediction learning with eligibility traces. In *Proceedings of the Third Conference on Artificial General Intelligence*, volume 1, pages 91–96, 2010. 27

Hamid Reza Maei, Csaba Szepesvári, Shalabh Bhatnagar, and Richard S Sutton. Toward off-policy learning control with function approximation. In *Proceedings of the 27th International Conference on Machine Learning*, pages 719–726, 2010. 44

Marvin Minsky. *Computers and Thought*. New York: McGraw-Hill, 1963. 14

Martin Møller. Exact calculation of the product of the hessian matrix of feed-forward network error functions and a vector in O(N) time. Technical Report DAIMI PB-432, Aarhus University, 1993. 209

Remi Munos. Policy gradient in continuous time. *Journal of Machine Learning Research*, 7:413–427, 2006. 59, 189

John J Murray, Chadwick J Cox, George G Lendaris, and Richard Saeks. Adaptive dynamic programming. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 32(2):140–153, 2002. 15

Andrew Y. Ng, H. Jin Kim, Michael I. Jordan, and Shankar Sastry. Inverted autonomous helicopter flight via reinforcement learning. In *International Symposium on Experimental Robotics*. MIT Press, 2004. 16, 21, 59

Barak A. Pearlmutter. Fast exact multiplication by the Hessian. *Neural Computation*, 6(1):147–160, 1994. 209, 212, 213, 218

Jan Peters and Stefan Schaal. Policy gradient methods for robotics. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2006. 137, 190

L. S. Pontryagin, V. G. Boltayanskii, R. V. Gamkrelidze, and E. F. Mishchenko. *The Mathematical Theory of Optimal Processes (Translated from Russian)*, volume 4. Wiley, 1962. ISBN 2-88124-077-1. 23, 88, 90

W.B. Powell. What you should know about approximate dynamic programming. *Naval Research Logistics (NRL)*, 56(3):239–249, 2009. 15

Lutz Prechelt et al. Proben1: A set of neural network benchmark problems and benchmarking rules. *Fakultät für Informatik, Univ. Karlsruhe, Karlsruhe, Germany, Tech. Rep*, 21:94, 1994. 210

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in C (2nd ed.): the art of scientific computing*. Cambridge University Press, New York, NY, USA, 1992. ISBN 0-521-43108-5. 14

Danil Prokhorov. *Computational intelligence in automotive applications*, volume 132. Springer, 2009. 15

Danil Prokhorov and Don Wunsch. Adaptive critic designs. *IEEE Transactions on Neural Networks*, 8(5):997–1007, 1997a. 15, 18, 31, 60, 64, 65, 101, 102, 208, 214

Danil V. Prokhorov and Donald C. Wunsch. Convergence of critic-based training. In *in Proc. IEEE Int. Conf. Syst*, pages 3057–3060, 1997b. 18, 161

Danil V Prokhorov, Roberto A Santiago, and Donald C Wunsch. Adaptive critic designs: A case study for neurocontrol. *Neural Networks*, 8(9):1367–1372, 1995. 15

L. B. Rall. Automatic differentiation: Techniques and applications. *Lecture Notes in Computer Science*, 120, 1981. 67, 209

Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Proc. of the IEEE Intl. Conf. on Neural Networks*, pages 586–591, San Francisco, CA, 1993. 120

David E Rumelhart, Geoffrey E Hintont, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986. 67, 212, 225

G. Rummery and M. Niranjan. On-line q-learning using connectionist systems. *Tech. Rep. Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department*, 1994. 171

AL Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959. 14

Arthur L Samuel. Some studies in machine learning using the game of checkers. iirecent progress. *IBM Journal of research and development*, 11(6):601–617, 1967. 14

David Silver, Richard S Sutton, and Martin Müller. Temporal-difference search in computer go. *Machine learning*, pages 1–37, 2012. 16, 23

L. Sonneborn and F. Van Vleck. The bang-bang principle for linear control systems. *SIAM J. Control*, 2:152–159, 1965. 156

Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988. 5, 15, 17, 18, 30, 31, 32, 39, 75, 169

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, Massachussetts, USA, 1998. 4, 14, 15, 19, 20, 21, 33, 34, 37, 38, 42, 53, 54, 102

Richard S. Sutton, David Mcallester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12*, volume 12, pages 1057–1063, 2000. 8, 18, 42, 43, 44

Richard S Sutton, Satinder Singh, and David McAllester. Comparing policy-gradient algorithms. Available from author's homepage, 2001. 8, 44

Richard S. Sutton, Hamid Reza Maei, Doina Precup, Shalabh Bhatnagar, David Silver, Csaba Szepesvári, and Eric Wiewiora. Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 993–1000, New York, NY, USA, 2009. ACM. 18, 41, 169

Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994. 16

E. Todorov. Optimal control theory. *Bayesian brain: probabilistic approaches to neural coding*, pages 269–298, 2006. 23, 89, 90, 115

John N. Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. Technical report, IEEE Transactions on Automatic Control, 1996a. 18, 40, 169, 182

John N. Tsitsiklis and Benjamin Van Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, 22(1-3):59–94, 1996b. 182

Ganesh Kumar Venayagamoorthy and Donald C. Wunsch. Dual heuristic programming excitation neurocontrol for generators in a multimachine power system. *IEEE Transactions on Industry Applications*, 39:382–394, 2003. 16, 60, 208

Ganesh Kumar Venayagamoorthy, Ronald G. Harley, and Donald C. Wunsch. Comparison of heuristic dynamic programming and dual heuristic programming adaptive critics for neurocontrol of a turbogenerator. *IEEE Transactions on Neural Networks*, 13(3):764–773, 2002. 208

Fei-Yue Wang, Huaguang Zhang, and Derong Liu. Adaptive dynamic programming: An introduction. *IEEE Computational Intelligence Magazine*, 4(2):39–47, 2009. 15, 60, 64, 115, 208

Christopher J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, 1989. 33, 37, 53

P. J. Werbos. Building and understanding adaptive systems: a statistical/numerical approach to factory automation and brain research. *IEEE Trans. Syst. Man Cybern.*, 17(1):7–20, January 1987. ISSN 0018-9472. 68, 208

Paul Werbos. Reinforcement learning and approximate dynamic programming (RLADP)–foundations, common misconceptions, and the challenges ahead. In Frank Lewis and Derong Liu, editors, *Reinforcement Learning and Approximate Dynamic Programming for Feedback Control*, pages 3–30. Wiley-IEEE Press, New York, 2012. 13

Paul J. Werbos. The elements of intelligence. In *Namur, Belgium: Cybernetica*, number 3, 1968. 15

Paul J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974. 67, 209, 212, 225

Paul J. Werbos. Advanced forecasting for global crisis warning and models of intelligence. In *General Systems Yearbook*, 1977. 15

Paul J. Werbos. Applications of advances in nonlinear sensitivity analysis. In R. Drenick and R. Kozin, editors, *System Modeling and Optimization: Proceedings of the 10th IFIP Conference*. Springer New York, 1982, 1981. 15

Paul J Werbos. Neural networks for control and system identification. In *Proceedings of the 28th IEEE Conference on Decision and Control, 1989.*, pages 260–265. IEEE, 1989. 37

Paul J. Werbos. Backpropagation through time: What it does and how to do it. In *Proceedings of the IEEE*, volume 78, No. 10, pages 1550–1560, 1990a. 18, 123, 130

Paul J Werbos. A menu of designs for reinforcement learning over time. *Neural networks for control*, pages 67–95, 1990b. 6

Paul J. Werbos. Consistency of HDP applied to a simple reinforcement learning problem. *Neural Networks*, 3:179–189, 1990c. 41

Paul J. Werbos. Neurocontrol: Where it is going and why it is crucial. In I. Aleksander and J. Taylor, editors, *Artifical Neural Networks II (ICANN-2), North Holland, Amsterdam*, pages 61–68, 1992a. 24

Paul J. Werbos. Approximating dynamic programming for real-time control and neural modeling. In David A. White and Donald A. Sofge, editors, *Handbook of Intelligent Control*, chapter 13, pages 493–525. Van Nostrand Reinhold, New York, 1992b. 6, 15, 55, 64, 65

Paul J. Werbos. Stable adaptive control using new critic designs. *eprint arXiv:adap-org/9810001*, 1998. 15, 18, 41, 68, 161, 170

Paul J. Werbos. ADP: Goals, opportunities and principles. In J. Si, A.G. Barto, W.B. Powell, and D.C. Wunsch, editors, *Handbook of learning and approximate dynamic programming*, pages 3–44. Wiley-IEEE Press, New York, 2004. 15, 38, 69

Paul J. Werbos. Backwards differentiation in AD and neural nets: Past links and new opportunities. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, Lecture Notes in Computational Science and Engineering, pages 15–34. Springer, 2005. 67, 209

Paul J. Werbos. Using ADP to understand and replicate brain intelligence: the next level design. In *IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pages 209 – 216, 2007. 6

Paul J. Werbos. Foreword ADP: The key direction for future research in intelligent control and understanding brain intelligence. *IEEE Transactions on Systems, Man and Cybernetics-Part B: Cybernetics*, 38(4), 2008. 4

Paul J Werbos, Thomas McAvoy, and Ted Su. Neural networks, system identification, and control in the chemical process industries. In David A. White and Donald A. Sofge, editors, *Handbook of Intelligent Control*, chapter 10, pages 283–356. Van Nostrand Reinhold, New York, 1992. 59, 208

B. Widrow, N.K. Gupta, and S. Maitra. Punish/reward: Learning with a critic in adaptive threshold systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 3(5):455–465, September 1974. 15, 17

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–356, 1992. 18, 130, 136, 138, 139, 190

Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1:270–280, June 1989. ISSN 0899-7667. 133