Recurrent Neural Networks and Reinforcement Learning: Dynamic Approaches

Corentin Tallec

université PARIS-SACLAY

UNIVERSITÉ PARIS SUD
Comprendre le monde, construire l'avenir

NNT : 2019SACLS360

Thèse de doctorat

Réseaux Récurrents et Apprentissage par Renforcement: Approches Dynamiques

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université Paris-Sud

École doctorale n°580 Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Gif-sur-Yvette, le 7 Octobre 2019, par

CORENTIN TALLEC

Composition du Jury :

Anne Vilnat Professeur, Université Paris-Sud	Président
Joan Bruna Associate Professor, Courant Institute of Mathematical Sciences and Center for Data Science, NYU Department of Computer Science, Department of Mathematics	Rapporteur
Pascal Vincent Associate Professor, Université de Montréal	Rapporteur
Francis Bach Directeur de Recherche, Ecole Normale Superieure (Departement d'Informatique)	Examinateur
Jean-Philippe Vert Research professor, Mines ParisTech.	Examinateur
Yann Ollivier Research Scientist, Facebook AI Research	Directeur de thèse

Résumé

D'un agent intelligent plongé dans le monde, nous attendons à la fois qu'il comprenne, et interagisse avec son environement. La compréhension du monde environnant requiert typiquement l'assimilation de séquences de stimulations sensorielles diverses. Interagir avec l'environnement requiert d'être capable d'adapter son comportement dans le but d'atteindre un objectif fixé, ou de maximiser une notion de récompense. Cette vision bipartite de l'interaction agent-environnement motive les deux parties de cette thèse : les réseaux de neurone récurrents sont des outils puissants pour traiter des signaux multimodaux, comme ceux résultants de l'interaction d'un agent avec son environnement, et l'apprentissage par renforcement et le domaine privilégié pour orienter le comportement d'un agent en direction d'un but. Cette thèse a pour but d'apporter des contributions théoriques et pratiques dans ces deux champs. Dans le domaine des réseaux récurrents, les contributions de cette thèse sont doubles : nous introduisons deux nouveaux algorithmes d'apprentissage de réseaux récurrents en ligne, théoriquement fondés, et passant à l'échelle. Par ailleurs, nous approfondissons les connaissances sur les réseaux récurrents à portes, en analysant leurs propriétés d'invariance. Dans le domaine de l'apprentissage par renforcement, notre contribution principale est de proposer une méthode pour robustifier les algorithmes existant par rapport à la discrétisation temporelle. Toutes ces contributions sont motivées théoriquements, et soutenues par des éléments expérimentaux.

Abstract

An intelligent agent immerged in its environment must be able to both understand and interact with the world. Understanding the environment requires processing sequences of sensorial inputs. Interacting with the environment typically involves issuing actions, and adapting those actions to strive towards a given goal, or to maximize a notion of reward. This view of a two parts agent-environment interaction motivates the two parts of this thesis: recurrent neural networks are powerful tools to make sense of complex and diverse sequences of inputs, such as those resulting from an agent-environment interaction; reinforcement learning is the field of choice to direct the behavior of an agent towards a goal. This thesis aim is to provide theoretical and practical insights in those two domains. In the field of recurrent networks, this thesis contribution is twofold: we introduce two new, theoretically grounded and scalable learning algorithms that can be used online. Besides, we advance understanding of gated recurrent networks, by examining their invariance properties. In the field of reinforcement learning, our main contribution is to provide guidelines to design time discretization robust algorithms. All these contributions are theoretically grounded, and backed up by experimental results.

Acknowledgements

I would like first and foremost to thank Yann Ollivier, for his guidance, support and supervision. He has been and still remains a constant source of inspiration in his approach to scientific research, as well as an inexhaustible fount of knowledge and original ideas. I would also like to thank Thomas, Léonard and Diviyan my co-authors, as well as all the members of the TAckling the Underspecified team for making my work environment as pleasant and cheerful as one could hope for during these three years. A special thanks to Léonard for proofreading this thesis.

Most of all, I would like to thank my parents, as well as my younger brother for their constant support and encouragement throughout my Ph.D.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An intelligent agent should be able to both understand and interact with the world. Such an agent could typically be implemented using a recurrent network to understand the flow of inputs provided by the world, and reinforcement learning, potentially making use of the built model, to interact with it. The aim of this thesis is to highlight, and when possible, solve some of the inherent problems to building such an agent.

To the agent, the world is a nearly infinite stream of sensory inputs. *Recurrent neural networks* (RNNs) are flexible function approximators that can process and extract signal from any kind of sequential data. This makes them a good fit to learn from the diverse stream of data coming from multi-sensorial captors that we would typically want to integrate in an intelligent agent. However, RNN typically lack two essential components to directly be applied in a black box fashion on diverse sensorial streams:

- They are hard to train on streaming data. Theoretically grounded learning methods for recurrent networks either require storing all inputs and intermediate recurrent activations, or suffer from high computational burden. While storing inputs and intermediate computations is feasible for fixed size short length sequences, this hardly scale to streaming sensorial data, where the number of datapoint can grow arbitrarily large.

- They exhibit difficulties in learning long term dependencies, in term of number of steps. This is problematic: the typical number of steps on which relevant dependencies occur increases when the quality of sensors increases, or equivalently when the sampling rate increases. This means that algorithms are likely to perform worse when hardware improves.

Reinforcement learning algorithms are very general methods to solve goal based problem and can be used to learn the interactive part of the agent. However, they suffer from a number of problems, and notably lack robustness. Importantly for our purpose, reinforcement learning turns out to be non robust to changes in time discretization, or equivalently, to changes in framerate: with standard algorithms, a decrease in time discretization leads to poorer performance. As for recurrent network training, a decrease of performance following a hardware improvment should be avoided at all cost.

1.1 Contributions

Learning algorithms for recurrent neural networks typically work in an offline fashion; they require a fixed dataset of finite, typically small length, sequences to learn. The most well-known, theoretically grounded algorithm that can deal with streaming data is *Real Time Recurrent Learning* [Williams and Zipser, 1989], but it is prohibitively expensive to use with big networks. We introduce *Unbiased Online Recurrent Optimization* (UORO) and *Anticipated Reweighted Truncated Backpropagation* (ARTBP) two scalable alternatives to RTRL, able to process streaming data, theoretically grounded, based on unbiased estimations of the gradient.

The former, UORO, uses forward mode differentiation, much like RTRL. The computational complexity of RTRL is related to the need to maintain and update online the derivative of the recurrent network hidden state w.r.t. its parameters. This makes RTRL unusable for reasonably large networks. In contrast UORO only maintains a rank one unbiased approximation of the derivative of the hidden state w.r.t. to the parameter. It can be shown that, by carefully chosing the form and update of this rank one approximation, one can maintain the unbiasedness property through time, and obtain a gradient estimate whose variance remains bounded, making it an elligible direction to perform gradient descent.

The latter, ARTBP, relies on backward mode differentiation, also referred to as backpropagation. The problem with backpropagation in recurrent network is that obtaining an exact gradient estimate using backward mode differentiation at each time step requires performing as many backward computations as the number of timesteps that were previously processed by the network. This notably means that, when faced with infinite data streams, training algorithms have to keep in memory all the data that were previously encountered, imposing an ever increasing memory demand. The standard solution to this memory problem, *Truncated Backpropagation Through Time*, cuts backward computations after a fixed, finite amount of timesteps. While practically effective the resulting algorithm provides biased gradient estimates, and has no theoretical guarantees. In contrast, ARTBP samples randomized truncation lengths, and modifies the backward computations, to account for the overepresentation of short backpropagation paths. With proper reweighting, one can obtain unbiased gradient estimates.

Difficulties in learning long term dependencies is often explained as an effect of vanishing gradients. However, this explanation gives relatively few insights on how to design networks able to handle specific ranges of dependencies. We provide an analysis of recurrent networks in terms of invariance to time transformation. This analysis gives information on how to initialize recurrent networks to bias learning towards specific ranges of time dependencies. More precisely, we show that the class of standard recurrent networks is not invariant to time transformations; when considering a data stream with two different time discretizations δt_0 and δt_1, and a standard recurrent network processing data with δt_0, it is not easily possible to find a standard recurrent network replicating the behavior of the first network, with time discretization δt_1. We further exhibit recurrent architectures that are invariant to time discretization, and relate the property

of being time discretization invariant to the use of a learnable forget gate mechanism, that is directly used to learn the time discretization used to process the data. Given this new insight on forget gates, we provide a mean to initialize forget gate biases to improve learning when the range of temporal dependencies of interest is known a priori.

While reinforcement learning algorithms can be quite powerful and general, they have been shown to suffer from brittleness, and their results can be difficult to reproduce [Henderson et al., 2017, Zhang et al., 2018]. We reproduced [Ha and Schmidhuber, 2018], one of the first approach to both simply and efficiently combine a recurrent model of the world with a simple evolution based control algorithm. We showed that results were reproducible relatively easily, even though training of the recurrent model only seems to moderately improve results.

Robustness is critical to designing all purpose reinforcement learning algorithms. We show theoretically and empirically that standard Q-learning based RL algorithms are not robust to changes in time discretization. We design an algorithm that could theoretically handle arbitrarily small time discretization, and show that this algorithm displays better robustness properties than standard algorithms. In details, two components of standard Q-learning critically fail when the time discretization becomes small. First, the Q-function, which is the object of interest in Q-learning, collapses to the V-function when approaching the continuous time limit. This notably means that it does not provide relevant information regarding the hierarchy of actions, and can thus not be used to perform policy improvement. Second, standard exploration methods, such as ε-greedy exploration, do not explore in continuous time, and thus fail to provide the off-policyness required by Q-learning to learn an optimal policy. We show that, by learning a properly rescaled version of the advantage function and by introducing temporally coherent off-policy exploration, both limitations can be overcome, resulting in a time-discretization robust algorithm.

We further mention [Lucas et al., 2018], a piece of research, related to generative models, that was performed during the thesis but does not directly relate to its main topic. [Lucas et al., 2018] introduces a mechanism to combat mode collapse in *Generative Adversarial Network* (GAN) training. In standard GAN training, a discriminator tries to distinguish between real data and data produced by a generator network, while the generator network tries to fool the discriminator. While such training procedure can lead to the generation of uncannily realistic new data samples, it is often faced with a *mode dropping* problem, where the generator only generates sample from a subpart of the initial data distribution. We relate this problem to the fact that the discriminator only discriminates at the level of individual samples, and not at the level of the distribution, making the task of detecting artifacts in individual images easy, but the task of detecting mismatches in the distribution global statistics hard. To remedy this problem, we introduce a batchwise discriminator, using a batch permutation invariant architecture to enable the computation of arbitrary batch statistics, and introduce a batch level loss, to allow the discriminator to discriminate using batch statistics, which are proxies to distribution statistics.

1.2 Overview

This thesis is divided into two parts. The first part deals with recurrent neural networks while the second part deals with reinforcement learning. Each of those two parts begin with a general introduction to the topic, as well as a litterature review.

Chapter 2 introduces recurrent neural network architecture, as well as usual training algorithms, main problematics, notably the problem of capturing long term dependencies, related to the famous vanishing gradient problem, and standard approaches to solve these problems. Chapters 3,4 and 5 introduce our contributions in the field of RNNS, with Chapters 3 and 4 presenting new online recurrent learning algorithms, and 5 providing new insights on gated recurrent networks. Chapter 6 presents a short overview of the reinforcement learning field, describing the core principles, as well as usual methods and related state of the art. Chapter 7 describes a reproductive work of a model based reinforcement learning approach, based on [Ha and Schmidhuber, 2018]. Chapter 8 introduces a study of Q-learning based methods in terms of resilience to time discretization, and presents a robust Q-learning variant. Finally, Chapter 9 concludes this thesis, and provides directions for future works.

Chapter 2

Recurrent neural networks

This chapter provides background and litterature for recurrent neural networks, detailing the key elements that are of interest in the rest of the thesis. Section 2.1 introduces the concept or neural networks and recurrent architectures. Section 2.2 discusses usual learning algorithms for recurrent architectures, as well as their pitfalls. Section 2.3 focuses on long term dependencies learning, the related vanishing gradient problem and introduces usual answers to this problem.

2.1 From neural networks to recurrent neural networks

In informal terms, neural networks are parametric function approximators, comprised of simple layers (linear transformations, pointwise nonlinearities, ...), combined with one another to form more complex modules. The standard, and probably simplest form of multi-layer neural network is the multi-layer perceptron, which stacks linear layers, separated by non linearities. It can be shown that, with a proper choice of non linearity and unconstrained layer widths a multi-layer perceptron with more than one layer is a universal function approximator. An appealing aspect of neural networks is that the gradient of any neuron, or any intermediate quantity, w.r.t. any parameter or neuron is efficiently computed using the gradient backpropagation algorithm, which is a clever application of the chain rule [LeCun et al., 1999].

For our purpose, neural networks are simply going to be considered as parametric function approximators, whose error derivative can be computed efficiently via backpropagation (i.e. with a time complexity similar to that of their forward pass). Formally, we define a neural network F as a parametric function

$$y = F(x, \theta) \tag{2.1}$$

where y, x and θ belong in real vector space, such that, for any δy of the same size as y, $\delta y^T \partial_\theta F(x, \theta)$ and $\delta y^T \partial_x F(x, \theta)$ exist and are computable in a time complexity similar to the time complexity of computing $F(x, \theta)$. What this tells us is that if we define a

prediction loss on the output of the neural network (typically a mean squared error)

$$\ell = \frac{1}{2}\|\hat{y} - F(x,\theta)\|^2 \tag{2.2}$$

then obtaining the gradient of ℓ w.r.t. either θ or x is doable in a reasonable time, since

$$\frac{\partial \ell}{\partial \theta} = (\hat{y} - F(x,\theta))^T \frac{\partial F}{\partial \theta}(x,\theta). \tag{2.3}$$

Multi-layer perceptrons obviously belong in this category of functional approximators. More interestingly, if we consider a Directed Acyclic Graph where each node belongs to this class of function approximation, and edges correspond to output forwarding, then the new function approximation represented by this DAG also belongs to this class.

Standard non recurrent neural networks are typically used to deal with i.i.d. datasets, where each data sample does not depend on the next or previous datapoint. When dealing with sequential datasets, e.g. where datapoints are temporally correlated, basic neural networks, such as vanilla multi-layer perceptrons, are not equipped to process temporal dependencies. Given a sequential dataset, i.e. a sequence of datapoints $(x_t, y_t)_{t\geq 0}$, where the x's are inputs, the y's are outputs to be predicted, and y_t can depend on all previous inputs, i.e. on all the $(x_{t'})_{t'\leq t}$, a simple neural network taking the x's as input one at a time will only be able to capture the dependency of y_t on x_t. Several strategies exist to circumvent this limitation. A straightforward strategy is to predict each y_t feeding as input to the network a sliding window $x_{t-K}, ..., x_t$ of input datapoints, where K is the *fixed* sliding window size. Such a strategy is however still limited to only learning temporal dependencies up to K steps in the past.

Recurrent neural networks, on the other hand, overcome the fixed temporal horizon problem by providing a standard neural network with a learnable *memory* component, in the form of a *recurrent state*. A recurrent network can then both store useful information from the datasequence in its memory component, and use this memory component to predict targets. Formally, we define a recurrent neural network as a dynamical system

$$s_{t+1} = F_{\text{state}}(x_{t+1}, s_t, \theta) \tag{2.4}$$

$$o_{t+1} = F_{\text{output}}(x_{t+1}, s_t, \phi), \tag{2.5}$$

where the s's are recurrent states, the o's are the network outputs, and θ and ϕ are trainable parameters. Schematically, Figure 2.1 shows a simplified view of a recurrent network. Note that F_{state} and F_{output} can be arbitrarily complex, and are typically not simple linear layers.

Alternatively, one can define a RNN dynamic through a single recurrent equation, that encompasses both states and outputs evolution

$$\tilde{s}_{t+1} = F(x_{t+1}, \tilde{s}_t, \Theta) \tag{2.6}$$

where $\tilde{s}_t = (s_t, o_t)$, $\Theta = (\theta, \phi)$ and F translates the effect of F_{state} and F_{output} on $\tilde{s}$. This formulation typically makes backpropagation equations simpler, and will be used for most subsequent derivations.

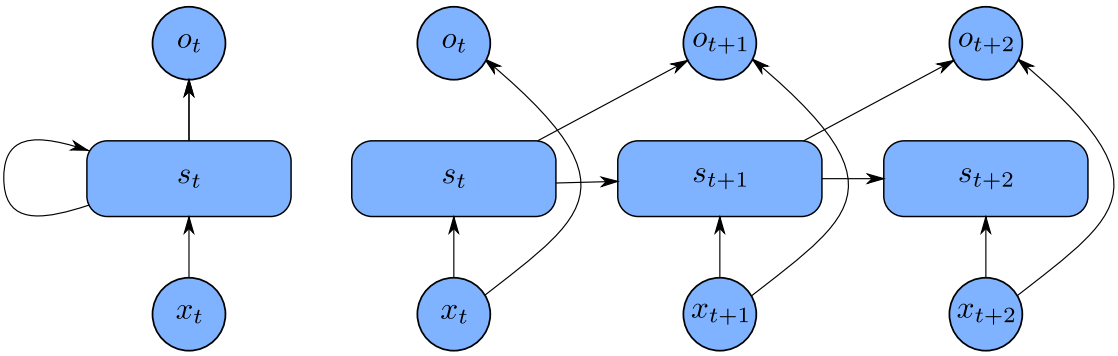


Figure 2.1: Folded and unfolded RNN representation.

Once a recurrent network model is specified, one still has to understand how to train it. There are several learning algorithm for recurrent network, with different constraints, and we are going to delve more specifically into the most well known in the next section.

2.2 Learning temporal dependencies: algorithms

The specificity of learning a recurrent neural network compared to learning a standard neural network is that we not only want to capture static dependencies, from input x_t to output y_t, but also temporal dependecies, from all $(x_{t'})_{t' \leq t}$ to output y_t. This means that we want the model to learn what to store, and how to use what it has stored to predict outputs. A typical goal of recurrent learning algorithms is to optimize a recurrent network, so that its output sequence, or predictions, is as close as possible to some target sequence, known a priori. Keeping the notations introduced in (2.5), with $\Theta = (\theta, \phi)$, the training goal is typically to minimize a loss

$$\mathcal{L}_T = \sum_{t=1}^{T} \ell_t = \sum_{t=1}^{T} \ell(o_t, y_t). \tag{2.7}$$

w.r.t Θ. To make things more concrete, let us derive a simple regression example using a standard recurrent neural network (RNN). In this case, the dynamical system takes the form

$$s_{t+1} = \tanh(W_x\, x_{t+1} + W_h\, s_t + b) \tag{2.8}$$
$$o_{t+1} = W_o s_{t+1} \tag{2.9}$$
$$\ell_{t+1} = \frac{1}{2\sigma^2} \|o_{t+1} - y_{t+1}\| \tag{2.10}$$

with parameters $\Theta = (W_x, W_h, b)$, and σ a non learnable standard deviation hyperparameter. Commonly, Θ is optimized via a gradient descent procedure, i.e. iterating

$$\Theta \leftarrow \Theta - \eta_k \frac{\partial \mathcal{L}_T}{\partial \Theta} \tag{2.11}$$

where the η's are learning rates. The focus is then to efficiently compute $\partial \mathcal{L}_T / \partial \Theta$. To derive methods for gradient computation, we use the general RNN formulation introduced in Eq. 2.6, which typically makes backpropagation equations simpler. We also slightly abuse notations, and write interchangeably $\ell(o_t, y_t)$ or $\ell(\tilde{s}_t, y_t)$, where the second equation simply discards the non output part of $\tilde{s}_t$. A first observation is that $\partial \mathcal{L}_T / \partial \Theta$ is not $\sum_{t=1}^{T} \partial_{\tilde{s}} \ell(\tilde{s}_t, y_t) \partial_\Theta F(x_t, \tilde{s}_{t-1})$. Indeed, this erroneous calculation does not take into account the effect of a change of Θ on the *trajectory* of s's: when Θ changes, starting from a given $\tilde{s}_0$, all subsequent $(\tilde{s}_t)_{t \geq 1}$ also change, and the gradient must reflect this change.

A natural way to compute the gradient is to represent computations performed by the recurrent neural network as a directed acyclic computational graphs, with edges going from quantities at time t to quantities at time $t+1$, and simply backpropagate through this time unfolded graph. Intuitively, when unfolding a recurrent network through time, each timestep corresponds to a layer, and the weights of each of the instances of the same layer at different timesteps are tied together. This gradient computation method is known as *Backpropagation Through Time* (BPTT [Jaeger, 2002]). Formally, BPTT decomposes the gradient as a sum, over timesteps t, of the effect of a change of parameter at time t on all subsequent losses:

$$\frac{\partial \mathcal{L}_T}{\partial \Theta} = \sum_{t=1}^{T} \delta \ell_t \, \frac{\partial F}{\partial \Theta}(x_t, \tilde{s}_{t-1}, \Theta) \tag{2.12}$$

where $\delta \ell_t := \frac{\partial \mathcal{L}_T}{\partial s_t}$ is computed backward iteratively according to the backpropagation equation

$$\begin{cases} \delta \ell_T = \dfrac{\partial \ell}{\partial \tilde{s}}(\tilde{s}_T, y_T) \\ \delta \ell_t = \delta \ell_{t+1} \dfrac{\partial F}{\partial \tilde{s}}(x_{t+1}, \tilde{s}_t, \Theta) + \dfrac{\partial \ell}{\partial \tilde{s}}(\tilde{s}_t, y_t). \end{cases} \tag{2.13}$$

These backpropagation equations extend the classical ones [Jaeger, 2002], which deal with the case of a simple RNN for F. BPTT is the de facto standard in training recurrent networks, and yields a computationnally efficient and unbiased estimation of the gradient.

Unfortunately, BPTT requires processing the full data sequence both forward and backward before performing a gradient step. This requires either maintaining the full unfolded network , i.e. storing the full history of inputs and activations, or recomputing part of the activations on the fly, using well chosen activation checkpoints, while backpropagating (see [Gruslys et al., 2016a]). This is impractical when very long sequences are processed with large networks. This is even more problematic when data is accessible in a streaming fashion, and there is thus no fixed sequence length. In that case, each time a new data is encountered, errors must be backpropagated through the whole history, making the cost of processing the T first samples of the data sequence quadratic in T.

Practically, this is alleviated by truncating gradient flows after a fixed number of timesteps, or equivalently, splitting the input sequence into subsequences of fixed length, and only backpropagating through those subsequences.[1] This approximation of BPTT is referred to as *Truncated BPTT* [Sutskever, 2013]. With truncation length $L < T$, the corresponding equations just drop the recurrent term $\delta\ell_{t+1}\frac{\partial F}{\partial \tilde{s}}(x_{t+1}, \tilde{s}_t, \theta)$ every L time steps, namely,

$$\delta\hat{\ell}_t := \begin{cases} \dfrac{\partial \ell}{\partial \tilde{s}}(\tilde{s}_t, y_t) & \text{if } t \text{ is a multiple of } L \\ \delta\hat{\ell}_{t+1}\dfrac{\partial F}{\partial \tilde{s}}(x_{t+1}, \tilde{s}_t, \theta) + \dfrac{\partial \ell}{\partial \tilde{s}}(\tilde{s}_t, y_t) & \text{otherwise.} \end{cases} \tag{2.14}$$

This also allows for online application: one only needs to backpropagate for L steps every L steps, making the algorithm linear in the size of the sequence, even for streaming data.

However, this gradient estimation scheme is heuristic and provides biased gradient estimates. Contrary to what one may believe, this does not necessarily prevent the algorithm from learning temporal dependencies ranging on more than L timesteps: while gradient flows are cut every L steps, recurrent states could still, due to their initialization or to generalization of short term dependencies to longer ones, still carry information from timesteps more than L steps in the past. In that case, such information can still be used by the network. Still, in general, the resulting gradient estimate is biased, and can be quite far from the true gradient even with large truncations L. Undesired behavior, and, sometimes, divergence can follow when performing gradient descent with truncated BPTT.

When unbiasedness of gradient estimates and online processing of data are the primary concern, both BPTT and TBPTT fail to efficiently fit both requirements. A third, less known alternative is *Real Time Recurrent Learning* (RTRL [Williams and Zipser, 1989, Pearlmutter, 1995]). While BPTT and TBPTT rely on backpropagation, or backward mode automatic differentiation, RTRL relies on forward mode automatic differentiation. In a nutshell, RTRL works by inductively maintaining a tensor representing the effect on the recurrent state at time t of an infinitesimal change of parameter at time 0. This quantity is exactly $\partial_\Theta \tilde{s}_t$. Once again, this is not $\partial_\Theta F(x_t, \tilde{s}_{t-1}, \Theta)$. The latter only takes into account the instantaneous effect on the state of a change of Θ. Changing Θ not only changes the value of $\tilde{s}_t$ given a fixed $\tilde{s}_{t-1}$, but also changes the value of $\tilde{s}_{t-1}$. Both effects must be taken into account when computing $\partial_\Theta \tilde{s}_t$. Differentiating Eq. (2.6), one obtains the following recurrent equation on $\partial_\Theta \tilde{s}_t$

$$\frac{\partial \tilde{s}_{t+1}}{\partial \Theta} = \frac{\partial F(x_{t+1}, \tilde{s}_t, \Theta)}{\partial \tilde{s}}\frac{\partial \tilde{s}_t}{\partial \Theta} + \frac{\partial F(x_{t+1}, \tilde{s}_t, \Theta)}{\partial \Theta}. \tag{2.15}$$

The ability to compute $\partial_\Theta \tilde{s}_t$ at each time step directly provide access to the full gradient of $\mathcal{L}_T$ w.r.t. Θ

$$\frac{\partial \mathcal{L}_T}{\partial \Theta} = \sum_{t=1}^{T} \frac{\partial \ell(\tilde{s}_t, y_t)}{\partial \tilde{s}}\frac{\partial \tilde{s}_t}{\partial \Theta}. \tag{2.16}$$

[1] Usually the internal state s_t is maintained from one subsequence to the other, not reset to a default value.

This yields an online learning algorithm for recurrent networks, that doesn't require maintaining the history of previous input data and activations to learn. Indeed, instead of performing a single gradient descent step using the full gradient Eq. (2.16) after observing T data, one can perform a stochastic gradient step

$$\Theta \leftarrow \Theta - \eta_t \frac{\partial \ell(\tilde{s}_t, y_t)}{\partial \tilde{s}} \frac{\partial \tilde{s}_t}{\partial \Theta}. \tag{2.17}$$

If the learning rates η_t's are properly scheduled, one can prove, under smoothness conditions on both the data and the model, that parameters will ultimately converge to a properly defined notion of local minimum [Massé, 2017].

While RTRL provides a way to perform memoryless online learning of recurrent networks, performing unbiased stochastic gradient descent on the loss in the limit $\eta \to 0$, it is often too computationally demanding for large networks. Indeed, RTRL requires maintaining and updating $\partial_\Theta \tilde{s}_t$. Typically, in the case of a fully connected recurrent network, as described in Eq. 2.10 with n units, $\partial_\Theta \tilde{s}_t$ keeps track of the dependency of each recurrent unit w.r.t. each parameter, making it a $n \times n^2$ object. This can be quite expensive for huge n's. Similarily, updating $\partial_\Theta \tilde{s}_t$ requires multiplying a $n \times n$ matrix by a $n \times n^2$ matrix, which requires in the order of n^4 operations at each time step, a cost n times higher than that of running the network. This is the main reason why RTRL is rarely used in practice.

Descriptions of these three algorithms, as well as their inherent tradeoffs are dwelled upon in more details in chapters 3 and 4.

2.3 Learning temporal dependencies: architectures

One of the most studied and simplest recurrent network architecture is the standard recurrent neural network (SRNN) as defined in Eq. (2.10). A promising property of standard RNNs is that they are universal program approximators [Siegelmann and Sontag, 1995], and are therefore as expressive as a model can be. However, SRNNs are considered to be hard to train, and notably fail in learning medium to long term dependencies [Bengio et al., 1994], rendering their expressivity of little use. The standard explanation for this learning deficiency is that SRNNs gradients are ill-conditionned, and only yield very little signal on long term dependencies. This is known as the *vanishing gradient problem* [Hochreiter, 1991]. In the next sections, we are going to look at the formalisation, as well as some partial solutions to the vanishig gradient problem.

2.3.1 Standard RNNs and the vanishing gradient problem

At the core of the vanishing gradient problem is the fact that, for many recurrent architectures, and notably for standard recurrent networks, the derivative of the current recurrent state with respect to a distant recurrent state, $\partial_{s_{t-T}} s_t$ is ill conditionned, and, in some cases, vanishes exponentially with T.

To have a deeper understanding of why this could happen, let us consider a simplified network, where a single input is given to the network at time 0, through the initial state

s_0:

$$s_{t+1} = f(W_s s_t) \tag{2.18}$$

where f is a non-linearity. In such a case, the gradient of s_t with respect to s_{t-T} is easily computable

$$\frac{\partial s_t}{\partial s_{t-T}} = \prod_{k=t-T}^{t-1} \text{diag}(f'(W_s s_k)) W_s. \tag{2.19}$$

We are interested in what happens to an error vector, when it is backpropagated through the network. If we backpropagate a vector δs at time t, the backpropagation component that goes exactly T steps in the past is $\delta s^T \partial_{s_{t-T}} s_t$. The norm of the resulting vector is bounded by $\|\delta s\|_2 \|(\partial_{s_{t-T}} s_t)^T\|_2$, where $\|\cdot\|_2$ denotes the ℓ_2-norm on vectors, and the associated operator norm on matrices. Consequently, the fraction of the norm of δs that remains after T steps of backpropagation is upper bounded by $\|(\partial_{s_{t-T}} s_t)^T\|_2$. Now this quantity can be further bounded. If the derivative of f is bounded by a constant γ, as is the case for sigmoid or hyperbolic tangent nonlinearities, one gets

$$\|(\partial_{s_{t-T}} s_t)^T\|_2 \leq \prod_{k=t-T}^{t-1} \|\text{diag}(f'(W_s s_k))\|_2 \|W_s\|_2 \tag{2.20}$$

$$\leq \gamma^{T+1} \sigma_{\max}(W_s)^T \tag{2.21}$$

where $\sigma_{\max}(W_s)$ is the highest singular value of W_s (i.e. the highest eigenvalue of $W_s W_s^T$). If $\sigma_{\max}(W_s) < \frac{1}{\gamma}$, then $\|(\partial_{s_{t-T}} s_t)^T\|_2$ vanishes exponentially with T.

These assumptions are however rarely met in practice. Typically, for a *tanh* network with n recurrent units, if W is initalized with each entry being a standard gaussian of variance $1/n$, the singular values of W typically lie in the interval $(0, 2)$. With a sufficiently high n, there is a high probability that one of the singular values lies above 1, rendering the bound useless. A problem with this analysis of vanishing gradient is that it does not take into account the saturating effect of nonlinearities, which can strongly accentuate gradient vanishing. This effect is difficult to formally analyse, since it would require differentiating between activations lying near zeros, and activations lying far from zero.

More discussions on the vanishing gradient problem can be found in [Hochreiter and Schmidhuber, 1997, Bengio et al., 1994, Hochreiter, 1991, Martens and Sutskever, 2011, Pascanu et al., 2012, Graves et al., 2013].

2.3.2 From RNNs to LSTMs

To date, the most practically successful approach to mitigating the vanishing gradient problem has been the use of direct feedback connections from neurons to themselves. This is the central idea of *Long Short Term Memories* (LSTM [Hochreiter and Schmidhuber, 1997]). [Hochreiter and Schmidhuber, 1997] equips each unit with a direct feedback connection, providing a bypass connection for gradient flows. The original LSTMs update equations involve a recurrent state split into two parts, $s_t = (c_t, h_t)$, and updates

written as

$$i_{t+1} = \sigma(W_{ix}x_{t+1} + W_{ih}h_t + W_{ic}c_t + b_i) \tag{2.22}$$
$$c_{t+1} = c_t + i_{t+1} \odot \tanh(W_{cx}x_{t+1} + W_{ch}h_t + W_{cc}c_t + b_c) \tag{2.23}$$
$$o_{t+1} = \sigma(W_{ox}x_{t+1} + W_{oh}h_t + W_{oc}c_t + b_o) \tag{2.24}$$
$$h_{t+1} = o_{t+1} \tanh(c_{t+1}) \tag{2.25}$$

with i_{t+1} the input gates, which allow inputs to flow in when close to 1, and block inputs when close to 0, o_{t+1} the output gates, which allow outputs to flow out when close to 1 and block outputs when close to 0, c_{t+1} the LSTM inner cells, which are used to store information for long period of time and h_{t+1} the LSTM hidden units, which expose the cell to the outside world, when the output gates allow it. In [Hochreiter and Schmidhuber, 1997], all recurrent gradient flows that are not directly flowing from the unit to itself are simply cut, suppressing the vanishing gradient problem alltogether, but providing biased gradient estimates. Additionnally to being biased, such networks have problems forgetting information [Gers et al., 1999]. This motivates the introduction of a forget gate, mitigating the contribution of c_t on c_{t+1}. Typically, one flavour of such LSTMs updates its state as

$$i_{t+1} = \sigma(W_{ix}x_{t+1} + W_{ih}h_t + W_{ic}c_t + b_i) \tag{2.26}$$
$$f_{t+1} = \sigma(W_{fx}x_{t+1} + W_{fh}h_t + W_{fc}c_t + b_f) \tag{2.27}$$
$$c_{t+1} = f_{t+1} \odot c_t + i_{t+1} \odot \tanh(W_{cx}x_{t+1} + W_{ch}h_t + W_{cc}c_t + b_c) \tag{2.28}$$
$$o_{t+1} = \sigma(W_{ox}x_{t+1} + W_{oh}h_t + W_{oc}c_t + b_o) \tag{2.29}$$
$$h_{t+1} = o_{t+1} \tanh(c_{t+1}) \tag{2.30}$$

A schematic representation of the recurrent updates of a LSTM Cell is given in Figure 2.2. Subsequent work [Graves et al., 2013] removed the arbitrary cuts of gradient flows, and obtained similar results, without the exact constant flow of recurrent gradients.

2.3.3 Other forms of vanishing gradient mitigation

Other approaches have been proposed to mitigate vanishing gradient. Among the most well studied are orthogonal recurrent networks, the use of multi steps, potentially adaptive bypass connections and the use of external memory modules.

Orthogonal Recurrent Neural Networks

The usual analysis of vanishing gradient, as exposed in Section 2.3.1 mostly rely on the observation that the highest singular value of the state to state recurrent matrix plays a huge role in the advent of vanishing gradient. A natural solution to ensure more reliable gradient propagation is to constraint the singular values of W_s to $\frac{1}{\gamma}$, typically 1 in the case of tanh nonlinearities. This amount to picking W_s in the set of orthogonal matrices.

Optimizing in the set of orthogonal matrices is trickier than optimizing in the set of all matrices, since one needs to preserve the orthogonality property from one gradient step

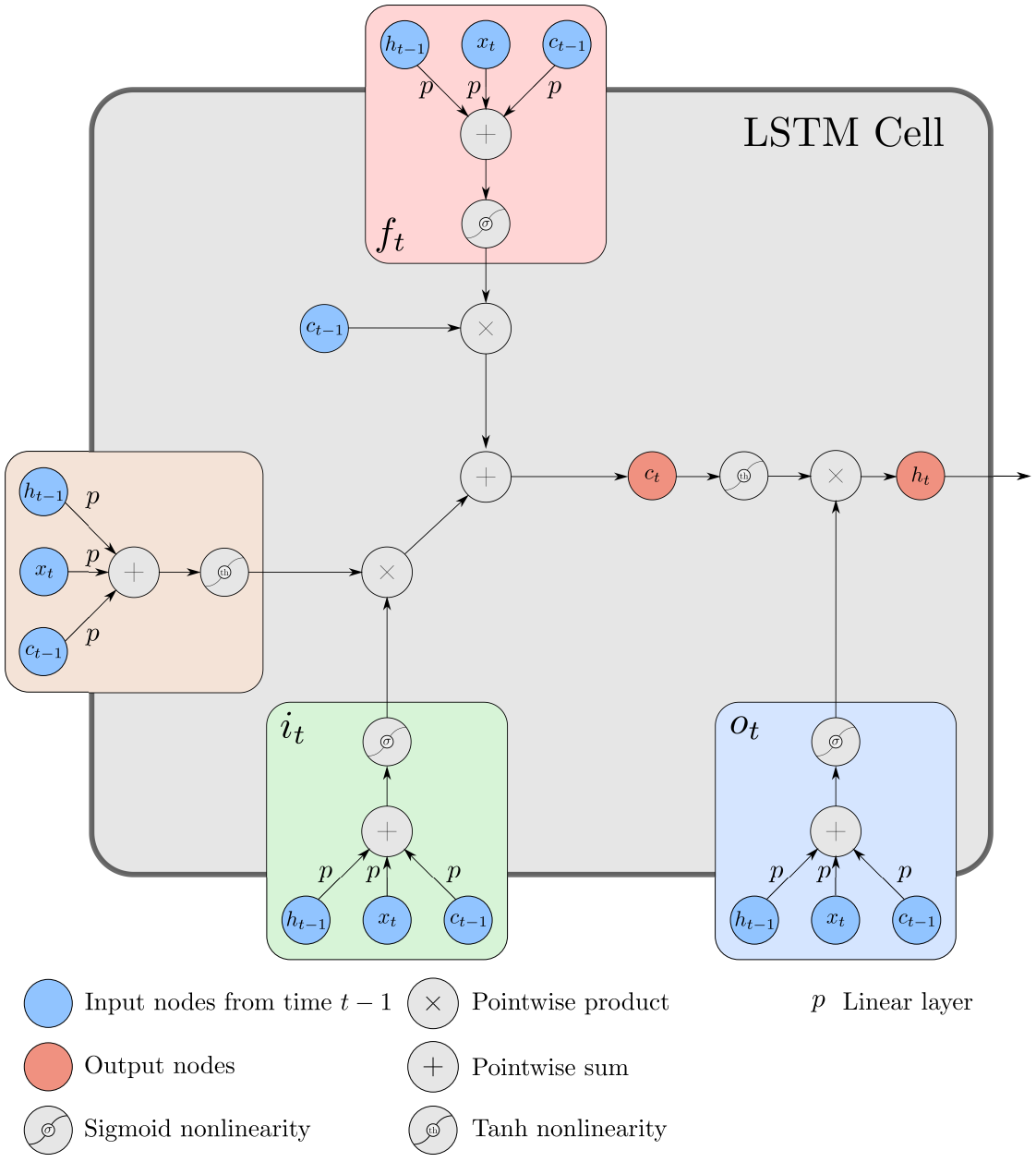


Figure 2.2: A LSTM network unfolded on 1 timestep

to the other. Several approach have been introduced to efficiently optimize in this set. Some approaches involve optimizing in an easily parametrizable subset of all orthogonal matrices [Arjovsky et al., 2016]. Such sets can be made arbitrarily general [Helfrich et al., 2017, Mhammedi et al., 2017]. Other approaches propose to directly restrict the gradient descent procedure to the manifold of orthogonal matrices [Wisdom et al., 2016a].

While Orthogonal RNNs (ORNN) often display strong performance on synthetic benchmarks involving long to very long term dependencies, in practice, LSTMs are often preferred to ORNN. A notable downside of orthogonal recurrent networks is that they lack an efficient forgetting mechanism [Jing et al., 2019], which may explain this performance gap on real world benchmarks.

Memory networks and hierarchical architectures

To directly cope for the difficulty of learning long term dependencies, explicit temporally hierarchical RNN and explicit external memory modules architectures have been introduced.

Finding efficient temporally hierarchical architectures has been a long standing challenge in RNN design [El Hihi and Bengio, 1995]. Attempts include *Clockwork RNNs* [Koutnik et al., 2014] or *Hierarchical Multiscale Recurrent Neural Networks* [Chung et al., 2016]. Such approaches typically solve the vanishing gradient problem by introducing hierarchical bypass connections that can directly propagate gradients over long temporal horizon. A drawback of such approaches is that building hard bypass connections is difficult. Some approaches [Koutnik et al., 2014] hardcode bypass connections making gradient flow only from certain timesteps to others. This can cause the network to only capture specific long term dependencies, and to miss precise timings. On the other hand, trying to learn hard bypass connections requires learning discrete variables, which is not easily doable using backpropagation. This requires either using biased gradient approximator such as the *Straight through* approximator [Bengio et al., 2013], or using very high variance unbiased estimates, inspired by the REINFORCE estimator [Williams, 1992]. Both approaches can significantly hurt learning.

Typical examples of networks introducing explicit memory modules are the *Neural Turing Machine* [Graves et al., 2014a], its successor, the *Dynamic Neural Computer* [Graves et al., 2016], or simpler variations, such as *Memory Networks* [Weston et al., 2014]. Most of these architectures rely on a memory module with a content based addressing scheme: the memory module provided to the network stores $(\mathbf{key}, \mathbf{value})$ pairs. At each timestep, a controller (typically a LSTM) produces a certain number of keys. The produced keys are compared to the memory module keys using a similarity metric, and a value is produced by taking a weighted average of the memory module values, weighted by the similarity metric. This is a specific case of attention mechanism [Bahdanau et al., 2014]. Variations of such mechanisms, were the memory module is directly replaced by an attention mechanism on the input sequence, have produced impressive results in sequence to sequence tasks, such as machine translation [Luong et al., 2015], and were only very recently outdone by pure, non recurrent,

attention based architectures such as the *Transformer* [Vaswani et al., 2017]. One of the drawbacks of recurrent architectures with external memory modules is that the memory module is part of the network activations. This means that BPTT requires storing the state of the memory at all points when processing inputs, and backpropagate through all those memory checkpoints. This memory cost can quickly become overwhelming for long sequences.

2.4 Contributions

In Chapters 3 and 4, we provide algorithms that can process sequential streams of data online, without having to backtrack through the history of inputs and activations, thus addressing the problems with learning algorithms exposed in Section 2.2. In Chapter 5, we provide a different take on why standard neural networks fail to capture long term dependencies, in term of invariance to temporal transformations, thus addressing the problem exposed in Section 2.3.

Chapter 3

Unbiased Online Recurrent Optimization

Foreword:

As mentionned in section 2.2, when faced with an infinite stream of data, standard recurrent networks learning algorithms are ill-adapted. *NoBackTrack*, as presented in [Ollivier et al., 2015], conveniently filled the gap by providing an unbiased estimate of the loss gradient that could be efficiently computed in online setups. While being generally applicable in theory, NoBackTrack proved to be difficult to implement for complex RNN architectures. This motivated *Unbiased Online Recurrent Optimization* [Tallec and Ollivier, 2017a], a simplification of NoBackTrack, that provides the same theoretical guarantees, but is much easier to implement for complex networks. UORO, as described in this chapter, was originally presented at the *International Conference on Learning Representation* (ICLR) in 2018.

In this chapter, we present *Unbiased Online Recurrent Optimization* (UORO) an algorithm that allows for online learning of general recurrent computational graphs such as recurrent network models. It works in a streaming fashion and avoids backtracking through past activations and inputs. UORO is computationally as costly as *Truncated Backpropagation Through Time* (truncated BPTT), a widespread algorithm for online learning of recurrent networks [Jaeger, 2002]. UORO is a modification of *NoBackTrack* [Ollivier et al., 2015] that bypasses the need for model sparsity and makes implementation easy in current deep learning frameworks, even for complex models. Like NoBackTrack, UORO provides unbiased gradient estimates; unbiasedness is the core hypothesis in stochastic gradient descent theory, without which convergence to a local optimum is not guaranteed. On the contrary, truncated BPTT does not provide this property, leading to possible divergence. On synthetic tasks where truncated BPTT is shown to diverge, UORO converges. For instance, when a parameter has a positive short-term but negative long-term influence, truncated BPTT diverges unless the truncation span is very significantly longer than the intrinsic temporal range of the interactions, while UORO performs well thanks to the unbiasedness of its gradients.

Introduction

Current recurrent network learning algorithms are ill-suited to online learning via a single pass through long sequences of temporal data. *Backpropagation Through Time* (BPTT [Jaeger, 2002]), the current standard for training recurrent architectures, is well suited to many short training sequences. Treating long sequences with BPTT requires either storing all past inputs in memory and waiting for a long time between each learning step, or arbitrarily splitting the input sequence into smaller sequences, and applying BPTT to each of those short sequences, at the cost of losing long term dependencies.

This paper introduces *Unbiased Online Recurrent Optimization* (UORO), an *online* and *memoryless* learning algorithm for recurrent architectures: UORO processes and learns from data samples sequentially, one sample at a time. Contrary to BPTT, UORO does not maintain a history of previous inputs and activations. Moreover, UORO is *scalable*: processing data samples with UORO comes at a similar computational and memory cost as just running the recurrent model on those data.

Like most neural network training algorithms, UORO relies on stochastic gradient optimization. The theory of stochastic gradient crucially relies on the unbiasedness of gradient estimates to provide convergence to a local optimum. To this end, in the footsteps of *NoBackTrack* (NBT) [Ollivier et al., 2015], UORO provides provably *unbiased* gradient estimates, in a scalable, streaming fashion.

Unlike NBT, though, UORO can be easily implemented in a black-box fashion on top of an existing recurrent model in current machine learning software, without delving into the structure and code of the model.

The framework for recurrent optimization and UORO is introduced in Section 3.2. The final algorithm is reasonably simple (Alg. 1), but its derivation (Section 3.3) is more complex. In Section 3.7, UORO is shown to provide convergence on a set of

synthetic experiments where truncated BPTT fails to display reliable convergence. An implementation of UORO is provided as supplementary material.

3.1 Related work

A widespread approach to online learning of recurrent neural networks is *Truncated Backpropagation Through Time* (truncated BPTT) [Jaeger, 2002], which mimics Backpropagation Through Time, but zeroes gradient flows after a fixed number of timesteps. This truncation makes gradient estimates biased; consequently, truncated BPTT does not provide any convergence guarantee. Learning is biased towards short-time dependencies. Storage of some past inputs and states is required.

Online, exact gradient computation methods have long been known (*Real Time Recurrent Learning* (RTRL) [Williams and Zipser, 1989, Pearlmutter, 1995]), but their computational cost discards them for reasonably-sized networks.

NoBackTrack (NBT) [Ollivier et al., 2015] also provides unbiased gradient estimates for recurrent neural networks. However, contrary to UORO, NBT cannot be applied in a blackbox fashion, making it extremely tedious to implement for complex architectures.

Other previous attempts to introduce generic online learning algorithms with a reasonable computational cost all result in biased gradient estimates. *Echo State Networks* (ESNs) [Jaeger, 2002, Jaeger et al., 2007] simply set to 0 the gradients of recurrent parameters. Others, e.g., [Maass et al., 2002, Steil, 2004], introduce approaches resembling ESNs, but keep a partial estimate of the recurrent gradients. The original *Long Short Term Memory* algorithm [Hochreiter and Schmidhuber, 1997] (LSTM now refers to a particular architecture) cuts gradient flows going out of gating units to make gradient computation tractable. *Decoupled Neural Interfaces* [Jaderberg et al., 2016] bootstrap truncated gradient estimates using synthetic gradients generated by feedforward neural networks. The algorithm in [Movellan et al., 2002] provides zeroth-order estimates of recurrent gradients via diffusion networks; it could arguably be turned online by running randomized alternative trajectories. Generally these approaches lack a strong theoretical backing, except arguably ESNs.

3.2 Background

UORO is a learning algorithm for recurrent computational graphs. Formally, the aim is to optimize θ, a parameter controlling the evolution of a dynamical system

$$s_{t+1} = F_{\text{state}}(x_{t+1}, s_t, \theta) \tag{3.1}$$

$$o_{t+1} = F_{\text{out}}(x_{t+1}, s_t, \theta) \tag{3.2}$$

in order to minimize a total loss

$$\mathcal{L} := \sum_{0 \leq t \leq T} \ell_t(o_t, o_t^*), \tag{3.3}$$

where o_t^* is a target output at time t. For instance, a standard recurrent neural network, with hidden state s_t (preactivation values) and output o_t at time t, is described with the update equations

$$F_{\text{state}}(x_{t+1}, s_t, \theta) := W_x\, x_{t+1} + W_s \tanh(s_t) + b \tag{3.4}$$

$$F_{\text{out}}(x_{t+1}, s_t, \theta) := W_o \tanh(F_{\text{state}}(x_{t+1}, s_t, \theta)) + b_o; \tag{3.5}$$

here the parameter is $\theta = (W_x, W_s, b, W_o, b_o)$, and a typical loss might be $\ell_s(o_s, o_s^*) := (o_s - o_s^*)^2$.

Optimization by gradient descent is standard for neural networks. In the spirit of stochastic gradient descent, we can optimize the total loss one term at a time and update the parameter online at each time step via

$$\theta \leftarrow \theta - \eta_t \frac{\partial \ell_t}{\partial \theta}^\top \tag{3.6}$$

where η_t is a scalar learning rate at time t. (Other gradient-based optimizers can also be used, once $\frac{\partial \ell_t}{\partial \theta}$ is known.) The focus is then to compute, or approximate, $\frac{\partial \ell_t}{\partial \theta}$.

BPTT computes $\frac{\partial \ell_t}{\partial \theta}$ by unfolding the network through time, and backpropagating through the unfolded network, each timestep corresponding to a layer. BPTT thus requires maintaining the full unfolded network, or, equivalently, the history of past inputs and activations. [1] *Truncated BPTT* only unfolds the network for a fixed number of timesteps, reducing computational cost in online settings [Jaeger, 2002]. This comes at the cost of biased gradients, and can prevent convergence of the gradient descent even for large truncations, as clearly exemplified in Fig. 3.2a.

3.3 Unbiased Online Recurrent Optimization

Unbiased Online Recurrent Optimization is built on top of a forward computation of the gradients, rather than backpropagation. Forward gradient computation for neural networks (RTRL) is described in [Williams and Zipser, 1989] and we review it in Section 3.4. The derivation of UORO follows in Section 3.4.1. Implementation details are given in Section 3.4.2. UORO's derivation is strongly connected to [Ollivier et al., 2015] but differs in one critical aspect: the sparsity hypothesis made in the latter is relieved, resulting in reduced implementation complexity without any model restriction. The proof of UORO's convergence to a local optimum can be found in [Massé, 2017].

3.4 Forward computation of the gradient

Forward computation of the gradient for a recurrent model (RTRL) is directly obtained by applying the chain rule to both the loss function and the state equation (3.1), as follows.

[1] Storage of past activations can be reduced, e.g. [Gruslys et al., 2016b]. However, storage of all past inputs is necessary.

Direct differentiation and application of the chain rule to ℓ_{t+1} yields

$$\frac{\partial \ell_{t+1}}{\partial \theta} = \frac{\partial \ell_{t+1}}{\partial o}(o_{t+1}, o^*_{t+1}) \cdot \left(\frac{\partial F_{\text{out}}}{\partial s}(x_{t+1}, s_t, \theta) \frac{\partial s_t}{\partial \theta} + \frac{\partial F_{\text{out}}}{\partial \theta}(x_{t+1}, s_t, \theta) \right). \tag{3.7}$$

Here, the term $\partial s_t / \partial \theta$ represents the effect on the state at time t of a change of parameter during the whole past trajectory. This term can be computed inductively from time t to $t+1$. Intuitively, looking at the update equation (3.1), there are two contributions to $\partial s_{t+1}/\partial \theta$:

- The direct effect of a change of θ on the computation of s_{t+1}, given s_t.
- The past effect of θ on s_t via the whole past trajectory.

With this in mind, differentiating (3.1) with respect to θ yields

$$\frac{\partial s_{t+1}}{\partial \theta} = \frac{\partial F_{\text{state}}}{\partial \theta}(x_{t+1}, s_t, \theta) + \frac{\partial F_{\text{state}}}{\partial s}(x_{t+1}, s_t, \theta) \frac{\partial s_t}{\partial \theta}. \tag{3.8}$$

This gives a way to compute the derivative of the instantaneous loss without storing past history: at each time step, update $\partial s_t / \partial \theta$ from $\partial s_{t-1}/\partial \theta$, then use this quantity to directly compute $\partial \ell_{t+1}/\partial \theta$. This is how RTRL [Williams and Zipser, 1989] proceeds.

A huge disadvantage of RTRL is that $\partial s_t / \partial \theta$ is of size dim(state) $\times$ dim(params). For instance, with a fully connected standard recurrent network with n units, $\partial s_t / \partial \theta$ scales as n^3. This makes RTRL impractical for reasonably sized networks.

UORO modifies RTRL by only maintaining a scalable, rank-one, provably unbiased approximation of $\partial s_t / \partial \theta$, to reduce the memory and computational cost. This approximation takes the form $\tilde{s}_t \otimes \tilde{\theta}_t$, where $\tilde{s}_t$ is a column vector of the same dimension as s_t, $\tilde{\theta}_t$ is a *row* vector of the same dimension as $\theta^\top$, and $\otimes$ denotes the outer product. The resulting quantity is thus a matrix of the same size as $\partial s_t / \partial \theta$. The memory cost of storing $\tilde{s}_t$ and $\tilde{\theta}_t$ scales as dim(state) + dim(params). Thus UORO is as memory costly as simply running the network itself (which indeed requires to store the current state and parameters). The following section details how $\tilde{s}_t$ and $\tilde{\theta}_t$ are built to provide unbiasedness.

3.4.1 Rank-one trick: from RTRL to UORO

Given an unbiased estimation of $\partial s_t / \partial \theta$, namely, a stochastic matrix $\tilde{G}_t$ such that $\mathbb{E}\, \tilde{G}_t = \partial s_t / \partial \theta$, unbiased estimates of $\partial \ell_{t+1}/\partial \theta$ and $\partial s_{t+1}/\partial \theta$ can be derived by plugging $\tilde{G}_t$ in (3.7) and (3.8). Unbiasedness is preserved thanks to linearity of the mean, because both (3.7) and (3.8) are affine in $\partial s_t / \partial \theta$.

Thus, assuming the existence of a rank-one unbiased approximation $\tilde{G}_t = \tilde{s}_t \otimes \tilde{\theta}_t$ at time t, we can plug it in (3.8) to obtain an unbiased approximation $\hat{G}_{t+1}$ at time $t+1$

$$\hat{G}_{t+1} = \frac{\partial F_{\text{state}}}{\partial \theta}(x_{t+1}, s_t, \theta) + \frac{\partial F_{\text{state}}}{\partial s}(x_{t+1}, s_t, \theta)\, \tilde{s}_t \otimes \tilde{\theta}_t. \tag{3.9}$$

However, in general this is no longer rank-one.

To transform $\hat{G}_{t+1}$ into $\tilde{G}_{t+1}$, a rank-one unbiased approximation, the following rank-one trick, introduced in [Ollivier et al., 2015] is used:

Proposition 1. *Let A be a real matrix that decomposes as*

$$A = \sum_{i=1}^{k} v_i \otimes w_i. \tag{3.10}$$

Let ν be a vector of k independent random signs, and ρ a vector of k positive numbers. Consider the rank-one matrix

$$\tilde{A} := \left(\sum_{i=1}^{k} \rho_i \nu_i v_i\right) \otimes \left(\sum_{i=1}^{k} \frac{\nu_i w_i}{\rho_i}\right) \tag{3.11}$$

Then $\tilde{A}$ is an unbiased rank-one approximation of A: $\mathbb{E}_\nu \tilde{A} = A$.

The rank-one trick can be applied for any ρ. The choice of ρ influences the variance of the approximation; choosing

$$\rho_i = \sqrt{\|w_i\| \,/\, \|v_i\|} \tag{3.12}$$

minimizes the variance of the approximation, $\mathbb{E}\left[\|A - \tilde{A}\|_2^2\right]$ [Ollivier et al., 2015].

The UORO update is obtained by applying the rank-one trick twice to (3.9). First, $\frac{\partial F_{\text{state}}}{\partial \theta}(x_{t+1}, s_t, \theta)$ is reduced to a rank one matrix, without variance minimization. [2] Namely, let ν be a vector of independant random signs; then,

$$\frac{\partial F_{\text{state}}}{\partial \theta}(x_{t+1}, s_t, \theta) = \mathbb{E}_\nu \left[\nu \otimes \nu^\top \frac{\partial F_{\text{state}}}{\partial \theta}(x_{t+1}, s_t, \theta)\right]. \tag{3.13}$$

This results in a rank-two, unbiased estimate of $\partial s_{t+1}/\partial \theta$ by substituting (3.13) into (3.9)

$$\frac{\partial F_{\text{state}}}{\partial s}(x_{t+1}, s_t, \theta)\, \tilde{s}_t \otimes \tilde{\theta}_t + \nu \otimes \left(\nu^\top \frac{\partial F_{\text{state}}}{\partial \theta}(x_{t+1}, s_t, \theta)\right). \tag{3.14}$$

Applying Prop. 1 again to this rank-two estimate, with variance minimization, yields UORO's estimate $\tilde{G}_{t+1}$

$$\tilde{G}_{t+1} = \left(\rho_0 \frac{\partial F_{\text{state}}}{\partial s}(x_{t+1}, s_t, \theta)\, \tilde{s}_t + \rho_1\, \nu\right) \otimes \left(\frac{\tilde{\theta}_t}{\rho_0} + \frac{\nu}{\rho_1}^\top \frac{\partial F_{\text{state}}}{\partial \theta}(x_{t+1}, s_t, \theta)\right) \tag{3.15}$$

which satisfies that $\mathbb{E}_\nu \tilde{G}_{t+1}$ is equal to (3.9). (By elementary algebra, some random signs that should appear in (3.15) cancel out.) Here

$$\rho_0 = \sqrt{\frac{\|\tilde{\theta}_t\|}{\|\frac{\partial F_{\text{state}}}{\partial s}(x_{t+1}, s_t, \theta)\, \tilde{s}_t\|}}, \quad \rho_1 = \sqrt{\frac{\|\nu^\top \frac{\partial F_{\text{state}}}{\partial \theta}(x_{t+1}, s_t, \theta)\|}{\|\nu\|}} \tag{3.16}$$

[2] Variance minimization is not used at this step, since computing $\sqrt{\frac{\|w_i\|}{\|v_i\|}}$ for every i is not scalable.

minimizes variance of the second reduction.

The unbiased estimation (3.15) is rank-one and can be rewritten as $\tilde{G}_{t+1} = \tilde{s}_{t+1} \otimes \tilde{\theta}_{t+1}$ with the update

$$\tilde{s}_{t+1} \leftarrow \rho_0 \frac{\partial F_{\text{state}}}{\partial s}(x_{t+1}, s_t, \theta)\, \tilde{s}_t + \rho_1\, \nu \tag{3.17}$$

$$\tilde{\theta}_{t+1} \leftarrow \frac{\tilde{\theta}_t}{\rho_0} + \frac{\nu^\top}{\rho_1} \frac{\partial F_{\text{state}}}{\partial \theta}(x_{t+1}, s_t, \theta). \tag{3.18}$$

Initially, $\partial s_0/\partial\theta = 0$, thus $\tilde{s}_0 = 0$, $\tilde{\theta}_0 = 0$ yield an unbiased estimate at time 0. Using this initial estimate and the update rules (3.17)–(3.18), an estimate of $\partial s_t/\partial\theta$ is obtained at all subsequent times, allowing for online estimation of $\partial \ell_t/\partial\theta$. Thanks to the construction above, by induction all these estimates are unbiased. [3]

We are left to demonstrate that these update rules are scalably implementable.

3.4.2 Implementation

Implementing UORO requires maintaining the rank-one approximation and the corresponding gradient loss estimate. UORO's estimate of the loss gradient $\partial \ell_{t+1}/\partial_\theta$ at time $t+1$ is expressed by plugging into (3.7) the rank-one approximation $\partial s_t/\partial\theta \approx \tilde{s}_t \otimes \tilde{\theta}_t$, which results in

$$\left(\frac{\partial \ell_{t+1}}{\partial o}(o_{t+1}, o^*_{t+1}) \frac{\partial F_{\text{out}}}{\partial s}(x_{t+1}, s_t, \theta) \cdot \tilde{s}_t \right) \tilde{\theta}_t + \frac{\partial \ell_{t+1}}{\partial o}(o_{t+1}, o^*_{t+1}) \frac{\partial F_{\text{out}}}{\partial \theta}(x_{t+1}, s_t, \theta). \tag{3.19}$$

Backpropagating $\partial \ell_{t+1}/\partial o_{t+1}$ once through F_{out} returns $(\partial \ell_{t+1}/\partial o_{t+1} \cdot \partial F_{\text{out}}/\partial x_{t+1},\ \partial \ell_{t+1}/\partial o_{t+1} \cdot \partial F_{\text{out}}/\partial s_t,\ \partial \ell_{t+1}/\partial o_{t+1} \cdot \partial F_{\text{out}}/\partial\theta)$, thus providing all necessary terms to compute (3.19).

Updating $\tilde{s}$ and $\tilde{\theta}$ requires applying (3.17)–(3.18) at each step. Backpropagating the vector of random signs ν once through F_{state} returns $\big(\_, \_, \nu^\top \partial F_{\text{state}}(x_{t+1}, s_t, \theta)/\partial\theta\big)$, providing for (3.18).

Updating $\tilde{s}$ via (3.17) requires computing $(\partial F_{\text{state}}/\partial s_t) \cdot \tilde{s}_t$. This is computable numerically through

$$\frac{\partial F_{\text{state}}}{\partial s}(x_{t+1}, s_t, \theta) \cdot \tilde{s}_t = \lim_{\varepsilon \to 0} \frac{F_{\text{state}}(x_{t+1}, s_t + \varepsilon\, \tilde{s}_t, \theta) - F_{\text{state}}(x_{t+1}, s_t, \theta)}{\varepsilon} \tag{3.20}$$

computable through two applications of F_{state}. This operation is referred to as tangent forward propagation [Simard et al., 1991] and can also often be computed algebraically.

This allows for complete implementation of one step of UORO (Alg. 1). The cost of UORO (including running the model itself) is three applications of F_{state}, one application of F_{out}, one backpropagation through F_{out} and F_{state}, and a few elementwise operations on vectors and scalar products.

[3] In practice, since θ changes during learning, unbiasedness only holds exactly in the limit of small learning rates. This is not specific to UORO as it also affects RTRL.

The resulting algorithm is detailed in Alg. 1. $F.\textbf{forward}(v)$ denotes pointwise application of F at point v, $F.\textbf{backprop}(v, \delta o)$ backpropagation of row vector δo through F at point v, and $F.\textbf{forwarddiff}(v, \delta v)$ tangent forward propagation of column vector δv through F at point v. Notably, $F.\textbf{backprop}(v, \delta o)$ has the same dimension as $v^\top$, e.g. $F_{\text{out}}.\textbf{backprop}((x_{t+1}, s_t, \theta), \delta o_{t+1})$ has three components, of the same dimensions as $x_{t+1}^\top$, $s_t^\top$ and $\theta^\top$.

The proposed update rule for stochastic gradient descent (3.6) can be directly adapted to other optimizers, e.g. *Adaptative Momentum* (Adam) [Kingma and Ba, 2014] or *Adaptative Gradient* [Duchi et al., 2010]. Vanilla stochastic gradient descent (SGD) and Adam are used hereafter. In Alg. 1, such optimizers are denoted by SGDOpt and the corresponding parameter update given current parameter θ, gradient estimate g_t and learning rate η_t is denoted $\text{SGDOpt}.\textbf{update}(g_t, \eta_t, \theta)$.

3.4.3 Memory-T UORO and rank-k UORO

The unbiased gradient estimates of UORO injects noise via ν, thus requiring smaller learning rates. To reduce noise, UORO can be used on top of truncated BPTT so that recent gradients are computed exactly.

Formally, this just requires applying Algorithm 1 to a new transition function F^T which is just T consecutive steps of the original model F. Then the backpropagation operation in Algorithm 1 becomes a backpropagation over the last T steps, as in truncated BPTT. The loss of one step of F^T is the sum of the losses of the last T steps of F, namely

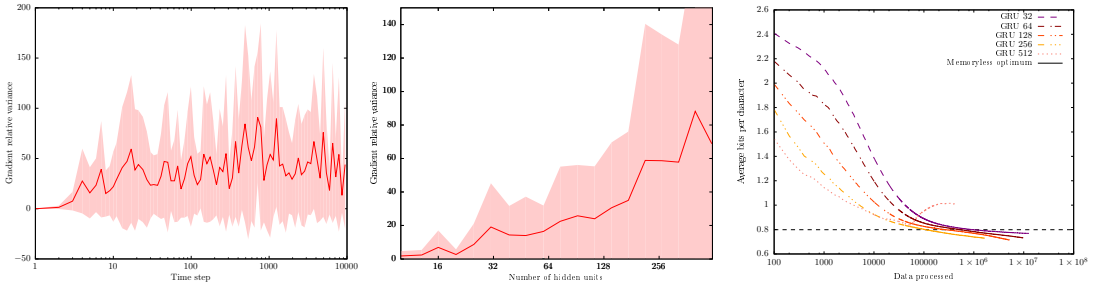
$$\ell_{t+1}^{t+T} := \sum_{k=t+1}^{t+T} \ell_k. \tag{3.21}$$

Likewise, the forward tangent propagation is performed through F^T. This way, we obtain an unbiased gradient estimate in which the gradients from the last T steps are computed exactly and incur no noise. The resulting algorithm is referred to as memory-T UORO. Its scaling in T is similar to T-truncated BPTT, both in terms of memory and computation. In the experiments below, memory-T UORO reduced variance early on, but did not significantly impact later performance.

The noise in UORO can also be reduced by using higher-rank gradient estimates (rank-r instead of rank-1), which amounts to maintaining r distinct values of $\tilde{s}$ and $\tilde{\theta}$ in Algorithm 1 and averaging the resulting values of $\tilde{g}$. We did not exploit this possibility in the experiments below, although $r = 2$ visibly reduced variance in preliminary tests.

3.5 UORO's variance is stable as time goes by

Gradient-based sequential learning on an unbounded data stream requires that the variance of the gradient estimate does not explode through time. UORO is specifically built to provide an unbiased estimate whose variance does not explode over time.



(a) (b) (c)

Figure 3.1: (a) The relative variance of UORO gradient estimates does not significantly increase with time. Note the logarithmic scale on the time axis. (b) The relative variance of UORO gradient estimates significantly increases with network size. Note the logarithmic scale on number of units. (c) Variance of larger networks affects learning on a small range copy task.

A precise statement regarding UORO's convergence and boundedness of the variance of gradients is provided in [Massé, 2017]. Informally, when the largest eigenvalue of the differential transition operator $\partial F_{\text{state}}/\partial s$ is uniformly bounded by a constant $\delta < 1$ (which characterizes stable dynamical systems), the normalizing factors in (3.17) and (3.18) enforce that the influence of previous ν's decrease exponentially with time.

We hereby provide an experimental validation of the boundedness of UORO's variance in Fig. 3.1a. To monitor the variance of UORO's estimate over time, a 64-unit GRU recurrent network is trained on the first 10^7 characters of the full works of Shakespeare using UORO. The network is then rerun 100 times on the 10000 first characters of the text, and gradients estimates at each time steps are computed, but not applied. The gradient relative variance, that is

$$\frac{\mathbb{E}\left[\|g_t - \mathbb{E}\left[g_t\right]\|^2\right]}{\|\mathbb{E}\left[g_t\right]\|^2}, \tag{3.22}$$

is computed, where the average is taken with respect to runs. This quantity appears to be stationary over time (Fig. 3.1a).

3.6 UORO's variance increases with the number of hidden units

As the number of hidden units in the recurrent network increases, the rank one approximation that is used to provide an unbiased gradient estimate becomes coarser. Consequently, the relative variance, as defined in (3.22), should increase as the number of hidden units increases.

This increase is experimentally verified in Fig. 3.1b. Untrained GRU networks with various number of units are run for 10 timesteps, 100 times for each size, and the UORO

gradient estimate after these 10 timesteps is computed (but not applied). The relative variance of these gradients over the 100 runs is evaluated, for each network size. As shown in the figure, the relative variance increases with the number of units. Note the horizontal log scale.

The increase of the variance of the estimate with network size underlines the need for smaller learning rates when training large networks with UORO, compared to truncated backpropagation. This can imply slower learning for the kind of dependencies that truncated backpropagation can learn. The need for lower learning rates with larger networks is exemplified in Fig. 3.1c. GRU networks of various hidden sizes are trained with UORO on a simple copy task, as presented in [Hochreiter and Schmidhuber, 1997], with a lag of $T = 5$. The networks are all trained with the same decreasing learning rate, $\eta_t = \frac{10^{-4}}{1+3\cdot 10^{-3}t}$. For all network sizes except the largest, the error decreases slowly but steadily. For the largest network, the variance is too large compared to the learning rate, and the error jumps sharply midway through.

3.7 Experiments illustrating truncation bias

The set of experiments below aims at displaying specific cases where the biases from truncated BPTT are likely to prevent convergence of learning. On this test set, UORO's unbiasedness provides steady convergence, highlighting the importance of unbiased estimates for general recurrent learning.

Influence balancing. The first test case exemplifies learning of a scalar parameter θ which has a positive influence in the short term, but a negative one in the long run. Short-sightedness of truncated algorithms results in abrupt failure, with the parameter exploding in the wrong direction, even with truncation lengths exceeding the temporal dependency range by a factor of 10 or so.

Consider the linear dynamics

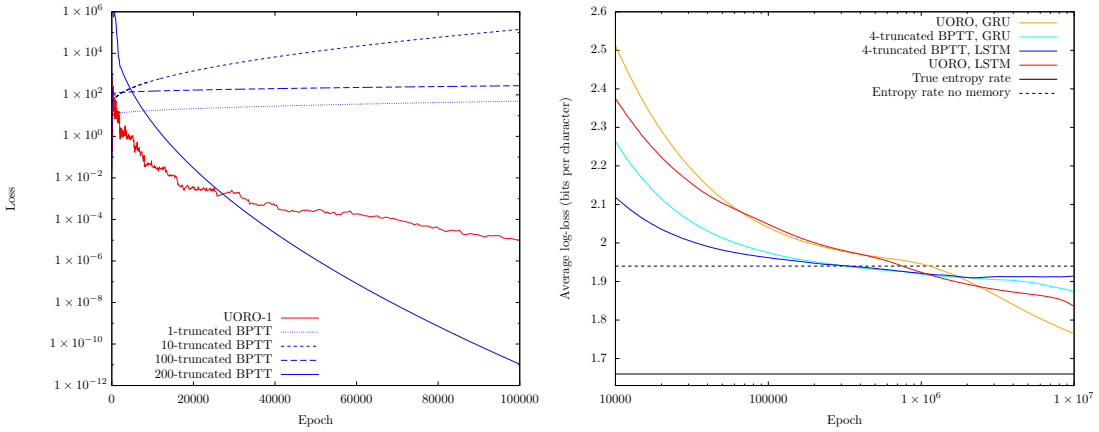
$$s_{t+1} = A\, s_t + (\theta, \dots, \theta, -\theta, \dots, -\theta)^\top \tag{3.23}$$

with A a square matrix of size n with $A_{i,i} = 1/2$, $A_{i,i+1} = 1/2$, and 0 elsewhere; $\theta \in \mathbb{R}$ is a scalar parameter. The second term has p positive-θ entries and $n-p$ negative-θ entries. Intuitively, the effect of θ on a unit diffuses to shallower units over time (Fig. 3.3a). Unit i only feels the effect of θ from unit $i+n$ after n time steps, so the intrinsic time scale of the system is $\approx n$. The loss considered is a target on the shallowest unit s^1,

$$\ell_t = \tfrac{1}{2}(s_t^1 - 1)^2. \tag{3.24}$$

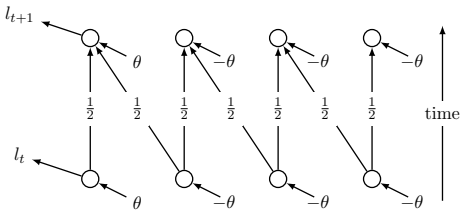
Learning is performed online with vanilla SGD, using gradient estimates either from UORO or T-truncated BPTT with various T. Learning rates are of the form $\eta_t = \frac{\eta}{1+\sqrt{t}}$ for suitable values of η.

As shown in Fig. 3.2a, UORO solves the problem while T-truncated BPTT fails to converge for any learning rate, even for truncations T largely above n. Failure is caused



(a) (b)

Figure 3.2: (a)Results for influence balancing with 23 units and 13 minus; note the vertical log scale. (b)Learning curves on distant brackets $(1, 5, 10)$.




```
[a]eecbe[a]
[j]fbfjd[j]
[c]bgddc[c]
[d]gjhai[d]
[e]iaghb[e]
[h]bigaj[h]
```

```
aaaaaa
bbbbbb
aaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbb
aaaaaaaa
bbbbbbbb
```

(a) Influence balancing, 4 units, 3 minus. (b) Distant brackets $(1, 5, 10)$. (c) $a^n b^n(1, 32)$.

Figure 3.3: Datasets.

by ill balancing of time dependencies: the influence of θ on the loss is estimated with the wrong sign due to truncation. For $n = 23$ units, with 13 minus signs, truncated BPTT requires a truncation $T \geq 200$ to converge.

Next-character prediction. The next experiment is character-level synthetic text prediction: the goal is to train a recurrent model to predict the $t + 1$-th character of a text given the first t online, with a single pass on the data sequence.

A single layer of 64 units, either GRU or LSTM, is used to output a probability vector for the next character. The cross entropy criterion is used to compute the loss. At each time t we plot the cumulated loss per character on the first t characters, $\frac{1}{t}\sum_{s=1}^{t} \ell_s$. (Losses for individual characters are quite noisy, as not all characters in the sequence are equally difficult to predict.) This would be the compression rate in bits per character if the models were used as online compression algorithms on the first t characters. In addition, in Table 3.1 we report a "recent" loss on the last $100,000$ characters, which is more representative of the model at the end of learning.

Optimization was performed using Adam with the default setting $\beta_1 = 0.9$ and

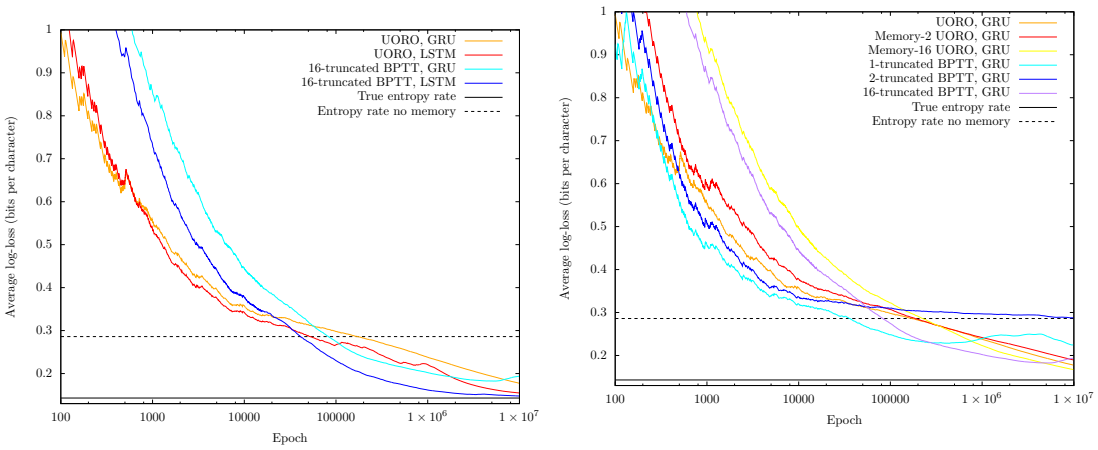


Figure 3.4: Learning curves on $a^n b^n_{(1,32)}$

$\beta_2 = 0.999$, and a decreasing learning rate $\eta_t = \frac{\gamma}{1+\alpha\sqrt{t}}$, with t the number of characters processed. As convergence of UORO requires smaller learning rates than truncated BPTT, this favors UORO. Indeed UORO can fail to converge with non-decreasing learning rates, due to its stochastic nature.

Distant brackets dataset (s, k, a)**.** The distant brackets dataset is generated by repeatedly outputting a left bracket, generating s random characters from an alphabet of size a, outputting a right bracket, generating k random characters from the same alphabet, repeating the same first s characters between brackets and finally outputting a line break. A sample is shown in Fig. 3.3b.

UORO is compared to 4-truncated BPTT. Truncation is deliberately shorter than the inherent time range of the data, to illustrate how bias can penalize learning if the inherent time range is unknown a priori. The results are given in Fig. 3.2b (with learning rates using $\alpha = 0.015$ and $\gamma = 10^{-3}$). UORO beats 4-truncated BPTT in the long run, and succeeds in reaching near optimal behaviour both with GRUs and LSTMs. Truncated BPTT remains stuck near a memoryless optimum with LSTMs; with GRUs it keeps learning, but at a slow rate. Still, truncated BPTT displays faster early convergence.

$a^n b^n(k, l)$ **dataset.** The $a^n b^n(k, l)$ dataset tests memory and counting [Gers and Schmidhuber, 2001]; it is generated by repeatedly picking a random number n between k and l, outputting a string of n a's, a line break, n b's, and a line break (see Fig. 3.3c). The difficulty lies in matching the number of a's and b's.

Plots for a few setups are given in Fig. 3.4. The learning rates used $\alpha = 0.03$ and $\gamma = 10^{-3}$. Numerical results at the end of training are given in Table 3.1. For reference, the true entropy rate is 0.14 bpc, while the entropy rate of a model that does not understand that the numbers of a's and b's coincide is double, 0.28 bpc.

Table 3.1: Averaged loss on the 10^5 last iterations on $a^n b^n(1, 32)$.

	Truncation	LSTM	GRU
UORO	No memory (default)	0.147	0.155
	Memory-2	0.149	0.174
	Memory-16	0.154	0.149
Truncated BPTT	1	0.178	0.231
	2	0.149	0.285
	16	0.144	0.207

Here, in every setup, UORO reliably converges and reaches near optimal performance. Increasing UORO's range does not significantly improve results: providing an unbiased estimate is enough to provide reliable convergence in this case. Meanwhile, truncated BPTT performs inconsistently. Notably, with GRUs, it either converges to a poor local optimum corresponding to no understanding of the temporal structure, or exhibits gradient reascent in the long run. Remarkably, with LSTMs rather than GRUs, 16-truncated BPTT reliably reaches optimal behavior on this problem even with biased gradient estimates.

Conclusion

We introduced UORO, an algorithm for training recurrent neural networks in a streaming, memoryless fashion. UORO is easy to implement, and requires as little computation time as truncated BPTT, at the cost of noise injection. Importantly, contrary to most other approaches, UORO scalably provides unbiasedness of gradient estimates. Unbiasedness is of paramount importance in the current theory of stochastic gradient descent. Furthermore, UORO is experimentally shown to benefit from its unbiasedness, converging even in cases where truncated BPTT fails to reliably achieve good results or diverges pathologically.

Algorithm 1 — One step of UORO (from time t to $t+1$)

Input

x_{t+1}, o^*_{t+1}, s_t and θ	input, target, previous recurrent state, and parameters
$\tilde{s}_t$ and $\tilde{\theta}_t$	column and row vector of size *state* and *params* such that $\mathbb{E}\,\tilde{s}_t \otimes \tilde{\theta}_t = \partial s_t / \partial \theta$
SGDOpt and η_{t+1}	stochastic optimizer and its learning rate

Output

ℓ_{t+1}, s_{t+1} and θ	loss, new recurrent state, and updated parameters
$\tilde{s}_{t+1}$ and $\tilde{\theta}_{t+1}$	such that $\mathbb{E}\,\tilde{s}_{t+1} \otimes \tilde{\theta}_{t+1} = \partial s_{t+1} / \partial \theta$
$\tilde{g}_{t+1}$	such that $\mathbb{E}\,\tilde{g}_{t+1} = \partial \ell_{t+1} / \partial \theta$

/* compute next state and loss */

$s_{t+1} \leftarrow F_{\text{state}}.\textbf{forward}(x_{t+1}, s_t, \theta), \quad o_{t+1} \leftarrow F_{\text{out}}.\textbf{forward}(x_{t+1}, s_t, \theta)$

$\ell_{t+1} \leftarrow \ell(o_{t+1}, o^*_{t+1})$

/* compute gradient estimate */

$$(\_, \delta s, \delta\theta) \leftarrow F_{\text{out}}.\textbf{backprop}\left((x_{t+1}, s_t, \theta), \frac{\partial \ell_{t+1}}{\partial o_{t+1}}\right)$$

$\tilde{g}_{t+1} \leftarrow (\delta s \cdot \tilde{s}_t)\,\tilde{\theta}_t + \delta\theta$

/* prepare for reduction */

Draw ν, column vector of random signs ± 1 of size *state*

$\tilde{s}_{t+1} \leftarrow F_{\text{state}}.\textbf{forwarddiff}((x_{t+1}, s_t, \theta), (0, \tilde{s}_t, 0))$

$(\_, \_, \delta\theta_g) \leftarrow F_{\text{state}}.\textbf{backprop}((x_{t+1}, s_t, \theta), \nu^\top)$

/* compute normalizers */

$$\rho_0 \leftarrow \sqrt{\frac{\|\tilde{\theta}_t\|}{\|\tilde{s}_{t+1}\| + \varepsilon}} + \varepsilon\,, \quad \rho_1 \leftarrow \sqrt{\frac{\|\delta\theta_g\|}{\|\nu\| + \varepsilon}} + \varepsilon \text{ with } \varepsilon = 10^{-7}$$

/* reduce */

$$\tilde{s}_{t+1} \leftarrow \rho_0\,\tilde{s}_{t+1} + \rho_1\,\nu, \quad \tilde{\theta}_{t+1} \leftarrow \frac{\tilde{\theta}_t}{\rho_0} + \frac{\delta\theta_g}{\rho_1}$$

/* update θ */

SGDOpt.$\textbf{update}(\tilde{g}_{t+1}, \eta_{t+1}, \theta)$

Chapter 4

Unbiasing Truncated Backpropagation Through Time

Foreword:

Backpropagation Through Time provides exact estimates of the gradient of the loss for general recurrent networks, but proves impractical in online settings, since it requires maintaining the full history of encountered inputs. At the other extreme Unbiased Online Recurrent Optimization provides a very coarse unbiased online estimate of the gradient, but has a very light memory footprint, only requiring to store the current datapoint. The algorithm presented in this chapter, *Anticipated Reweighted Truncated BackPropagation* (ARTBP), is a middle ground between the two: it provides potentially nearly exact gradient estimates, at the cost of maintaining arbitrarily large, finite, but of fixed average length sequences of input in memory. ARTBP was originally presented in [Tallec and Ollivier, 2017b].

Truncated Backpropagation Through Time (truncated BPTT, [Jaeger, 2002]) is a widespread method for learning recurrent computational graphs. Truncated BPTT keeps the computational benefits of *Backpropagation Through Time* (BPTT [Werbos, 1990]) while relieving the need for a complete backtrack through the whole data sequence at every step. However, truncation favors short-term dependencies: the gradient estimate of truncated BPTT is biased, so that it does not benefit from the convergence guarantees from stochastic gradient theory. We introduce *Anticipated Reweighted Truncated Backpropagation* (ARTBP), an algorithm that keeps the computational benefits of truncated BPTT, while providing unbiasedness. ARTBP works by using variable truncation lengths together with carefully chosen compensation factors in the backpropagation equation. We check the viability of ARTBP on two tasks. First, a simple synthetic task where careful balancing of temporal dependencies at different scales is needed: truncated BPTT displays unreliable performance, and in worst case scenarios, divergence, while ARTBP converges reliably. Second, on Penn Treebank character-level language modelling [Mikolov et al., 2012], ARTBP slightly outperforms truncated BPTT.

Introduction

Backpropagation Through Time (BPTT) [Werbos, 1990] is the de facto standard for training recurrent neural networks. However, BPTT has shortcomings when it comes to learning from very long sequences: learning a recurrent network with BPTT requires unfolding the network through time for as many timesteps as there are in the sequence. For long sequences this represents a heavy computational and memory load. This shortcoming is often overcome heuristically, by arbitrarily splitting the initial sequence into subsequences, and only backpropagating on the subsequences. The resulting algorithm is often referred to as *Truncated Backpropagation Through Time* (truncated BPTT, see for instance [Jaeger, 2002]). This comes at the cost of losing long term dependencies.

We introduce *Anticipated Reweighted Truncated BackPropagation* (ARTBP), a variation of truncated BPTT designed to provide an unbiased gradient estimate, accounting for long term dependencies. Like truncated BPTT, ARTBP splits the initial training sequence into subsequences, and only backpropagates on those subsequences. However, unlike truncated BPTT, ARTBP splits the training sequence into variable size subsequences, and suitably modifies the backpropagation equation to obtain unbiased gradients.

Unbiasedness of gradient estimates is the key property that provides convergence to a local optimum in stochastic gradient descent procedures. Stochastic gradient descent with biased estimates, such as the one provided by truncated BPTT, can lead to divergence even in simple situations and even with large truncation lengths (Fig. 4.3).

ARTBP is experimentally compared to truncated BPTT. On truncated BPTT failure cases, typically when balancing of temporal dependencies is key, ARTBP achieves reliable convergence thanks to unbiasedness. On small-scale but real world data, ARTBP slightly outperforms truncated BPTT on the test case we examined.

ARTBP formalizes the idea that, on a day-to-day basis, we can perform short term

optimization, but must reflect on long-term effects once in a while; ARTBP turns this into a provably unbiased overall gradient estimate. Notably, the many short subsequences allow for quick adaptation to the data, while preserving overall balance.

4.1 Related Work

BPTT [Werbos, 1990] and its truncated counterpart [Jaeger, 2002] are nearly uncontested in the recurrent learning field. Nevertheless, BPTT is hardly applicable to very long training sequences, as it requires storing and backpropagating through a network with as many layers as there are timesteps [Sutskever, 2013]. Storage issues can be partially addressed as in [Gruslys et al., 2016a], but at an increased computational cost. Backpropagating through very long sequences also implies performing fewer gradient descent steps, which significantly slows down learning [Sutskever, 2013].

Truncated BPTT heuristically solves BPTT deficiencies by chopping the initial sequence into evenly sized subsequences. Truncated BPTT truncates gradient flows between contiguous subsequences, but maintains the recurrent hidden state of the network. Truncation biases gradients, removing any theoretical convergence guarantee. Intuitively, truncated BPTT has trouble learning dependencies above the range of truncation. [1]

NoBackTrack [Ollivier et al., 2015] and *Unbiased Online Recurrent Optimization* (UORO) [Tallec and Ollivier, 2017a] both scalably provide unbiased online recurrent learning algorithms. They take the more extreme point of view of requiring memorylessness, thus forbidding truncation schemes and any storage of past states. NoBackTrack and UORO's fully online, streaming structure comes at the price of noise injection into the gradient estimates via a random rank-one reduction. ARTBP's approach to unbiasedness is radically different: ARTBP is not memoryless but does not inject artificial noise into the gradients, instead, compensating for the truncations directly inside the backpropagation equation.

4.2 Background on recurrent models

The goal of recurrent learning algorithms is to optimize a parametric dynamical system, so that its output sequence, or predictions, is as close as possible to some target sequence, known a priori. Formally, given a dynamical system with state s, inputs x, parameter θ, and transition function F,

$$s_{t+1} = F(x_{t+1}, s_t, \theta) \tag{4.1}$$

the aim is to find a θ minimizing a total loss with respect to target outputs o_t^* at each time,

$$\mathcal{L}_T = \sum_{t=1}^{T} \ell_t = \sum_{t=1}^{T} \ell(s_t, o_t^*). \tag{4.2}$$

[1] Still, as the hidden recurrent state is not reset between subsequences, it may contain hidden information about the distant past, which can be exploited [Sutskever, 2013].

A typical case is that of a standard recurrent neural network (RNN). In this case, $s_t = (o_t, h_t)$, where o_t are the activations of the output layer (encoding the predictions), and h_t are the activations of the hidden recurrent layer. For this simple RNN, the dynamical system takes the form

$$h_{t+1} = \tanh(W_x\, x_{t+1} + W_h\, h_t + b) \tag{4.3}$$
$$o_{t+1} = W_o h_{t+1} \tag{4.4}$$
$$\ell_{t+1} = \ell(o_{t+1}, o^*_{t+1}) \tag{4.5}$$

with parameters $\theta = (W_x, W_h, b)$.

Commonly, θ is optimized via a gradient descent procedure, i.e. iterating

$$\theta \leftarrow \theta - \eta \frac{\partial \mathcal{L}_T}{\partial \theta} \tag{4.6}$$

where η is the learning rate. The focus is then to efficiently compute $\partial \mathcal{L}_T / \partial \theta$.

Backpropagation through time is a method of choice to perform this computation. BPTT computes the gradient by unfolding the dynamical system through time and backpropagating through it, with each timestep corresponding to a layer. BPTT decomposes the gradient as a sum, over timesteps t, of the effect of a change of parameter at time t on all subsequent losses. Formally,

$$\frac{\partial \mathcal{L}_T}{\partial \theta} = \sum_{t=1}^{T} \delta\ell_t \, \frac{\partial F}{\partial \theta}(x_t, s_{t-1}, \theta) \tag{4.7}$$

where $\delta\ell_t := \frac{\partial \mathcal{L}_T}{\partial s_t}$ is computed backward iteratively according to the backpropagation equation

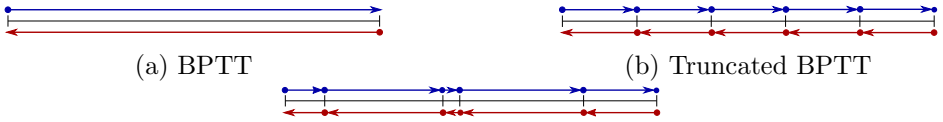
$$\begin{cases} \delta\ell_T = \dfrac{\partial \ell}{\partial s}(s_T, o^*_T) \\ \delta\ell_t = \delta\ell_{t+1} \dfrac{\partial F}{\partial s}(x_{t+1}, s_t, \theta) + \dfrac{\partial \ell}{\partial s}(s_t, o^*_t). \end{cases} \tag{4.8}$$

These backpropagation equations extend the classical ones [Jaeger, 2002], which deal with the case of a simple RNN for F.

Unfortunately, BPTT requires processing the full sequence both forward and backward. This requires maintaining the full unfolded network, or equivalently storing the full history of inputs and activations (though see [Gruslys et al., 2016a]). This is impractical when very long sequences are processed with large networks: processing the whole sequence at every gradient step slows down learning.

Practically, this is alleviated by truncating gradient flows after a fixed number of timesteps, or equivalently, splitting the input sequence into subsequences of fixed length, and only backpropagating through those subsequences. [2] This algorithm is referred to

[2] Usually the internal state s_t is maintained from one subsequence to the other, not reset to a default value.



(a) BPTT (b) Truncated BPTT

(c) ARTBP

Figure 4.1: Graphical representation of BPTT, truncated BPTT and ARTBP. Blue arrows represent forward propagations, red arrows backpropagations. Dots represent either internal state resetting or gradient resetting.

as *Truncated BPTT*. With truncation length $L < T$, the corresponding equations just drop the recurrent term $\delta\ell_{t+1}\,\frac{\partial F}{\partial s}(x_{t+1}, s_t, \theta)$ every L time steps, namely,

$$\delta\hat{\ell}_t := \begin{cases} \dfrac{\partial \ell}{\partial s}(s_t, o_t^*) & \text{if } t \text{ is a multiple of } L \\ \delta\hat{\ell}_{t+1}\,\dfrac{\partial F}{\partial s}(x_{t+1}, s_t, \theta) + \dfrac{\partial \ell}{\partial s}(s_t, o_t^*) & \text{otherwise.} \end{cases} \tag{4.9}$$

This also allows for online application: for instance, the gradient estimate from the first subsequence $t = 1 \ldots, L$ does not depend on anything at time $t > L$.

However, this gradient estimation scheme is heuristic and provides biased gradient estimates. In general the resulting gradient estimate can be quite far from the true gradient even with large truncations L (Section 4.6). Undesired behavior, and, sometimes, divergence can follow when performing gradient descent with truncated BPTT (Fig. 4.3).

4.3 Anticipated Reweighted Backpropagation Through Time: unbiasedness through reweighted stochastic truncation lengths

Like truncated BPTT, ARTBP splits the initial sequence into subsequences, and only performs backpropagation through time on subsequences. However, contrary to the latter, it does not split the sequence evenly. The length of each subsequence is sampled according to a specific probability distribution. Then the backpropagation equation is modified by introducing a suitable reweighting factor at every step to ensure unbiasedness. Figure 4.1 demonstrates the difference between BPTT, truncated BPTT and ARTBP.

Simply sampling arbitrarily long truncation lengths does not provide unbiasedness. Intuitively, it still favors short term gradient terms over long term ones. When using full BPTT, gradient computations flow back [3] from every timestep t to every timestep $t' < t$. In truncated BPTT, gradients do not flow from t to t' if $t - t'$ exceeds the truncation length. In ARTBP, since random truncations are introduced, gradient computations

[3] Gradient flows between timesteps t and t' if there are no truncations occuring between t and t'.

flow from t to t' with a certain probability, decreasing with $t - t'$. To restore balance, ARTBP rescales gradient flows by their inverse probability. Informally, if a flow has a probability p to occur, multiplication of the flow by $\frac{1}{p}$ restores balance on average.

Formally, at each training epoch, ARTBP starts by sampling a random sequence of truncation points, that is $(X_t)_{1\leq t\leq T} \in \{0,1\}^T$. A truncation will occur at all points t such that $X_t = 1$. Here X_t may have a probability law that depends on $X_1, \dots, X_{t-1}$, and also on the sequence of states $(s_t)_{1\leq t\leq T}$ of the system. The reweighting factors that ARTBP introduces in the backpropagation equation depend on these truncation probabilities. (Unbiasedness is not obtained just by global importance reweighting between the various truncated subsequences: indeed, the backpropagation equation inside each subsequence has to be modified at every time step, see (4.11).)

The question of how to choose good probability distributions for the truncation points X_t is postponed till Section 4.4. Actually, unbiasedness holds for any choice of truncation probabilities (Prop 2), but different choices for X_t lead to different variances for the resulting gradient estimates.

Proposition 2. *Let $(X_t)_{t=1\dots T}$ be any sequence of binary random variables, chosen according to probabilities*

$$c_t := \mathbb{P}(X_t = 1 \mid X_{t-1}, \dots, X_1) \tag{4.10}$$

and assume $c_t \neq 1$ for all t.

Define ARTBP to be backpropagation through time with a truncation between t and $t+1$ iff $X_t = 1$, and a compensation factor $\frac{1}{1-c_t}$ when $X_t = 0$, namely:

$$\delta\tilde{\ell}_t := \begin{cases} \dfrac{\partial \ell}{\partial s}(s_t, o_t^*) & \text{if } X_t = 1 \text{ or } t = T \\ \dfrac{1}{1-c_t}\, \delta\tilde{\ell}_{t+1}\, \dfrac{\partial F}{\partial s}(x_{t+1}, s_t, \theta) + \dfrac{\partial \ell}{\partial s}(s_t, o_t^*) & \text{otherwise.} \end{cases} \tag{4.11}$$

Let $\tilde{g}$ be the gradient estimate obtained by using $\delta\tilde{\ell}_t$ instead of $\delta\ell_t$ in ordinary BPTT (4.7), namely

$$\tilde{g} := \sum_{t=1}^{T} \delta\tilde{\ell}_t\, \frac{\partial F}{\partial \theta}(x_t, s_{t-1}, \theta) \tag{4.12}$$

Then, on average over the ARTBP truncations, this is an unbiased gradient estimate of the total loss:

$$\mathbb{E}_{X_1,\dots,X_T}[\,\tilde{g}\,] = \frac{\partial \mathcal{L}_T}{\partial \theta}. \tag{4.13}$$

The core of the proof is as follows: With probability c_t (truncation), $\delta\tilde{\ell}_{t+1}$ does not contribute to $\delta\tilde{\ell}_t$. With probability $1 - c_t$ (no truncation), it contributes with a factor $\frac{1}{1-c_t}$. So on average, $\delta\tilde{\ell}_{t+1}$ contributes to $\delta\tilde{\ell}_t$ with a factor 1, and ARTBP (4.11) reduces to standard, non-truncated BPTT (4.8) on average. The detailed proof is given in Section 4.7.

While the ARTBP gradient estimate above is unbiased, some noise is introduced due to stochasticity of the truncation points. It turns out that ARTBP trades off memory consumption (larger truncation lengths) for variance, as we now discuss.

4.4 Choice of c_t and memory/variance tradeoff

ARTBP requires specifying the probability c_t of truncating at time t given previous truncations. Intuitively the c's regulate the average truncation lengths. For instance, with a constant $c_t \equiv c$, the lengths of the subsequences between two truncations follow a geometric distribution, with average truncation length $\frac{1}{c}$. Truncated BPTT with fixed truncation length L and ARTBP with fixed $c = \frac{1}{L}$ are thus comparable memorywise.

Small values of c_t will lead to long subsequences and gradients closer to the exact value, while large values will lead to shorter subsequences but larger compensation factors $\frac{1}{1-c_t}$ and noisier estimates. In particular, the product of the $\frac{1}{1-c_t}$ factors inside a subsequence can grow quickly. For instance, a constant c_t leads to exponential growth of the cumulated $\frac{1}{1-c_t}$ factors when iterating (4.11).

To mitigate this effect, we suggest to set c_t to values such that the probability to have a subsequence of length L decreases like $L^{-\alpha}$. The variance of the lengths of the subsequences will be finite if $\alpha > 3$. Moreover we might want to control the average truncation length L_0. This is achieved via

$$c_t = \mathbb{P}(X_t = 1 \mid X_{t-1}, \dots, X_1) = \frac{\alpha - 1}{(\alpha - 2)L_0 + \delta t} \tag{4.14}$$

where δt is the time elapsed since the last truncation, $\delta t = t - \sup\{s \mid s < t, X_s = 1\}$. Intuitively, the more time spent without truncating, the lower the probability to truncate. This formula is chosen such that the average truncation length is approximately L_0, and the standard deviation from this average length is finite. The parameter α controls the regularity of the distribution of truncation lengths: all moments lower than $\alpha - 1$ are finite, the others are infinite. With larger α, large lengths will be less frequent, but the compensating factors $\frac{1}{1-c_t}$ will be larger.

With this choice of c_t, the product of the $\frac{1}{1-c_t}$ factors incurred by backpropagation inside each subsequence grows polynomially like $L^{\alpha-1}$ in a subsequence of length L. If the dynamical system has geometrically decaying memory, i.e., if the operator norm of the transition operator $\frac{\partial F}{\partial s}$ is less than $1 - \varepsilon$ most of the time, then the value of $\delta\tilde{\ell}_t$ will stay controlled, since $(1 - \varepsilon)^L \cdot L^\alpha$ stays bounded. On the other hand, using a constant $c_t \equiv c$ provides bounded $\delta\tilde{\ell}_t$ only for small values $c < \varepsilon$.

In the experiments below, we use the c_t from (4.14) with $\alpha = 4$ or $\alpha = 6$.

4.5 Online implementation

Importantly, ARTBP can be directly applied online, thus providing unbiased gradient estimates for recurrent networks.

Indeed, not all truncation points have to be drawn in advance: ARTBP can be applied by sampling the first truncation point, performing both forward and backward passes of BPTT up until this point, and applying a partial gradient descent update based on the resulting gradient on this subsequence. Then one moves to the next subsequence and the next truncation point, etc. (Fig. 4.1c).

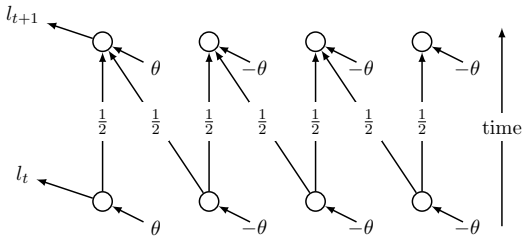


Figure 4.2: Influence balancing dynamics, 1 positive influence, 3 negative influences.

4.6 Experimental validation

The experimental setup below aims both at illustrating the theoretical properties of ARTBP compared to truncated BPTT, and at testing the soundness of ARTBP on real world data.

4.6.1 Influence balancing

The influence balancing experiment is a synthetic example demonstrating, in a very simple model, the importance of being unbiased. Intuitively, a parameter has a positive short term influence, but a negative long term one that surpasses the short term effect. Practically, we consider a row of agents, numbered from left to right from 1 to $p+n$ who, at each time step, are provided with a signal depending on the parameter, and diffuse part of their current state to the agent directly to their left. The p leftmost agents receive a positive signal at each time step, and the n rightmost agents a negative signal. The training goal is to control the state of the leftmost agent. The first p agents contribute positively to the first agent state, while the next n contribute negatively. However, agent 1 only feels the contribution from agent k after k timesteps. If optimization is blind to dependencies above k, the effect of k is never felt. A typical instantiation of such a problem would be that of a drug whose effect varies after various delays; the parameter to be optimized is the quantity of drug to be used daily.

Such a model can be formalized as [Tallec and Ollivier, 2017a]

$$s_{t+1} = A\, s_t + (\theta, \dots, \theta, -\theta, \dots, -\theta)^\top \tag{4.15}$$

with A a square matrix of size $p+n$ with $A_{k,k} = 1/2$, $A_{k,k+1} = 1/2$, and 0 elsewhere; s_t^k corresponds to the state of the k-th agent. $\theta \in \mathbb{R}$ is a scalar parameter corresponding to the intensity of the signal observed at each time step. The right-hand-side has p positive-θ entries and n negative-θ entries. The loss considered is an arbitrary target on the leftmost agent s^1,

$$\ell_t = \tfrac{1}{2}(s_t^1 - 1)^2. \tag{4.16}$$

The dynamics is illustrated schematically in Figure 4.2.

Fixed-truncation BPTT is experimentally compared with ARTBP for this problem. The setting is online: starting at $t = 1$, a first truncation length L is selected (fixed

for BPTT, variable for ARTBP), forward and backward passes are performed on the subsequence $t = 1, \ldots, L$, a vanilla gradient step is performed with the resulting gradient estimate, then the procedure is repeated with the next subsequence starting at $t = L+1$, etc..

Our experiment uses $p = 10$ and $n = 13$, so that after 23 steps the signal should have had time to travel through the network. Truncated BPTT is tested with various truncations $L = 10, 100, 200$. (As the initial θ is fixed, truncated BPTT is deterministic in this experiment, thus we only provide a single run for each L.) ARTBP is tested with the probabilities (4.14) using $L_0 = 16$ (average truncation length) and $\alpha = 6$. ARTBP is stochastic: five random runs are provided to test reliability of convergence.

The results are displayed in Fig. 4.3. We used decreasing learning rates $\eta_t = \frac{\eta_0}{\sqrt{1+t}}$ where $\eta_0 = 3 \times 10^{-4}$ is the initial learning rate and t is the timestep. We plot the average loss over timesteps 1 to t, as a function of t.

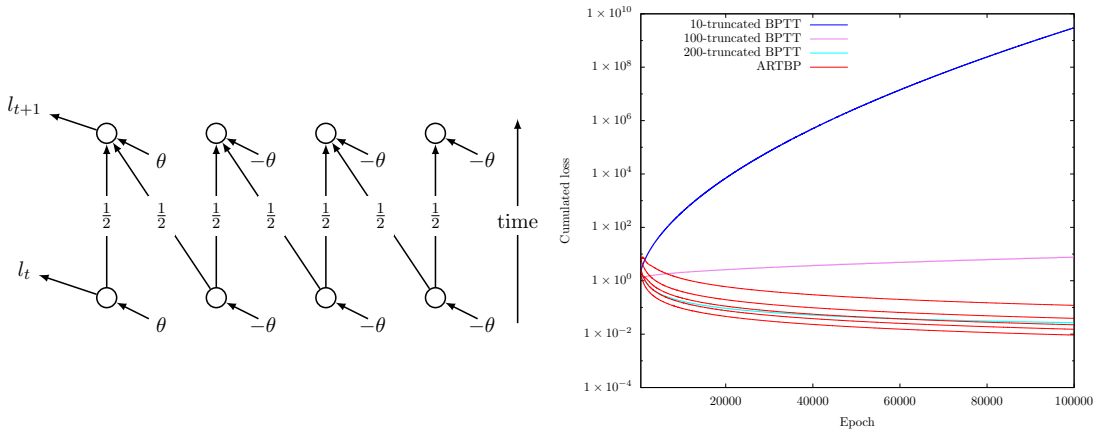
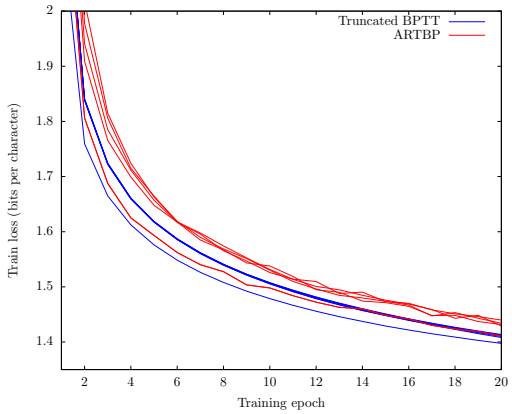


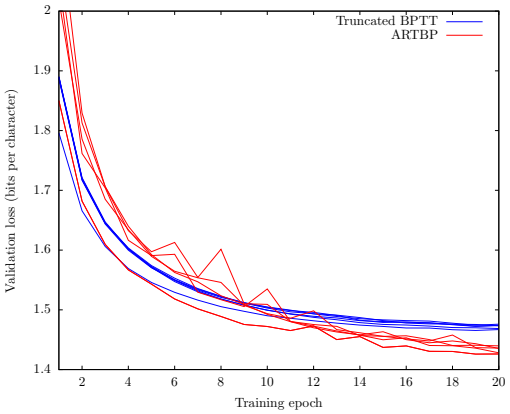
Figure 4.3: ARTBP and truncated BPTT on influence balancing, $n = 13$, $p = 10$. Note the log scale on the y-axis.

Truncated BPTT diverges even for truncation ranges largely above the intrinsic temporal scale of the system. This is an expected result: due to bias, truncated BPTT ill-balances temporal dependencies and estimates the overall gradient with a wrong sign. In particular, reducing the learning rate will *not* prevent divergence. On the other hand, ARTBP reliably converges on every run.

Note that for the largest truncation $L = 200$, truncated BPTT finally converges, and does so at a faster rate than ARTBP. This is because this particular problem is deterministic, so that a deterministic gradient scheme will converge (if it does converge) geometrically like $O(e^{-\lambda t})$, whereas ARTBP is stochastic due to randomization of truncations, and so will not converge faster than $O(t^{-1/2})$. This difference would disappear, for instance, with noisy targets or a noisy system.



(a) Learning curves on Penn Treebank train set.



(b) Learning curves on Penn Treebank validation set.

Figure 4.4: Results on Penn Treebank character-level language modelling.

4.6.2 Character-level Penn Treebank language model.

We compare ARTBP to truncated BPTT on the character-level version of the Penn Treebank dataset, a standard set of case-insensitive, punctuation-free English text [Marcus et al., 1993]. Character-level language modelling is a common benchmark for recurrent models.

The dataset is split into training, validation and test sets following [Mikolov et al., 2012]. Both ARTBP and truncated BPTT are used to train an LSTM model [Hochreiter and Schmidhuber, 1997] with a softmax classifier on its hidden state, on the character prediction task. The training set is batched into 64 subsets processed in parallel to increase computing speed. Before each full pass on the training set, the batched training sequences are split into subsequences:

- for truncated BPTT, of fixed size 50;
- for ARTBP, at random following the scheme (4.14) with $\alpha = 4$ and $L_0 = 50$.

Truncated BPTT and ARTBP process these subsequences sequentially, [4] as in Fig. 4.1. The parameter is updated after each subsequence, using the Adam [Kingma and Ba, 2014] stochastic gradient scheme, with learning rate 10^{-4}. The biases of the LSTM unit forget gates are set to 2, to prevent early vanishing gradients [Gers et al., 2000]. Results (in bits per character, bpc) are displayed in Fig. 4.4. Six randomly sampled runs are plotted, to test reliability.

In this test, ARTBP slightly outperforms truncated BPTT in terms of validation and test error, while the reverse is true for the training error (Fig. 4.4).

[4] Subsequences are not shuffled, as we do not reset the internal state of the network between subsequences.

Even with ordinary truncated BPTT, we could not reproduce reported state of the art results, and do somewhat worse. We reach a test error of 1.43 bpc with standard truncated BPTT and 1.40 bpc with ARTBP, while reported values with similar LSTM models range from 1.38 bpc [Cooijmans et al., 2016a] to 1.26 bpc [Graves, 2013] (the latter with a different test/train split). This may be due to differences in the experimental setup: we have applied truncated BPTT without subsequence shuffling or gradient clipping [Graves, 2013] (incidentally, both would break unbiasedness). Arguably, the numerical issues solved by gradient clipping are model specific, not algorithm specific, while the point here was to compare ARTBP to truncated BPTT for a given model.

Conclusion

We have shown that the bias introduced by truncation in the backpropagation through time algorithm can be compensated by the simple mathematical trick of randomizing the truncation points and introducing compensation factors in the backpropagation equation. The algorithm is experimentally viable, and provides proper balancing of the effects of different time scales when training recurrent models.

4.7 Proof of Proposition 2

First, by backward induction, we show that for all $t \leq T$, for all $x_1, \ldots, x_{t-1} \in \{0, 1\}$,

$$\mathbb{E}\left[\delta\tilde{\ell}_t \mid X_{1:t-1} = x_{1:t-1}\right] = \delta\ell_t \tag{4.17}$$

where $\delta\ell_t$ is the value obtained by ordinary BPTT (4.8). Here $x_{1:k}$ is short for $(x_1, \ldots, x_k)$.

For $t = T$, this holds by definition: $\delta\tilde{\ell}_T = \frac{\partial \ell}{\partial s}(s_T, o_T^*) = \delta\ell_T$.

Assume that the induction hypothesis (4.17) holds at time $t+1$. Note that the values s_t do not depend on the random variables X_t, as they are computed during the forward pass of the algorithm. In particular, the various derivatives of F and ℓ in (4.11) do not depend on $X_{1:T}$.

Thus

$$\begin{aligned}
&\mathbb{E}\left[\delta\tilde{\ell}_t \mid X_{1:t-1} = x_{1:t-1}\right] = \\
&\quad \Pr(X_t = 1 \mid X_{1:t-1} = x_{1:t-1})\, \mathbb{E}\left[\delta\tilde{\ell}_t \mid X_{1:t-1} = x_{1:t-1}, X_t = 1\right] + && (4.18)\\
&\qquad \Pr(X_t = 0 \mid X_{1:t-1} = x_{1:t-1})\, \mathbb{E}\left[\delta\tilde{\ell}_t \mid X_{1:t-1} = x_{1:t-1}, X_t = 0\right] && (4.19)\\
&= c_t\, \mathbb{E}\left[\delta\tilde{\ell}_t \mid X_{1:t-1} = x_{1:t-1}, X_t = 1\right] + (1 - c_t)\, \mathbb{E}\left[\delta\tilde{\ell}_t \mid X_{1:t-1} = x_{1:t-1}, X_t = 0\right] && (4.20)
\end{aligned}$$

If $X_t = 1$ then $\delta\tilde{\ell}_t = \frac{\partial \ell}{\partial s}(s_t, o_t^*)$. If $X_t = 0$, then $\delta\tilde{\ell}_t = \frac{\partial \ell}{\partial s}(s_t, o_t^*) + \frac{1}{1-c_t}\, \delta\tilde{\ell}_{t+1}\, \frac{\partial F}{\partial s}(x_{t+1}, s_t, \theta)$. Therefore, substituting into (4.20),

$$\mathbb{E}\left[\delta\tilde{\ell}_t \mid X_{1:t-1} = x_{1:t-1}\right] = \frac{\partial \ell}{\partial s}(s_t, o_t^*) + \mathbb{E}\left[\delta\tilde{\ell}_{t+1} \mid X_{1:t-1} = x_{1:t-1}, X_t = 0\right] \frac{\partial F}{\partial s}(x_{t+1}, s_t, \theta) \tag{4.21}$$

but by the induction hypothesis at time $t+1$, this is exactly $\frac{\partial \ell}{\partial s}(s_t, o_t^*) + \delta\ell_{t+1}\frac{\partial F}{\partial s}(x_{t+1}, s_t, \theta)$, which is $\delta\ell_t$.

Therefore, $\mathbb{E}\left[\delta\tilde{\ell}_t\right] = \delta\ell_t$ unconditionally. Plugging the $\delta\tilde{\ell}$'s into (4.7), and averaging

$$\mathbb{E}_{X_1,\dots,X_T}\left[\,\tilde{g}\,\right] = \sum_{t=1}^{T} \mathbb{E}_{X_t,\dots,X_T}\left[\delta\tilde{\ell}_t\right] \frac{\partial F}{\partial \theta}(x_t, s_{t-1}, \theta) \tag{4.22}$$

$$= \sum_{t=1}^{T} \delta\ell_t \frac{\partial F}{\partial \theta}(x_t, s_{t-1}, \theta) \tag{4.23}$$

$$= \frac{\partial \mathcal{L}_T}{\partial \theta} \tag{4.24}$$

which ends the proof.

Chapter 5

Can Recurrent Neural Networks Warp Time?

Foreword:

While providing unbiased gradients for recurrent networks ensures that sufficiently expressive models could potentially learn arbitrarily long term dependencies, an orthogonal problem is to design architectures that facilitate learning of such dependencies. The following chapter presents the content of the article *Can recurrent Neural Networks Warp* ([Tallec and Ollivier, 2018]), originally presented at ICLR 2018. [Tallec and Ollivier, 2018] gives a new take on what makes for a successful recurrent architecture when it comes to long term dependencies. More precisely, we relate the success of gated recurrent architectures to their invariance property when faced with time transformations.

Successful recurrent models such as long short-term memories (LSTMs) and gated recurrent units (GRUs) use *ad hoc* gating mechanisms. Empirically these models have been found to improve the learning of medium to long term temporal dependencies and to help with vanishing gradient issues.

We prove that learnable gates in a recurrent model formally provide *quasi-invariance to general time transformations* in the input data. We recover part of the LSTM architecture from a simple axiomatic approach.

This result leads to a new way of initializing gate biases in LSTMs and GRUs. Experimentally, this new *chrono initialization* is shown to greatly improve learning of long term dependencies, with minimal implementation effort.

Introduction

Recurrent neural networks (e.g. [Jaeger, 2002]) are a standard machine learning tool to model and represent temporal data; mathematically they amount to learning the parameters of a parameterized dynamical system so that its behavior optimizes some criterion, such as the prediction of the next data in a sequence.

Handling long term dependencies in temporal data has been a classical issue in the learning of recurrent networks. Indeed, stability of a dynamical system comes at the price of exponential decay of the gradient signals used for learning, a dilemma known as the *vanishing gradient* problem [Pascanu et al., 2012, Hochreiter, 1991, Bengio et al., 1994]. This has led to the introduction of recurrent models specifically engineered to help with such phenomena.

Use of feedback connections [Hochreiter and Schmidhuber, 1997] and control of feedback weights through gating mechanisms [Gers et al., 1999] partly alleviate the vanishing gradient problem. The resulting architectures, namely long short-term memories (LSTMs [Hochreiter and Schmidhuber, 1997, Gers et al., 1999]) and gated recurrent units (GRUs [Chung et al., 2014]) have become a standard for treating sequential data.

Using orthogonal weight matrices is another proposed solution to the vanishing gradient problem, thoroughly studied in [Saxe et al., 2013, Le et al., 2015, Arjovsky et al., 2016, Wisdom et al., 2016b, Henaff et al., 2016]. This comes with either computational overhead, or limitation in representational power. Furthermore, restricting the weight matrices to the set of orthogonal matrices makes forgetting of useless information difficult.

The contribution of this paper is threefold:

- We show that postulating invariance to time transformations in the data (taking invariance to time warping as an axiom) necessarily leads to a gate-like mechanism in recurrent models (Section 5.1). This provides a clean derivation of part of the popular LSTM and GRU architectures from first principles. In this framework, gate values appear as time contraction or time dilation coefficients, similar in spirit to the notion of time constant introduced in [Mozer, 1992].

- From these insights, we provide precise prescriptions on how to initialize gate biases

(Section 5.2) depending on the range of time dependencies to be captured. It has previously been advocated that setting the bias of the forget gate of LSTMs to 1 or 2 provides overall good performance [Gers and Schmidhuber, 2000, Jozefowicz et al., 2015]. The viewpoint here explains why this is reasonable in most cases, when facing medium term dependencies, but fails when facing long to very long term dependencies.

- We test the empirical benefits of the new initialization on both synthetic and real world data (Section 5.3). We observe substantial improvement with long-term dependencies, and slight gains or no change when short-term dependencies dominate.

5.1 From time warping invariance to gating

When tackling sequential learning problems, being resilient to a change in time scale is crucial. Lack of resilience to time rescaling implies that we can make a problem arbitrarily difficult simply by changing the unit of measurement of time. Ordinary recurrent neural networks are highly non-resilient to time rescaling: a task can be rendered impossible for an ordinary recurrent neural network to learn, simply by inserting a fixed, small number of zeros or whitespaces between all elements of the input sequence. An explanation is that, with a given number of recurrent units, the class of functions representable by an ordinary recurrent network is not invariant to time rescaling.

Ideally, one would like a recurrent model to be able to learn from time-warped input data $x(c(t))$ as easily as it learns from data $x(t)$, at least if the time warping $c(t)$ is not overly complex. The change of time c may represent not only time rescalings, but, for instance, accelerations or decelerations of the phenomena in the input data.

We call a class of models *invariant to time warping*, if for any model in the class with input data $x(t)$, and for any time warping $c(t)$, there is another (or the same) model in the class that behaves on data $x(c(t))$ in the same way the original model behaves on $x(t)$. (In practice, this will only be possible if the warping c is not too complex.) We will show that this is deeply linked to having gating mechanisms in the model.

Invariance to time rescaling

Let us first discuss the simpler case of a linear time rescaling. Formally, this is a linear transformation of time, that is

$$\begin{aligned} c\colon \mathbb{R}_+ &\longrightarrow \mathbb{R}_+ \\ t &\longmapsto \alpha t \end{aligned} \tag{5.1}$$

with $\alpha > 0$. For instance, receiving a new input character every 10 time steps only, would correspond to $\alpha = 0.1$.

Studying time transformations is easier in the continuous-time setting. The discrete time equation of a basic recurrent network with hidden state h_t,

$$h_{t+1} = \tanh\left(W_x\, x_t + W_h\, h_t + b\right) \tag{5.2}$$

can be seen as a time-discretized version of the continuous-time equation[1]

$$\frac{\mathrm{d}h(t)}{\mathrm{d}t} = \tanh\left(W_x\, x(t) + W_h\, h(t) + b\right) - h(t) \tag{5.3}$$

namely, (5.2) is the Taylor expansion $h(t+\delta t) \approx h(t) + \delta t\, \frac{\mathrm{d}h(t)}{\mathrm{d}t}$ with discretization step $\delta t = 1$.

Now imagine that we want to describe time-rescaled data $x(\alpha t)$ with a model from the same class. Substituting $t \leftarrow c(t) = \alpha t$, $x(t) \leftarrow x(\alpha t)$ and $h(t) \leftarrow h(\alpha t)$ and rewriting (5.3) in terms of the new variables, the time-rescaled model satisfies[2]

$$\frac{\mathrm{d}h(t)}{\mathrm{d}t} = \alpha\, \tanh\left(W_x\, x(t) + W_h\, h(t) + b\right) - \alpha\, h(t). \tag{5.4}$$

However, when translated back to a discrete-time model, this no longer describes an ordinary RNN but a *leaky* RNN [Jaeger, 2002, §8.1]. Indeed, taking the Taylor expansion of $h(t+\delta t)$ with $\delta t = 1$ in (5.4) yields the recurrent model

$$h_{t+1} = \alpha \tanh\left(W_x\, x_t + W_h\, h_t + b\right) + (1-\alpha) h_t \tag{5.5}$$

Thus, a straightforward way to ensure that a class of (continuous-time) models is able to represent input data $x(\alpha t)$ in the same way that it can represent input data $x(t)$, is to take a leaky model in which $\alpha > 0$ is a learnable parameter, corresponding to the coefficient of the time rescaling. Namely, the class of ordinary recurrent networks is not invariant to time rescaling, while the class of leaky RNNs (5.5) is.

Learning α amounts to learning the global characteristic timescale of the problem at hand. More precisely, $1/\alpha$ ought to be interpreted as the characteristic forgetting time of the neural network.[3]

Invariance to time warpings

In all generality, we would like recurrent networks to be resilient not only to time rescaling, but to all sorts of time transformations of the inputs, such as variable accelerations or decelerations.

An eligible time transformation, or *time warping*, is any increasing differentiable function c from $\mathbb{R}_+$ to $\mathbb{R}_+$. This amounts to facing input data $x(c(t))$ instead of $x(t)$. Applying a time warping $t \leftarrow c(t)$ to the model and data in equation (5.3) and reasoning as above yields

$$\frac{\mathrm{d}h(t)}{\mathrm{d}t} = \frac{\mathrm{d}c(t)}{\mathrm{d}t} \tanh\left(W_x\, x(t) + W_h\, h(t) + b\right) - \frac{\mathrm{d}c(t)}{\mathrm{d}t}\, h(t). \tag{5.6}$$

[1] We will use indices h_t for discrete time and brackets $h(t)$ for continuous time.

[2] More precisely, introduce a new time variable T and set the model and data with variable T to $H(T) := h(c(T))$ and $X(T) := x(c(T))$. Then compute $\frac{\mathrm{d}H(T)}{\mathrm{d}T}$. Then rename H to h, X to x and T to t to match the original notation.

[3] Namely, in the "free" regime if inputs stop after a certain time t_0, $x(t) = 0$ for $t > t_0$, with $b = 0$ and $W_h = 0$, the solution of (5.4) is $h(t) = e^{-\alpha\,(t-t_0)} h(t_0)$, and so the network retains information from the past $t < t_0$ during a time proportional to $1/\alpha$.

Ideally, one would like a model to be able to learn from input data $x(c(t))$ as easily as it learns from data $x(t)$, at least if the time warping $c(t)$ is not overly complex.

To be invariant to time warpings, a class of (continuous-time) models has to be able to represent Equation (5.6) for any time warping $c(t)$. Moreover, the time warping is unknown a priori, so would have to be learned.

Ordinary recurrent networks do not constitute a model class that is invariant to time rescalings, as seen above. A fortiori, this model class is not invariant to time warpings either.

For time warping invariance, one has to introduce a *learnable* function g that will represent the derivative[4] of the time warping, $\frac{\mathrm{d}c(t)}{\mathrm{d}t}$ in (5.6). For instance g may be a recurrent neural network taking the x's as input.[5] Thus we get a class of recurrent networks defined by the equation

$$\frac{\mathrm{d}h(t)}{\mathrm{d}t} = g(t) \tanh\left(W_x\, x(t) + W_h\, h(t) + b\right) - g(t)\, h(t) \tag{5.7}$$

where g belongs to a large class (universal approximator) of functions of the inputs.

The class of recurrent models (5.7) is *quasi*-invariant to time warpings. The quality of the invariance will depend on the learning power of the learnable function g: a function g that can represent any function of the data would define a class of recurrent models that is perfectly invariant to time warpings; however, a specific model for g (e.g., neural networks of a given size) can only represent a specific, albeit large, class of time warpings, and so will only provide quasi-invariance.

Heuristically, $g(t)$ acts as a time-dependent version of the fixed α in (5.4). Just like $1/\alpha$ above, $1/g(t_0)$ represents the local forgetting time of the network at time t_0: the network will effectively retain information about the inputs at t_0 for a duration of the order of magnitude of $1/g(t_0)$ (assuming $g(t)$ does not change too much around t_0).

Let us translate back this equation to the more computationally realistic case of discrete time, using a Taylor expansion with step size $\delta t = 1$, so that $\frac{\mathrm{d}h(t)}{\mathrm{d}t} = \cdots$ becomes $h_{t+1} = h_t + \cdots$. Then the model (5.7) becomes

$$h_{t+1} = g_t \tanh\left(W_x\, x_t + W_h\, h_t + b\right) + (1 - g_t)\, h_t. \tag{5.8}$$

where g_t itself is a function of the inputs.

This model is the simplest extension of the RNN model that provides invariance to time warpings.[6] It is a basic gated recurrent network, with input gating g_t and forget

[4] It is, of course, algebraically equivalent to introduce a function g that learns the derivative of c, or to introduce a function G that learns c. However, only the derivative of c appears in (5.6). Therefore the choice to work with $\frac{\mathrm{d}c(t)}{\mathrm{d}t}$ is more convenient. Moreover, it may also make learning easier, because the simplest case of a time warping is a time rescaling, for which $\frac{\mathrm{d}c(t)}{\mathrm{d}t} = \alpha$ is a constant. Time warpings c are increasing by definition: this translates as $g > 0$.

[5] The time warping has to be learned only based on the data seen so far.

[6] Even more: the weights (W_x, W_h, b) are the same for $h(t)$ in (5.3) and $h(c(t))$ in (5.6). This means that *in principle it is not necessary to re-train the model for the time-warped data.* (Assuming, of course, that g_t can learn the time warping efficiently.) The variable copy task (Section 5.3) arguably illustrates this. So the definition of time warping invariance could be strengthened to use the *same* model before and after warping.

gating $(1 - g_t)$.

Here g_t has to be able to learn an arbitrary function of the past inputs x; for instance, take for g_t the output of a recurrent network with hidden state h^g:

$$g_t = \sigma(W_{gx}\, x_t + W_{gh}\, h_t^g + b_g) \tag{5.9}$$

with sigmoid activation function σ (more on the choice of sigmoid below). Current architectures just reuse for h^g the states h of the main network (or, equivalently, relabel $h \leftarrow (h, h^g)$ to be the union of both recurrent networks and do not make the distinction).

The model (5.8) provides invariance to *global* time warpings, making all units face the same dilation/contraction of time. One might, instead, endow every unit i with its own local contraction/dilation function g^i. This offers more flexibility (gates have been introduced for several reasons beyond time warpings [Hochreiter, 1991]), especially if several unknown timescales coexist in the signal: for instance, in a multilayer model, each layer may have its own characteristic timescales corresponding to different levels of abstraction from the signal. This yields a model

$$h_{t+1}^i = g_t^i \tanh\left(W_x^i\, x_t + W_h^i\, h_t + b^i\right) + (1 - g_t^i)\, h_t^i \tag{5.10}$$

with h^i and (W_x^i, W_h^i, b^i) being respectively the activation and the incoming parameters of unit i, and with each g^i a function of both inputs and units.

Equation 5.10 defines a simple form of gated recurrent network, that closely resembles the evolution equation of *cell* units in LSTMs, and of hidden units in GRUs.

In (5.10), the forget gate is tied to the input gate (g_t^i and $1 - g_t^i$). Such a setting has been successfully used before (e.g. [Lample et al., 2016]) and saves some parameters, but we are not aware of systematic comparisons. Below, we *initialize* LSTMs this way but do not enforce the constraint throughout training.

Continuous time versus discrete time

Of course, the analogy between continuous and discrete time breaks down if the Taylor expansion is not valid. The Taylor expansion is valid when the derivative of the time warping is not too large, say, when $\alpha \lesssim 1$ or $g_t \lesssim 1$ (then (5.8) and (5.7) are close). Intuitively, for continuous-time data, the physical time increment corresponding to each time step $t \to t+1$ of the discrete-time recurrent model should be smaller than the speed at which the data changes, otherwise the situation is hopeless. So discrete-time gated models are invariant to time warpings that stretch time (such as interspersing the data with blanks or having long-term dependencies), but obviously not to those that make things happen too fast for the model.

Besides, since time warpings are monotonous, we have $\frac{\mathrm{d}c(t)}{\mathrm{d}t} > 0$, i.e., $g_t > 0$. The two constraints $g_t > 0$ and $g_t < 1$ square nicely with the use of a sigmoid for the gate function g.

5.2 Time warpings and gate initialization

If we happen to know that the sequential data we are facing have temporal dependencies in an approximate range $[T_{\min}, T_{\max}]$, it seems reasonable to use a model with memory (forgetting time) lying approximately in the same temporal range. As mentioned in Section 5.1, this amounts to having values of g in the range $\left[\frac{1}{T_{\max}}, \frac{1}{T_{\min}}\right]$.

The biases b_g of the gates g greatly impact the order of magnitude of the values of $g(t)$ over time. If the values of both inputs and hidden layers are centered over time, $g(t)$ will typically take values centered around $\sigma(b_g)$. Values of $\sigma(b_g)$ in the desired range $\left[\frac{1}{T_{\max}}, \frac{1}{T_{\min}}\right]$ are obtained by choosing the biases b_g between $-\log(T_{\max}-1)$ and $-\log(T_{\min}-1)$. This is a loose prescription: we only want to control the order of magnitude of the memory range of the neural networks. Furthermore, we don't want to bound $g(t)$ too tightly to some value forever: if rare events occur, abruplty changing the time scale can be useful. Therefore we suggest to use these values as initial values only.

This suggests a practical initialization for the bias of the gates of recurrent networks such as (5.10): when characteristic timescales of the sequential data at hand are expected to lie between $T_{\min}$ and $T_{\max}$, initialize the biases of g as $-\log(\mathcal{U}([T_{\min}, T_{\max}])-1)$ where $\mathcal{U}$ is the uniform distribution[7].

For LSTMs, using a variant of [Graves et al., 2013]:

$$i_t = \sigma(W_{xi}\, x_t + W_{hi}\, h_{t-1} + b_i) \tag{5.11}$$
$$f_t = \sigma(W_{xf}\, x_t + W_{hf}\, h_{t-1} + b_f) \tag{5.12}$$
$$c_t = f_t\, c_{t-1} + i_t\, \tanh(W_{xc}\, x_t + W_{hc}\, h_{t-1} + b_c) \tag{5.13}$$
$$o_t = \sigma(W_{xo}\, x_t + W_{ho}\, h_{t-1} + b_o) \tag{5.14}$$
$$h_t = o_t\, \tanh(c_t), \tag{5.15}$$

the correspondence between between the gates in (5.10) and those in (5.13) is as follows: $1-g_t$ corresponds to f_t, and g_t to i_t. To obtain a time range around T for unit i, we must both ensure that f_t^i lies around $1-1/T$, and that i_t lies around $1/T$. When facing time dependencies with largest time range $T_{\max}$, this suggests to initialize LSTM gate biases to

$$\begin{aligned} b_f &\sim \log(\mathcal{U}([1, T_{\max}-1])) \\ b_i &= -b_f \end{aligned} \tag{5.16}$$

with $\mathcal{U}$ the uniform distribution and $T_{\max}$ the expected range of long-term dependencies to be captured.

Hereafter, we refer to this as the *chrono initialization*.

[7] When the characteristic timescales of the sequential data at hand are completetly unknown, a possibility is to draw, for each gate, a random time range T according to some probability distribution on $\mathbb{N}$ with slow decay (such as $\mathbb{P}(T=k) \propto \frac{1}{k\log(k+1)^2}$) and initialize biases to $\log(T)$.

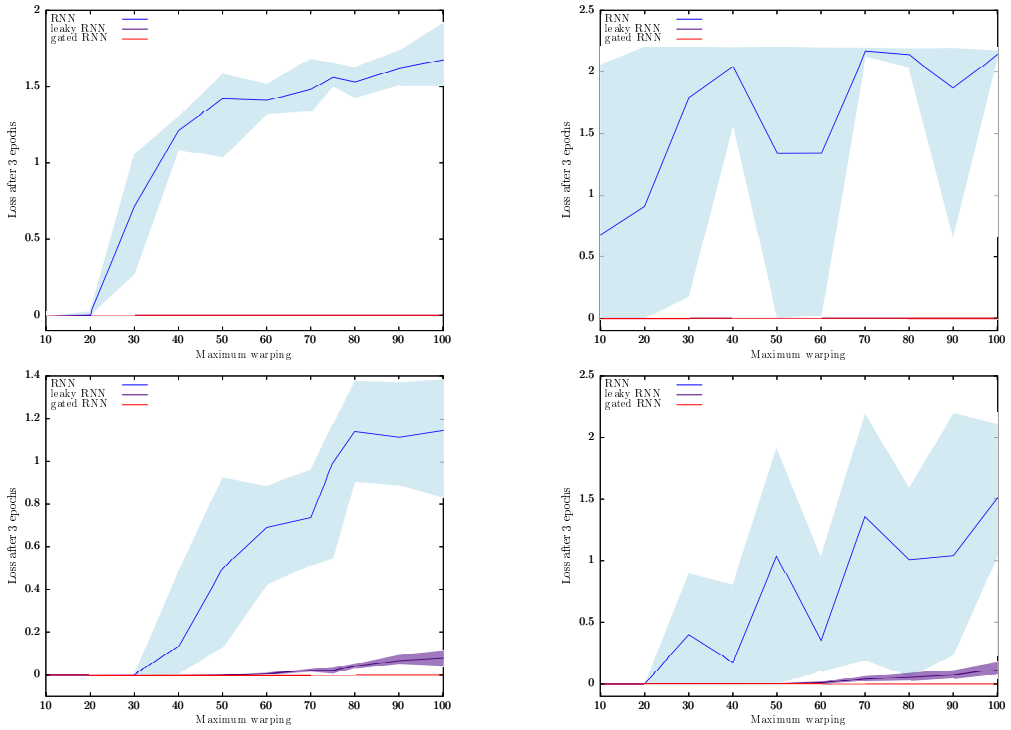


Figure 5.1: Performance of different recurrent architectures on warped and padded sequences sequences. From top left to bottom right: uniform time warping of length `maximum_warping`, uniform padding of length `maximum_warping`, variable time warping and variable time padding, from 1 to `maximum_warping`. (For uniform padding/warpings, the leaky RNN and gated RNN curves overlap, with loss 0.) Lower is better.

5.3 Experiments

First, we test the theoretical arguments by explicitly introducing random time warpings in some data, and comparing the robustness of gated and ungated architectures.

Next, the chrono LSTM initialization is tested against the standard initialization on a variety of both synthetic and real world problems. It heavily outperforms standard LSTM initialization on all synthetic tasks, and outperforms or competes with it on real world problems.

The synthetic tasks are taken from previous test suites for RNNs, specifically designed to test the efficiency of learning when faced with long term dependencies [Hochreiter and Schmidhuber, 1997, Le et al., 2015, Graves et al., 2014b, Martens and Sutskever, 2011, Arjovsky et al., 2016].

In addition (Appendix 5.4), we test the chrono initialization on next character prediction on the Text8 [Mahoney, 2011] dataset, and on next word prediction on the Penn Treebank dataset [Mikolov et al., 2012]. Single layer LSTMs with various layer sizes are used for all experiments, except for the word level prediction, where we use the best

Unwarped task example:
Input: `All human beings are born free and equal`
Output: `All human beings are born free and equa`
Uniform warping example (warping $\times 4$):
Input: `AAAAlllllll hhhhuuuummmmaaaannnn`
Output: `AAAAlllllll hhhhuuuummmmaaaa`
Variable warping example (random warping $\times 1$–$\times 4$):
Input: `Allllll hhhummmmaannn bbbbeeiiingssss`
Output: `AAAllll huuuummaaan bbeeeingggg`

Figure 5.2: A task involving pure warping.

model from [Zilly et al., 2016], a 10 layer deep recurrent highway network (RHN).

Pure warpings and paddings. To test the theoretical relationship between gating and robustness to time warpings, various recurrent architectures are compared on a task where the only challenge comes from warping.

The unwarped task is simple: remember the previous character of a random sequence of characters. Without time warping or padding, this is an extremely easy task and all recurrent architectures are successful. The only difficulty will come from warping; this way, we explicitly test the robustness of various architectures to time warping and nothing else.

Uniformly time-warped tasks are produced by repeating each character `maximum_warping` times both in the input and output sequence, for some fixed number `maximum_warping`.

Variably time-warped tasks are produced similarly, but each character is repeated a random number of times uniformly drawn between 1 and `maximum_warping`. The same warping is used for the input and output sequence (so that the desired output is indeed a function of the input). This exactly corresponds to transforming input $x(t)$ into $x(c(t))$ with c a random, piecewise affine time warping. Fig. 5.2 gives an illustration.

For each value of `maximum_warping`, the train dataset consists of $50,000$ length-500 randomly warped random sequences, with either uniform or variable time warpings. The alphabet is of size 10 (including a dummy symbol). Contiguous characters are enforced to be different. After warping, each sequence is truncated to length 500. Test datasets of $10,000$ sequences are generated similarily. The criterion to be minimized is the cross entropy in predicting the next character of the output sequence.

Note that each sample in the dataset uses a new random sequence from a fixed alphabet, and (for variable warpings) a new random warping.

A similar, slightly more difficult task uses *padded* sequences instead of warped sequences, obtained by padding each element in the input sequence with a fixed or variable number of 0's (in continuous-time, this amounts to a time warping of a continuous-time input sequence that is nonzero at certain points in time only). Each time the input is nonzero, the network has to output the previous nonzero character seen.

We compare three recurrent architectures: RNNs (Eq. (5.2), a simple, ungated recur-

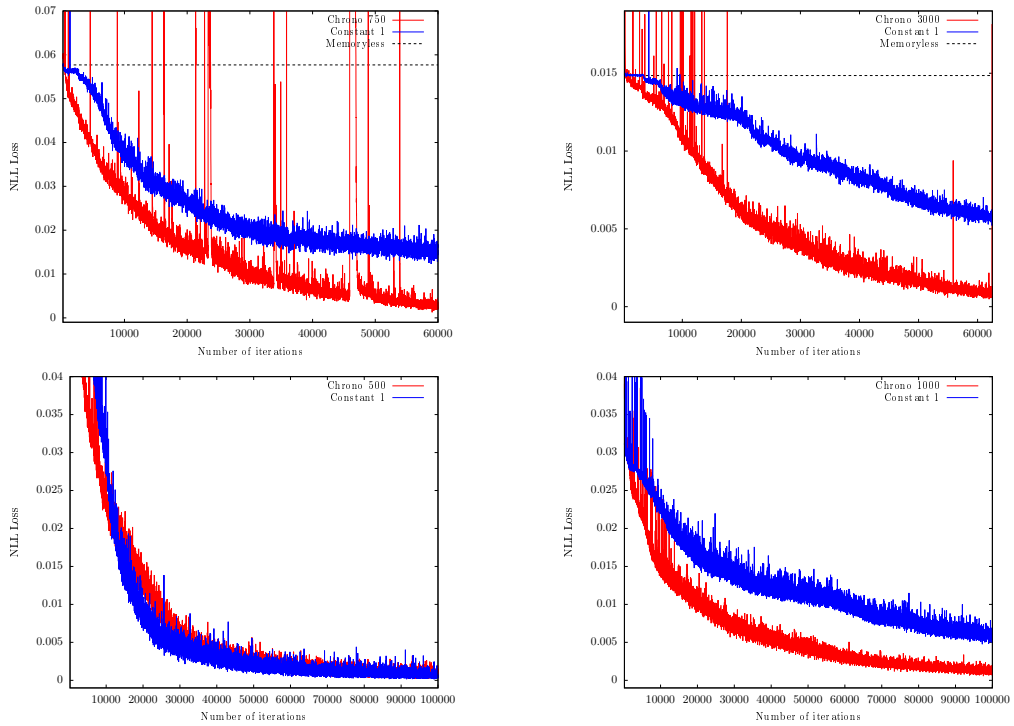


Figure 5.3: Standard initialization (blue) vs. chrono initialization (red) on the copy and variable copy task. From left to right, top to bottom, standard copy $T = 500$ and $T = 2000$, variable copy $T = 500$ and $T = 1000$. Chrono initialization heavily outperforms standard initialization, except for variable length copy with the smaller T where both perform well.

rent network), leaky RNNs (Eq. (5.5), where each unit has a constant learnable "gate" between 0 and 1) and gated RNNs, with one gate per unit, described by (5.10). All networks contain 64 recurrent units.

The point of using gated RNNs (5.10) ("LSTM-lite" with tied input and forget gates), rather than full LSTMs, is to explicitly test the relevance of the arguments in Section 5.1 for time warpings. Indeed these LSTM-lite already exhibit perfect robustness to warpings in these tasks.

RMSprop with an α parameter of 0.9 and a batch size of 32 is used. For faster convergence, learning rates are divided by 2 each time the evaluation loss has not decreased after 100 batches. All architectures are trained for 3 full passes through the dataset, and their evaluation losses are compared. Each setup is run 5 times, and mean, maximum and minimum results among the five trials are reported. Results on the test set are summarized in Fig. 5.1.

Gated architectures significantly outperform RNNs as soon as moderate warping coefficients are involved. As expected from theory, leaky RNNs perfectly solve uniform time warpings, but fail to achieve optimal behavior with variable warpings, to which

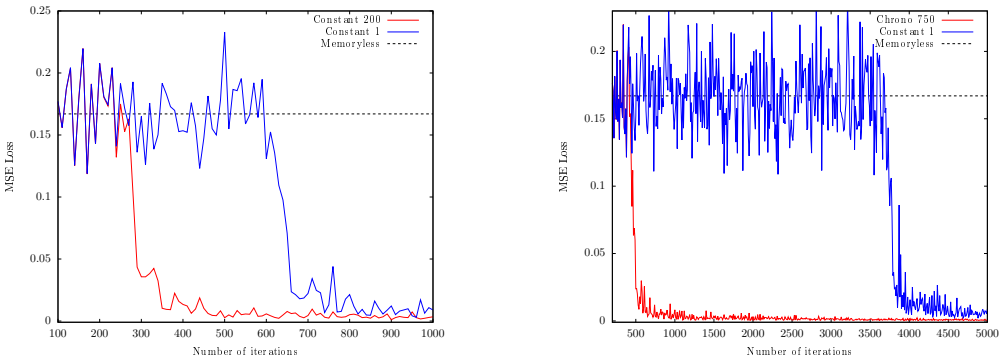


Figure 5.4: Standard initialization (blue) vs. chrono initialization (red) on the adding task. From left to right, $T = 200$, and $T = 750$. Chrono initialization heavily outperforms standard initialization.

they are not invariant. Gated RNNs, which are quasi invariant to general time warpings, achieve perfect performance in both setups for all values of `maximum_warping`.

Synthetic tasks. For synthetic tasks, optimization is performed using RMSprop [Tieleman and Hinton, 2012a] with a learning rate of 10^{-3} and a moving average parameter of 0.9. No gradient clipping is performed; this results in a few short-lived spikes in the plots below, which do not affect final performance.

Copy tasks. The copy task checks whether a model is able to remember information for arbitrarily long durations. We use the setup from [Hochreiter and Schmidhuber, 1997, Arjovsky et al., 2016], which we summarize here. Consider an alphabet of 10 characters. The ninth character is a dummy character and the tenth character is a signal character. For a given T, input sequences consist of $T + 20$ characters. The first 10 characters are drawn uniformly randomly from the first 8 letters of the alphabet. These first characters are followed by $T - 1$ dummy characters, a *signal* character, whose aim is to signal the network that it has to provide its outputs, and the last 10 characters are dummy characters. The target sequence consists of $T + 10$ dummy characters, followed by the first 10 characters of the input. This dataset is thus about remembering an input sequence for exactly T timesteps. We also provide results for the *variable* copy task setup presented in [Henaff et al., 2016], where the number of characters between the end of the sequence to copy and the signal character is drawn at random between 1 and T.

The best that a memoryless model can do on the copy task is to predict at random from among possible characters, yielding a loss of $\frac{10 \log(8)}{T+20}$ [Arjovsky et al., 2016].

On those tasks we use LSTMs with 128 units. For the standard initialization (baseline), the forget gate biases are set to 1. For the new initialization, the forget gate and input gate biases are chosen according to the chrono initialization (5.16), with $T_{\max} = \frac{3T}{2}$ for the copy task, thus a bit larger than input length, and $T_{\max} = T$ for the variable copy task. The results are provided in Figure 5.3.

Importantly, our LSTM baseline (with standard initialization) already performs better than the LSTM baseline of [Arjovsky et al., 2016], which did not outperform random prediction. This is presumably due to slightly larger network size, increased training time, and our using the bias initialization from [Gers and Schmidhuber, 2000].

On the copy task, for all the selected T's, chrono initialization largely outperforms the standard initialization. Notably, it does not plateau at the memoryless optimum. On the variable copy task, chrono initialization is even with standard initialization for $T = 500$, but largely outperforms it for $T = 1000$.

Adding task. The adding task also follows a setup from [Hochreiter and Schmidhuber, 1997, Arjovsky et al., 2016]. Each training example consists of two input sequences of length T. The first one is a sequence of numbers drawn from $\mathcal{U}([0, 1])$, the second is a sequence containing zeros everywhere, except for two locations, one in the first half and another in the second half of the sequence. The target is a single number, which is the sum of the numbers contained in the first sequence at the positions marked in the second sequence.

The best a memoryless model can do on this task is to predict the mean of $2 \times \mathcal{U}([0, 1])$, namely 1 [Arjovsky et al., 2016]. Such a model reaches a mean squared error of 0.167.

LSTMs with 128 hidden units are used. The baseline (standard initialization) initializes the forget biases to 1. The chrono initialization uses $T_{\max} = T$. Results are provided in Figure 5.4. For all T's, chrono initialization significantly speeds up learning. Notably it converges 7 times faster for $T = 750$.

5.4 Additional experiments

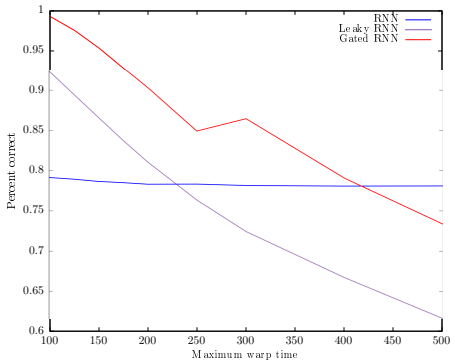


Figure 5.5: Generalization performance of different recurrent architectures on the warping problem. Networks are trained with uniform warps between 1 and 50 and evaluated on uniform warps between 100 and a variable maximum warp.

On the generalization capacity of recurrent architectures. We proceeded to test the generalization properties of RNNs, leaky RNNs and chrono RNNs on the pure

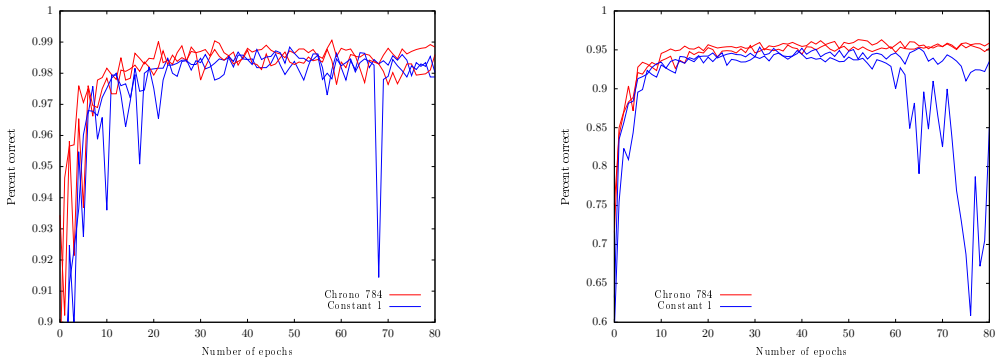


Figure 5.6: Standard initialization (blue) vs. chrono initialization (red) on pixel level classification tasks. From left to right, MNIST and pMNIST.

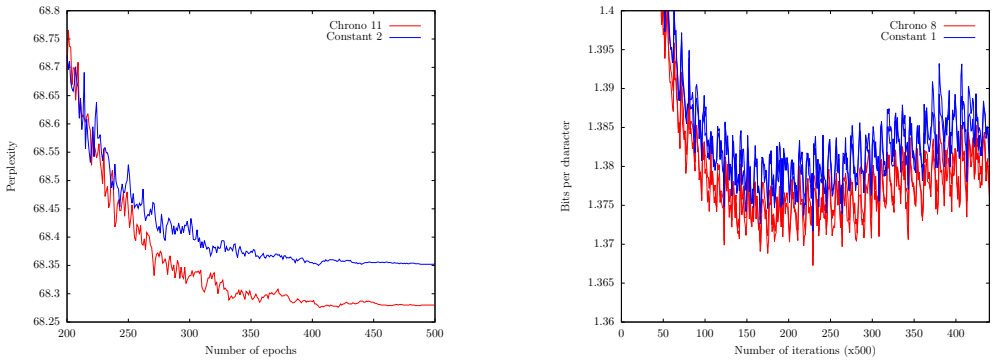


Figure 5.7: Standard initialization (blue) vs. chrono initialization (red) on the word level PTB (left) and on the character level text8 (right) validation sets.

warping experiments presented in Section 5.3. For each of the architectures, a recurrent network with 64 recurrent units is trained for 3 epochs on a variable warping task with warps between 1 and 50. Each network is then tested on warped sequences, with warps between 100 and an increasingly big maximum warping. Results are summarized in Figure 5.5.

All networks display reasonably good, but not perfect, generalization. Even with warps 10 times longer than the training set warps, the networks still have decent accuracy, decreasing from 100% to around 75%.

Interestingly, plain RNNs and gated RNNs display a different pattern: overall, gated RNNs perform better but their generalization performance decreases faster with warps eight to ten times longer than those seen during training, while plain RNN never have perfect accuracy, below 80% even within the training set range, but have a flatter performance when going beyond the training set warp range.

Pixel level classification: MNIST and pMNIST. This task, introduced in [Le et al., 2015], consists in classifying images using a recurrent model. The model is fed pixels one by one, from top to bottom, left to right, and has to output a probability distribution for the class of the object in the image.

We evaluate standard and chrono initialization on two image datasets: MNIST [LeCun et al., 1999] and permuted MNIST, that is, MNIST where all images have undergone the same pixel permutation.

LSTMs with 512 hidden units are used. Once again, standard initialization sets forget biases to 1, and the chrono initialization parameter is set to the length of the input sequences, $T_{\max} = 784$. Results on the validation set are provided in Figure 5.6. On non-permuted MNIST, there is no clear difference, even though the best validation error is obtained with chrono initialization. On permuted MNIST, chrono initialization performs better, with a best validation result of 96.3%, while standard initialization obtains a best validation result of 95.4%.

Next character prediction on text8. Chrono initialization is benchmarked against standard initialization on the character level text8 dataset [Mahoney, 2011]. Text8 is a 100M character formatted text sample from Wikipedia. [Mikolov et al., 2012]'s train-valid-test split is used: the first 90M characters are used as training set, the next 5M as validation set and the last 5M as test set.

The exact same setup as in [Cooijmans et al., 2016b] is used, with the code directly taken from there. Namely: LSTMs with 2000 units, trained with Adam [Kingma and Ba, 2014] with learning rate 10^{-3}, batches of size 128 made of non-overlapping sequences of length 180, and gradient clipping at 1.0. Weights are orthogonally initialized, and recurrent batch normalization [Cooijmans et al., 2016b] is used.

Chrono initialization with $T_{\max} = 8$ is compared to standard $b_f = 1$ initialization. Results are presented in Figure 5.7. On the validation set, chrono initialization uniformly outperforms standard initialization by a small margin. On the test set, the compression rate is 1.37 with chrono initialization, versus 1.38 for standard initialization.[8] This same slight difference is observed on two independent runs.

Our guess is that, on next character prediction, with moderately sized networks, short term dependencies dominate, making the difference between standard and chrono initialization relatively small.

Next word prediction on Penn Treebank. To attest for the resilience of chrono initialization to more complex models than simple LSTMs, we train on word level Penn Treebank [Mikolov et al., 2012] using the best deep RHN network from [Zilly et al., 2016]. All hyperparameters are taken from of [Zilly et al., 2016]. For the chrono bias initialization, a single bias vector b is sampled according to $b \sim \log(\mathcal{U}(1, T_{\max}))$, the carry gate bias vectors of all layers are initialized to $-b$, and the transform gate biases

[8] Both those results are slightly below the 1.36 reported in [Cooijmans et al., 2016b], though we use the same code and same random seed. This might be related to a smaller number of runs, or to a different version of the libraries used.

to b. $T_{\max}$ is chosen to be 11 (because this gives an average bias initialization close to the value 2 from [Zilly et al., 2016]).[9]. Without further hyperparameter search and with a single run, we obtain test results similar to [Zilly et al., 2016], with a test perplexity of 6.54.

Conclusion

The self loop feedback gating mechanism of recurrent networks has been derived from first principles via a postulate of invariance to time warpings. Gated connections appear to regulate the local time constants in recurrent models. With this in mind, the chrono initialization, a principled way of initializing gate biases in LSTMs, has been introduced. Experimentally, chrono initialization is shown to bring notable benefits when facing long term dependencies.

[9]This results in small characteristic times: RHNs stack D update steps in every timestep, where D is the depth of the RHN, so timescales are divided by D.

Chapter 6

Reinforcement learning: framework and overview

Reinforcement Learning (RL) is a natural framework to consider the interaction between an agent and its environment. The agent behavior is motivated using a reward function, that specifies the gains obtained by the agent for performing certain actions in certain subparts of the environment. This interaction, as well as its participants, can take many forms, and span a huge range of applications: for instance a robotic agent, interacting with the physical world, trying to pick up objects, a player, in a two player board game, interacting with both the board and its opponent, and trying to overcome its opponent, or a collaborative agent, trying to understand the needs of its collaborators to achieve a greater common goal are all example of reinforcement learning problems.

The interaction process of the agent with the environment is usually defined through the following cycle, happening at regular time intervals:

1. The agent receives an observation from the environment, which contains information on the current state of the world.
2. The agent processes the received information, and produces an action.
3. The state of the environment is potentially modified after its interaction with the agent.
4. The agent receives a new observation, as well as a reward from the environment.

This interaction cycle is illustrated in Figure 6.1.

As it interacts with the environment, the agent can and will most likely modify the state of the world. The distribution of states that the agent observes is thus dependent on the policy followed by the agent. This is perhaps the most crucial specificity of Reinforcement Learning compared to supervised or unsupervised learning: there is, most often, no controllable shift in the data distribution in supervised and unsupervised settings. The shift of distribution in observations and rewards induced by the agent policy gives rise to challenging problematics, specific to the Reinforcement Learning setup.

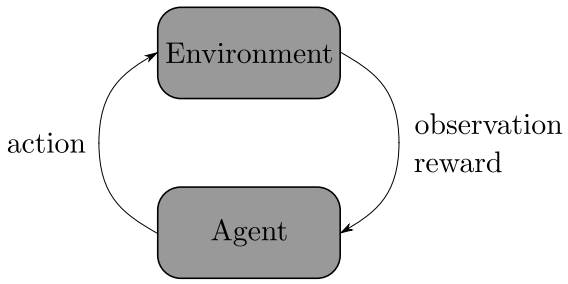


Figure 6.1: Reinforcement Learning interaction cycle.

Among these, the most well-known is the exploration/exploitation tradeoff. To observe that certain sub regions of the state space, or certain sequences of actions, yield high reward, in many cases, an agent needs to spend part of its time exploring new behaviors instead of exploiting known ones.

6.1 Framework

In what follows, we will start by stating and motivating the common *Markov decision process* framework. We then proceed to detail the standard formulation of a Reinforcement Learning problem, and define the relevant quantities to approach it. Finally, we introduce some of the usual methods to tackle reinforcement learning problems. A much more thorough introduction to RL is given in [Sutton and Barto, 1998].

6.1.1 Markov Decision Process

A common assumption in the study and modelisation of physical systems is that there exists a set of variables, as well as a transition dynamic such that if ones knows the state of the system at time t, as well as all the external interactions applied to the system between time t and time $t + \delta t$, one is able to fully describe the state of the system at time $t + \delta t$. For instance, given the angle and angular velocity of a pendulum at a certain time, as well as the torque applied, one can infer the position and velocity of the pendulum at a later time.

Markov Decision Processes (MDP) generalize this intuition to discrete stochastic systems. Formally, a MDP is described as a quadruplet $(\mathcal{S}, \mathcal{A}, T, R)$, where

- $\mathcal{S}$ is the state space of the environment.
- $\mathcal{A}$ is the action space of the agent .
- $\mathcal{T}$ is the transition kernel of the environment. $T(s'|s, a)$ represents the probability for the agent of landing in state s' after executing action a in state s. When the agent interacts with its environment, a Markov property is assumed, the next state only depends on the current state and the performed action, not on any previous action or state.

- $\mathcal{R}$ is the reward function. $R(s, a)$ represents the average reward obtained by the agent when executing action a in state s.

The agent interacts with the environment through a (potentially stochastic) policy π, where $\pi(a|s)$ represents the probability for the agent of executing action a in state s. Such policies only represent a small subset of all possible policies: a policy could, in theory depend on all the history visited states and actions of the agent, and not only on the previous state. However, for the purpose of Reinforcement Learning, one can show that in an MDP, a state dependent stationnary policy is enough to achieve optimal behavior in a sense we will define later. When deterministic policy are involved, the policy is defined as a mapping from state to action space, and the action taken in state s is denoted by $\pi(s)$.

The state in an MDP provides a full description of the world at time t, and an agent doesn't have to remember previous states to act optimally, or make predictions about its future. In general, such a state is not available to the agent. For instance, an agent trying to swing a pendulum up from visual inputs only does not have direct access to the angular velocity, and thus to a full description of the world at each timestep. The agent has to use both the current and the previous observation to access a notion of velocity. In the general case, an agent could require extracting information from the whole history of its past observations and actions to extract a complete representation of the world. In what follows, except for the reproduction work presented in Chapter 7, we always consider that we have access to a fully descriptive state, and do not tackle the complex challenge of building or learning this state. This would typically be the role of a recurrent model of the world.

6.1.2 Return, state value function and goal formulation

The informal goal of a Reinforcement Learning agent is to maximize its reward obtention accross time. Formally, given a trajectory $\tau = (s_t, a_t)_{t\geq 0}$, one can define the average return up to horizon T as

$$R^T(\tau) := \frac{1}{T}\sum_{t=0}^{T} R(s_t, a_t). \tag{6.1}$$

In what follows, the notation $\tau \sim T \times \pi$ denotes trajectories sampled from the interaction of an agent following policy π and the environment. Further define the V-function up to time T as the expected return up to time T, starting from state s

$$V^{\pi,T}(s) := \mathbb{E}_{\tau\sim T\times\pi}\left[R^T(\tau) \mid s_0 = s\right]. \tag{6.2}$$

One can then define, when it exists, the average V-function as

$$V^{\pi}(s) := \lim_{T\to\infty} V^{\pi,T}(s). \tag{6.3}$$

We would then like to find a policy π that has the highest value for all states, i.e. a policy that amasses as much rewards as possible, whatever the state it starts in. Formally, a

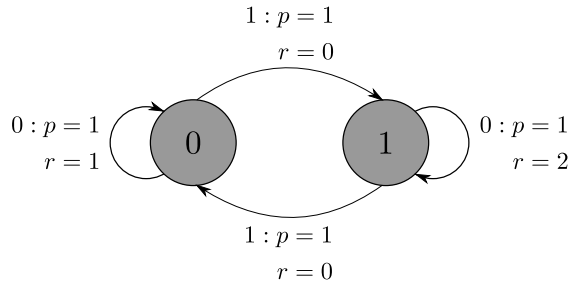


Figure 6.2: γ dependent MDP

policy π dominates a policy π' if for all state s, $V^{\pi}(s) \geq V^{\pi'}(s)$. One can show that there exists a policy π^* that dominates all the others.[1] The goal of reinforcement learning is then to find this optimal policy.

Oftentimes, the average state value function is hard to evaluate, since it weighs all future timesteps equally. To make estimations easier, one can estimate discounted versions of the original problem, which amounts to introducing a γ factor representing the agent's preference for the present. The notion of return for a γ-discounted agent is replaced by

$$R^{\gamma}(\tau) := \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t). \tag{6.4}$$

This quantity is well defined in many setups, and notably for bounded rewards. We can then similarily define a notion of discounted value function and optimal policy for the discounted problem.

It is worth noting that in general, the solution for the discounted problem *is not* the same as the solution for the undiscounted problem. Typically, take the 2 states MDP in Figure 6.2. In state 0, an optimal agent with $\gamma < 1/2$ will choose action 0, while an undiscounted agent chooses action 1. More often than not, our actual goal is to maximize the undiscounted return, and the discounted return only serves as an easier to evaluate proxy.

The average value function quantifies the effect of the starting state on the subsequent return. Similarily, instead of only considering the effect of the initial point, it is sometimes convenient to consider the combined effect of the initial state-action pair. The state-action value function quantifies this dependency, and, in the discounted case, is defined as

$$Q^{\pi}(s, a) = \mathbb{E}_{\tau \sim T \times \pi} \left[R^{\gamma}(\tau) \mid s_0 = s, a_0 = a \right] \tag{6.5}$$

The state and state-action value functions of the optimal policy are denoted by $V^* := V^{\pi^*}$ and $Q^* := Q^{\pi^*}$.

[1] There may be several such policies, but they all share the same value function.

6.1.3 Estimating the V and Q functions

Since reinforcement learning is about maximizing the V^π function w.r.t. the policy π, one is directly interested in computing this quantity. One of the most natural way to perform this estimation is using Monte Carlo samples of the return or discounted return. To be able to perform Monte Carlo estimation of the value, state trajectories must, at one point or another, reach termination. This is possible if one allow terminal states in the MDP, i.e. states from which an agent never escapes, and never gets further rewards, and if all trajectories reach such a state at some point. Under this assumption, Monte Carlo estimates of $V^\pi(s)$ are obtained by sampling interaction trajectories, and averaging all returns starting from state s. When following a given policy, this estimation method is guaranteed to converge, provided the MDP has a finite number of states and each state is visited sufficiently often.

A downside of Monte Carlo estimates is that they require all trajectories to terminate, and don't explicitly make use of the Markov property of the environment. Bootstrap methods overcome both limitations. Bootstrap methods rely on the observation that, under the markov hypothesis, the value functions verify a *Bellman equation*

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(\cdot|s),s'\sim T(\cdot|s,a)}\left[R(s,a) + \gamma V^\pi(s')\right]. \tag{6.6}$$

Using this equation, from an estimate of V^π at time k, V^k, one can obtain a new estimate

$$V^{k+1}(s) = \mathbb{E}_{a\sim\pi(\cdot|s),s'\sim T(\cdot|s,a)}\left[R(s,a) + \gamma V^k(s')\right]. \tag{6.7}$$

Iterating this update can be shown to converge to the true value function. Similar equations can be derived for Q and Q^*, for instance, Q^* verifies the optimal Bellman equation

$$Q^*(s,a) = \mathbb{E}_{s'\sim T(\cdot|s,a)}\left[R(s,a) + \gamma \max_{a'} Q^*(s',a')\right]. \tag{6.8}$$

However, computing these updates requires computing the average on a and s' exactly. This assumes knowledge of $T(\cdot \mid s,a)$, i.e. a perfect knowledge of the dynamic of the world, which is often not available. Sampled trajectory are not sufficient to perform this update. Instead of Eq. (6.7), one can use a Monte Carlo estimate on a single transition. Along with averaging of the old and new estimates, given a transition $s,a \to s'$, this yields

$$V^{k+1}(s) = (1-\alpha_k)V^k(s) + \alpha_k(R(s,a) + \gamma V^k(s')) \tag{6.9}$$

$$= V^k(s) + \alpha_k\Big(R(s,a) + \gamma V^k(s') - V^k(s)\Big) \tag{6.10}$$

where the α_k's are learning rates. Provided that the learning rates are properly scheduled, the sequence of estimates can be shown to converge to V^π.

There are various ways to interpolate between pure bootstrap and Monte Carlo estimates, notably n-step temporal differences and elligibility traces. Both topics are extensively covered in [Sutton and Barto, 1998].

6.2 Reinforcement learning algorithms

Knowing how to estimate value functions, we are now left to find a policy that optimizes the V-function. In this section, we will try to cover various possibilities to find such a policy, namely via policy or value improvement, evolution strategies and policy gradient methods.

6.2.1 Policy improvement Q-learning and SARSA

From estimates of V^π or Q^π, one would want to find a way to obtain a policy π' that is better, in some sense, than π. Intuitively, since V^π and Q^π provide return, and not instantaneous reward information, one would expect that acting greedily w.r.t. Q^π should improve the current policy. The policy improvement theorem confirms this intuition. It states that for two policies π and π', if for all states s, $\mathbb{E}_{a\sim\pi(\cdot|s)}Q^\pi(s,a) \geq V^\pi(s)$, and there exists a state where the inequality is strict, then for all s, $V^{\pi'}(s) \geq V^\pi(s)$ with at least one strict inequality case.

The policy improvement theorem motivates the policy iteration algorithm. Policy iteration picks an initial policy π_0, estimates its Q-function, Q^{π_0}, then chooses a new policy π_1 greedily w.r.t. Q^{π_0}, i.e. such that

$$\mathbb{E}_{a\sim\pi_1(\cdot|s)}\left[Q^{\pi_0}(s,a)\right] = \max_{a'} Q^{\pi_0}(s,a'). \tag{6.11}$$

If π_1 differs from π_0, the algorithm loops using π_1 in place of π_0. Policy iteration can be shown to converge to the optimal policy. A downside of policy iteration is that it requires exact evaluation of the state-action value function at each iteration. Exact evaluation can be relaxed, and simply use multiple steps of estimation of Q^π between each policy improvement. This familly of algorithms are often referred to as value iteration methods. Under mild conditions, this familly of algorithms also converges to the optimal policy.

Q-learning is another way of estimating the optimal policy, which lies at the other extreme of the tradeoff between policy evaluation and policy improvement. Q-learning iterates the optimal Bellman equation to directly approximate the optimal state-action value function. Q-learning updates the Q-function following

$$Q^{k+1}(s,a) = \mathbb{E}_{s'\sim\pi(\cdot|s,a)}\left[R(s,a) + \gamma \max_{a'} Q^k(s,a)\right] \tag{6.12}$$

when the environment is perfectly known. Once again, when iterating on all states with an exact expectation on transitions, Q-learning estimates converge to the optimal Q-function, from which the optimal policy can be easily derived.

There exists variations of the above algorithm that don't require perfect knowledge of the environment, and work using only interaction trajectories, but they typically require handling the exploitation/exploration tradeoff to compute reasonnable values on all state-action pairs, and not only on a sub-area of the state-action space. More precisely, given a trajectory $((s_t, a_t))_{t\geq 0}$, the sampled version of Q-learning yields the

update equation

$$Q^{k+1}(s_k, a_k) = Q^k(s_k, a_k) + \alpha_k \left(R(s_k, a_k) + \gamma \max_{a'} Q^k(s_{k+1}, a') - Q^k(s_k, a_k) \right). \quad (6.13)$$

This update converges to the optimal Q-function provided that learning rates are properly scheduled and the trajectory $((s_t, a_t))_{t \geq 0}$ explores each state-action pair an infinite number of times. A key component of Q-learning is that the policy that it learns, i.e. an argmax policy on the estimate of Q^* *is not necessarily* the policy it uses to generate trajectories. Algorithms that learn a different policy than the one they use to interact with the environment are called *off-policy algorithms*, in contrast to *on-policy algorithms*. Trajectories used to learn need to explore, and there exist several methods to generate exploratory trajectories. A standard and simple exploration method is ε-greedy exploration which selects actions greedily w.r.t. the current estimate of Q^* with probability $1 - \varepsilon$, or uniformly at random with probability ε, where ε is a scalar parameter between 0 and 1. Pseudocode for Q-learning with ε greedy exploration is given in Algorithm 2.

Algorithm 2 Tabular Q-learning with ε-greedy exploration.

Inputs:

Q	table of $\lvert\mathcal{S}\rvert \times \lvert\mathcal{A}\rvert$ values
$\varepsilon \in [0; 1]$	ε-greedy exploration parameter
N	number of iterations
env	environment

Observe s_0 initial state.
$s \leftarrow s_0$
for $t = 0$, N **do**
  Draw e uniformly at random in $[0; 1]$
  if $e \leq \varepsilon$ **then**
    Select a uniformly at random in $\mathcal{A}$
  else
    $a \leftarrow \text{argmax}_{a_0} Q(s, a_0)$
  end if
  Perform a in **env**
  Observe s' and r, next state and reward
  Update Q as
  $Q(s, a) \leftarrow Q(s, a) + \alpha_t \left(r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right)$
  $s \leftarrow s'$
end for

SARSA is an on-policy algorithm that lies very close to Q-learning. The core difference is that while Q-learning picks the value used for bootstrapping using a max, and not the action that was actually performed by the agent, SARSA bootstraps on-policy, with the chosen action. Since it behaves on policy, SARSA needs to incorporate its exploration into its trained policy. It typically uses exploration strategies similar to that

used for Q-learning. Precisely, the update equation and action choices of SARSA with ε-greedy exploration are

$$a_{k+1} = \begin{cases} \text{argmax}_a Q(s_{k+1}, a) \text{ w.p. } 1-\varepsilon \\ a \text{ chosen uniformly at random w.p. } \varepsilon \end{cases} \tag{6.14}$$

$$Q^{k+1}(s_k, a_k) = Q^k(s_k, a_k) + \alpha_k \left(R(s_k, a_k) + \gamma Q^k(s_{k+1}, a_{k+1}) - Q^k(s_k, a_k) \right). \tag{6.15}$$

Variations on Q-learning and SARSA have achieved great successes throughout the development of reinforcement learning. TD-Gammon [Tesauro, 1995] used a variation of SARSA with elligibility traces to compete with top level backgammon players. Deep Q-learning [Mnih et al., 2015] adapted Q-learning to deep networks function approximations to achieve human level performance on many Atari games, from raw visual inputs. Current state of the art approaches on the Atari benchmark, such as [Hessel et al., 2017], are Q-learning variations.

6.2.2 Cost minimization methods

Instead of trying to find the general optimal policy, one may want to select the best policy in a familly of parameterized policies. Formally, let $(\pi_\theta)_{\theta \in \mathbb{R}^p}$ be such a familly, where θ is the policies parameter. Policies are often compared by defining the following cost function

$$J(\theta) = -\mathbb{E}_{s \sim \rho_0} \left[V^{\pi_\theta}(s) \right] \tag{6.16}$$

where ρ_0 is the initial state distribution. A policy π_{θ_1} is considered better than a policy π_{θ_2} if it bears a smaller cost, i.e. $J(\theta_1) < J(\theta_2)$. We then seek to find a θ that minimizes J. Note that minimizing J does not necessarily provide the best policy in the sense given in Section 6.1.2. More precisely, one can find θ_1 and θ_2 such that $J(\theta_1) < J(\theta_2)$, but there exists s such that $V^{\pi_{\theta_1}}(s) < V^{\pi_{\theta_2}}(s)$. Typically, if ρ_0 is reduced to a dirac on a single point s_0, and $\gamma < 1$, a policy is deemed better than another if and only if its value at s_0 is greater. With such a ρ_0, the value at other points is not taken into account when comparing policies.

There are several methods to minimize J. In the next subsections, we will cover evolution strategies and policy gradient methods.

Evolution strategies

Evolution strategies (ES) are very simple, yet sometimes surprisingly successful ways of optimizing the policy of an agent. The principle behind evolution strategies is to let a whole population of agents, with different parameters, interact with the environment and to iteratively modify the population such that at each iteration, members of the population are better suited to the problem, i.e. have a lower overall cost. As an example, pseudocode for the version of ES presented in [Salimans et al., 2017] is given in Algorithm 3. The intuitive idea is to apply several small random modifications to the initial parameter, look at how these modifications affect the return of the agent, typically

Algorithm 3 Simple evolution strategy for reinforcement learning.

Inputs:

$\theta_0 \in \mathbb{R}^p$	initial parameter
env	environment
$\sigma > 0$	noise standard deviation
N	number of iterations
P	number of rollouts
N_{pop}	population size
α	learning rate

$\theta \leftarrow \theta_0$
for $n = 0; n < N; n++$ **do**
  Initialize **returns** and **advs** tables of N_{pop} reals.
  Initialize **noises**, table of N_{pop} p-dimensional noise vectors.
  for $k = 0; k < N_{\text{pop}}; k++$ **do**
    $\theta' \leftarrow \theta + \sigma\varepsilon$ with $\varepsilon \sim \mathcal{N}(0, I_p)$
    noises$[k] \leftarrow \varepsilon$
    Obtain return R as the average from P rollouts with policy $\pi_{\theta'}$
    returns$[k] \leftarrow R$
  end for
  $\mu_R \leftarrow \text{mean}(\textbf{returns})$
  $\sigma_R \leftarrow \text{std}(\textbf{returns})$
  for $k = 0; k < N_{\text{pop}}; k++$ **do**
    advs$[k] \leftarrow \frac{\textbf{returns}[k] - \mu_R}{\sigma_R}$
  end for
  $\theta \leftarrow \theta + \frac{\alpha}{\sigma N_{\text{pop}}} \sum_{k=0}^{N_{\text{pop}}} \textbf{noises}[k]\textbf{advs}[k]$
end for

by estimating J through rollouts (which requires the environment to be episodic), and push the parameters in the direction of the samples that were most successful.

More complicated evolution strategies have been used, notably *Covariance Matrix Adatpation Evolution Strategy* (CMAES [Hansen and Auger, 2014]), e.g. in [Ha and Schmidhuber, 2018]. In [Ha and Schmidhuber, 2018], CMAES, along with a recurrent model of the world, achieves state of the arts results on a toy driving environment. A reproduction of this work is presented in Chapter 7.

Policy gradient methods

Instead of relying on zero-th order optimization, policy gradient methods try to directly estimate the gradient of J from samples. Policy gradient methods rely on the policy gradient theorem, which gives the following equality for the gradient of J

$$\partial_\theta J(\theta) = \mathbb{E}_{s \sim \rho^{\gamma, \pi_\theta}, a \sim \pi_\theta(\cdot|s)} \left[\partial_\theta \log \pi_\theta(a \mid s) Q^{\pi_\theta}(s, a) \right] \tag{6.17}$$

where ρ^{γ,π_θ} is the discounted (unnormalized) state distribution

$$\rho^{\gamma,\pi_\theta}(s) = \sum_{s_0} \sum_{t=1}^{+\infty} \gamma^{t-1} \rho_0(s_0) p^{\pi_\theta}(s_0 \to s, t) \tag{6.18}$$

where $p^{\pi_\theta}(s_0 \to s, t)$ is the probability of going from s_0 to s in t steps. The proof can be found in [Sutton and Barto, 1998]. This estimate of the gradient is samplable, if one has access to Q^{π_θ}, simply by sampling trajectories from the initial state distribution, and selecting states in the trajectories with a preference for the present. If one only has access to an estimate of Q^{π_θ}, the gradient estimate can still be used, but is likely to be biased.

One may notice that, for any function of states b, and for any s,

$$\mathbb{E}_{a\sim\pi_\theta(\cdot|s)} \left[\partial_\theta \log \pi_\theta(a \mid s) b(s)\right] = 0. \tag{6.19}$$

Consequently,

$$\partial_\theta J(\theta) = \mathbb{E}_{s\sim\rho^{\gamma,\pi_\theta}, a\sim\pi_\theta(\cdot|s)} \left[\partial_\theta \log \pi_\theta(a \mid s) \left(Q^{\pi_\theta}(s,a) - b(s)\right)\right]. \tag{6.20}$$

Picking an appropriate b can be crucial to reduce the variance of the Monte Carlo estimates of $\partial_\theta J(\theta)$. A common choice is to take $b(s) = V^{\pi_\theta}(s)$, which yields,

$$\partial_\theta J(\theta) = \mathbb{E}_{s\sim\rho^{\gamma,\pi_\theta}, a\sim\pi_\theta(\cdot|s)} \left[\partial_\theta \log \pi_\theta(a \mid s) \left(A^{\pi_\theta}(s,a)\right)\right]. \tag{6.21}$$

where $A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$ is the *advantage function*, which measures the relative improvment of taking a specific action a in state s compared to the average value of the policy.

The policy gradient theorem, and the different gradient estimates that it provides, depending both on the method used to evaluate the Q-function and on the baseline used, have given birth to many successful algorithms. *Advantage Actor Critic* (A2C), and its asynchronous counterpart *Asynchronous Advantage Actor Critic* (A3C) [Mnih et al., 2016] result from direct applications of the policy gradient theorem, and display competitive results compared to DQN [Mnih et al., 2015]. *Deep Deterministic Policy Gradient* (DDPG [Lillicrap et al., 2015]) is a policy gradient method, derived from a deterministic continuous action version of the policy gradient theorem, widely used in continuous control RL problems. Lastly, recent state of the art methods, such as *Proximal Policy Optimization* (PPO [Schulman et al., 2017]), or *IMPALA* [Espeholt et al., 2018], are modifications of policy gradient methods that allow for moderate off-policyness.

6.3 Contributions

In Chapter 7, we provide a reimplementation of [Ha and Schmidhuber, 2018], a model based reinforcement learning method using features from a model of the world to improve the result of a CMAES based evolution strategy, as presented in Section 6.2.2. In Chapter 8, we detail why Q-learning, as introduced in Section 6.2.1, is not robust to time discretization, and provide a time discretization robust variant.

Chapter 7

Reproducing Recurrent World Models Facilitate Policy Evolution

Foreword:

The article *Recurrent World Models Facilitate Policy Evolution* [Ha and Schmidhuber, 2018] provides a simple yet efficient learning procedure for a reinforcement learning agent to benefit from a model of the world. In the context of this thesis, which aims at providing better tools to build intelligent agents able to understand and interact with the world, reproducing and testing the robustness of this work came as a natural goal. We showed, along with Léonard Blier, who provided an equal contribution to the code, and Diviyan Kalainathan, who setup the proper software environment to replicate the experiments, that the paper was replicable, and provided additional results regarding the usefulness of the model. The code of the replication is available at `https://github.com/ctallec/world-models`.

Recently, Deep Reinforcement Learning (DRL) has achieved impressive results in a variety of domains, such as video game playing [Mnih et al., 2015] zero-sum games [Silver et al., 2017], and continuous control [Lillicrap et al., 2015]. Still, DRL approaches are often brittle, sensitive to small changes in hyperparameters, implementation details and minor environment perturbations. Besides, training performance can widely vary from run to run. Those factors often make reproduction of experimental results challenging [Henderson et al., 2017, Zhang et al., 2018].

In addition to its sensitivity, DRL is also known to be sample inefficient, in the sense that it requires huge amounts of environment interactions to obtain good results, even for simple tasks. Model-based reinforcement learning has gained interest to improve sample efficiency. With an accurate and computationally cheap model of the world, the burden of collecting new samples could be considerably alleviated, since the model could, in principle, generate huge amounts of reliable samples, and be used for planning without interacting with the environment. Besides, features provided by a predictive model of the world could constitue relevant inputs to a controller, and ease the optimization process.

[Ha and Schmidhuber, 2018] provided a simple, yet successful model-based reinforcement learning approach. It revolves around a three part model, comprised of:

1. A Variational Auto-Encoder [Kingma and Welling, 2013], a generative model, which learns both an encoder and a decoder. The encoder's task is to compress the input images into a compact latent representation. The decoder's task is to recover the original image from the latent representation.

2. A Mixture-Density Recurrent Network [Graves, 2013], trained to predict the latent encoding of the next frame given past latent encodings and actions. The mixture-density network outputs a Gaussian mixture for predicting the distribution density of the next latent variable.

3. A simple linear Controller. It takes as inputs both the latent encoding of the current frame and the hidden state of the MDN-RNN given past latents and actions and outputs an action. It is trained to maximize the cumulated reward using the Covariance-Matrix Adaptation Evolution-Strategy [Hansen and Auger, 2014], a generic gradient-free black box optimization algorithm.

On a given environment, the model is trained sequentially as follows:

1. Sample randomly generated rollouts from a well suited *random policy*.

2. Train the VAE on images drawn from the rollouts.

3. Train the MDN-RNN on the rollouts encoded using the encoder of the VAE. To reduce computational load, we trained the MDN-RNN on fixed size subsequences of the rollouts.

4. Train the controller while interacting with the environment using CMA-ES. At each time step, the controller takes as input both the encoded current frame and the

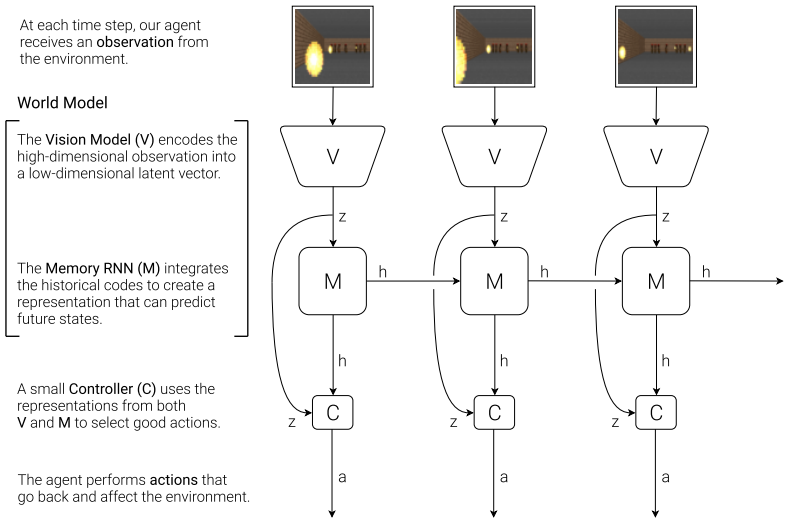


Figure 7.1: The three parts of the architecture (from the original paper)

recurrent state of the MDN-RNN, which contains information about all previous frames and actions.

Alternatively, if the MDN-RNN is good enough at modelling the environment, the controller can be trained directly on simulated rollouts in the dreamt environment.

7.1 Methods

7.1.1 Reproducibility of the original results

We reproduced the authors results on the CarRacing environment [Brockman et al., 2016]. The goal of the CarRacing environment is to drive a car along a track as fast as possible. The agent receives top down images of the car as input and outputs three real values, corresponding to a steering coeficient, a breaking coeficient and an acceleration. Rewards are given each time the agent traverses a checkpoint on the track. We only used the paper description, and took the same hyperparameters as the original paper, unless stated otherwise below, or originally unspecified. We did not use any original sources, and we did not contact the authors. The exact training procedure is detailled below.

7.1.2 Data generation

The original paper started by generating rollouts using a random policy interacting with the environment. The policy was not specified. In our experiments we tested two types of policies. The first policy generates independant standard normal actions at each step. The second policy generates actions according to a discretized brownian motion, with a discretization parameter of $\frac{1}{50}$. This means that each component of the action (the

action is three dimensional) is generated as

$$a_{t+1}^k = a_t^k + \frac{1}{\sqrt{50}}\varepsilon_t^k$$

where the ε's are i.i.d. standard normal random variables. The data samples generated by the first policy lack diversity, and the car only moves in a very restricted area of the track. The samples generated by the second policy, on the other are much more diverse, and were consequently used for the next stages of training. The original paper generated 10,000 rollouts. For our replication, we generated 1,000 rollouts, to reduce the computational load.

7.1.3 Variational auto-encoder (VAE) training

The VAE is trained following the training procedure of the original paper, on the rollouts previously generated. The model is the same as the one detailled in the paper. The first 600 rollouts are used as a training set, and the next 400 as a validation set. The VAE is optimized using Adam [Kingma and Ba, 2014] with default hyperparameters, learning rate 10^{-3}, $\beta_1 = 0.9$, $\beta_2 = 0.999$. Since the dataset is quite large, validation is perfomed each time a fifth of the dataset has been processed. Each step of validation is performed on 200 samples of the validation dataset. The learning rate is halved each time the validation performance has not improved for 5 consecutive evaluations. Training is stopped when the validation performance has not improved for 30 consecutive evaluations. Training was performed on a single Quadro GP100 GPU, with 16Go of RAM.

7.1.4 Mixture Density Recurrent Neural Network (MDN-RNN) training

Similarily the MDN-RNN is trained following the original paper procedure. The model is the same as the one detailled in the paper. The training splits are the same as for the VAE, and evaluations are also performed using the same schedule. The network is trained and validated on subrollouts of length 32. RMSprop [Tieleman and Hinton, 2012b] is used to optimize the MDN-RNN, with a learning rate 10^{-3}, and $\alpha = 0.9$. The learning rate schedule and early stopping policy of the VAE training are used without any modification. Training was performed on a single Quadro GP100 GPU, with 16Go of RAM.

7.1.5 Controller training with CMAES

Finally, the controller is trained using the Covariance Matrix Adaptation Evolution Strategy [Hansen and Auger, 2014], using the python library pycma [Hansen et al., 2019]. As in the original paper, the controller is a linear network that takes as inputs the concatenation of the output of the VAE and the hidden layer of the MDN-RNN. For a set of parameters of the controller, the loss function is obtained by averaging the

Table 7.1: Results from the original paper.

Method	Average score
Full World Models [Ha and Schmidhuber, 2018]	906 ± 21
without MDN-RNN [Ha and Schmidhuber, 2018]	632 ± 251

returns from 16 rollouts of length at most 1000. The population size used for CMAES is 64, and an initial standard deviation of 0.1 is used. Rollouts are executed in parallel, on 64 threads, with models sharing 8 V100 GPUs.

7.2 Results

7.2.1 Reproducibility

On the CarRacing-v0 environment, results were reproducible with relative ease. The model achieved good results on the first try, relatively to the usual reproducibility standards of deep reinforcement learning algorithms [Henderson et al., 2017, Zhang et al., 2018]. Our own implementation reached a best score of 895, below the 906 reported in the paper, but much better than the second best benchmark reported which is around 780. Results are further detailled in the tables and Figure 7.2. Figure 7.2 displays the learning curves of CMAES in the three considered setups. Solid lines represent the mean performance and standard deviation of the population, while dashed line represent the maximal performance. At test time, we select the best performing element of the CMAES population.

7.2.2 Additional experiments

We wanted to test the impact of the MDN-RNN on the results. Indeed, we observed during training that the model was rapidly learning the easy part of the dynamic, but mostly failed to account for long term effects and multimodality.

In the original paper, the authors performed an ablation study, and compared their results with a model without the MDN-RNN. They obtained the following scores:

Still, we wanted to investigate this question even more. We also trained the controller, but with an untrained MDN-RNN instead of the trained one (we kept it at its random initialization values). Similarily, we tried training a controller with both an untrained MDN-RNN and an untrained VAE. Surprisingly, controllers trained with untrained recurrent models achieve results close to the values of the original controller, while controllers with both untrained VAE and MDN-RNN achieves very low performance.

It seems that the training of the MDN-RNN does not significantly improve the performance. Our interpretation of this phenomenon is that even if the recurrent model is not able to predict the next state of the environment, its recurrent state still contains

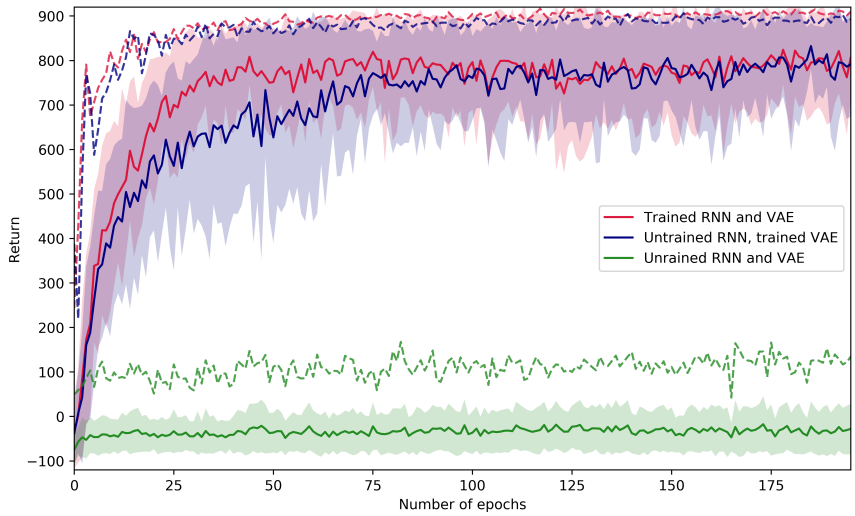


Figure 7.2: Learning curves of CMAES. This qualitatively replicates Figure 4 left from [Ha and Schmidhuber, 2018]. The number of generations is lower here, due to computational limitations.

Table 7.2: Our reproduction results.

Method	Average score
With a trained MDN-RNN	895 ± 79
With an untrained MDN-RNN	866 ± 69
With untrained MDN-RNN and VAE	131 ± 66

some crucial information on the environment dynamic. Without a recurrent model, first-order information such as the velocity of the car is absent from individual frames, and consequently from latent codes. Therefore, strategies learnt without the MDN-RNN cannot use such information. Even a random MDN-RNN still holds some useful temporal information, which is enough to learn a good strategy on this problem.

7.3 Conclusion

We reproduced the paper [Ha and Schmidhuber, 2018] on the CarRacing environment, and made some additional experiments. Overall, our conclusions are twofold:

- The results were easy to reproduce. It probably means that the method on this problem does not only achieve high perforance but is also very stable. This is an important remark for a deep reinforcement learning method.
- On the CarRacing-v0 environment, it seems that the recurrent network only serves as a recurrent reservoir, enabling access to crucial higher order information, such as velocity or acceleration. This observation needs some perspective, it comes with several interrogations and remarks:
  - [Ha and Schmidhuber, 2018] reports good results when training in the simulated environment on the VizDoom task. Without a trained recurrent forward model, we cannot expect to obtain such performance.
  - On CarRacing-v0, the untrained MDN-RNN already obtains near optimal results. Is the task sufficiently easy to alleviate the need for a good recurrent forward model?
  - Learning a good model of a high dimensional environment is hard. It is notably difficult to obtain coherent multi modal behaviors on long time ranges (i.e. predicting two futures, one where the next turn is a right turn, the other where it is a left turn). Visually, despite the latent gaussian mixture model, our model doesn't seem to overcome this difficulty. Is proper handling of multi modal behaviors key to leveraging the usefulness of a model of the world?

Chapter 8

Making Deep Q-learning Approaches Robust to Time Discretization

Foreword:

For an agent to be able to interact with the world, without the risk of catastrophically failing when encountering specific situations, we would like reinforcement learning algorithms to be applicable on a wide range of problems. A direction to provide algorithms as general as possible is to check that they are resilient to changes in the specification of environments that do not change the inherent difficulty of the environment but can affect algorithm performance. An example of such modification is the addition of no-ops actions, i.e. actions that have no effect on the environment. Some algorithms may succeed on environments without added no-ops, but catastrophically fail when provided with too many no-ops. Similarily, for environments that are discretizations of an inherent continuous-time environment, changes of time discretization do not intrinsically change the environment, and should therefore only mildly affect our learning algorithms. This line of thinking is the main motivation behind the work presented in this chapter, *Making Deep Q-learning Approaches Robust to Time Discretization* [Tallec et al., 2019], which is a collaboration with Léonard Blier, and was originally presented at ICML 2019.

Despite remarkable successes, *Deep Reinforcement Learning* (DRL) is not robust to hyperparameterization, implementation details, or small environment changes [Henderson et al., 2017, Zhang et al., 2018]. Overcoming such sensitivity is key to making DRL applicable to real world problems. In this paper, we identify sensitivity to *time discretization* in near continuous-time environments as a critical factor; this covers, e.g., changing the number of frames per second, or the action frequency of the controller. Empirically, we find that Q-learning-based approaches such as *Deep Q-learning* [Mnih et al., 2015] and *Deep Deterministic Policy Gradient* [Lillicrap et al., 2015] collapse with small time steps. Formally, we prove that Q-learning does not exist in continuous time. We detail a principled way to build an off-policy RL algorithm that yields similar performances over a wide range of time discretizations, and confirm this robustness empirically.

8.1 Introduction

In recent years, *Deep Reinforcement Learning* (DRL) approaches have provided impressive results in a variety of domains, achieving superhuman performance with no expert knowledge in perfect information zero-sum games [Silver et al., 2017], reaching top player level in video games [OpenAI, 2018b, Mnih et al., 2015]), or learning dexterous manipulation from scratch without demonstrations [OpenAI, 2018a]. In spite of those successes, DRL approaches are sensitive to a number of factors, including hyperparameterization, implementation details or small changes in the environment parameters [Henderson et al., 2017, Zhang et al., 2018]. This sensitivity, along with sample inefficiency, largely prevents DRL from being applied in real world settings. Notably, high sensitivity to environment parameters prevents transfer from imperfect simulators to real world scenarios.

In this paper we focus on the sensitivity to time discretization of DRL approaches, such as what happens when an agent receives 50 observations and is expected to take 50 actions per second instead of 10. In principle, decreasing time discretization, or equivalently shortening reaction time, should only improve agent performance. Robustness to time discretization is especially relevant in *near-continuous* environments, which includes most continuous control environments, robotics, and many video games.

Standard approaches based on estimation of state-action value functions, such as *Deep Q-learning* (DQN, [Mnih et al., 2015]) and *Deep deterministic policy gradient* (DDPG, [Lillicrap et al., 2015]) are not at all robust to changes in time discretization. This is shown experimentally in Sec. 8.5. Intuitively, as the discretization timestep decreases, the effect of individual actions on the total return decreases too: $Q^*(s, a)$ is the value of playing action a then playing optimally, and if a is only maintained for a very short time its advantage over other actions will be accordingly small. (This occurs even with a suitably adjusted decay factor γ.) If the discretization timestep becomes infinitesimal, the effect of every individual action vanishes: there is no continuous-time Q-function (Thm. 2), hence the poor performance of Q-learning with small time steps. These statements can be fully formalized in the framework of continuous-time reinforcement learning (Sec. 8.3) [Doya, 2000, Baird, 1994].

We focus on continuous time because this leads to a clear theoretical framework,

but our observations make sense in any setting in which the value results from taking a large number of small individual actions. Our results suggest standard Q-learning will fail in such settings without a delicate balance of hyperparameter scalings and reparameterizations.

We are looking for an algorithm that would be as invariant as possible to changing the discretization timestep. Such an algorithm should remain viable, in the sense that it should still learn, when this timestep is small, and in particular admit a continuous-time limit when the discretization timestep goes to 0. This leads to precise design choices in terms of agent architecture, exploration policy, and learning rates scalings. The resulting algorithm is shown to provide better invariance to time discretization than vanilla DQN or DDPG, on many environments (Sec. 8.5). On a new environment, as soon as the order of magnitude of the time discretization is known, our analysis readily provides relevant scalings for a number of hyperparameters.

Our contribution is threefold:

- Building on [Baird, 1994], we formally show that the Q-function collapses to the V-function in near-continuous time, and thus that standard Q-learning is ill-behaved in this setting.
- Our analysis of properties in the continuous-time limit leads to a robust off-policy algorithm. In particular, it provides insights on architecture design, and constrains exploration schemes and learning rates scalings.
- We empirically show that standard Q-learning methods are not robust to changes in time discretization, exhibiting degraded performance, while our algorithm demonstrates substantial robustness.

To the best of our knowledge, Thms 6 and 2 were known to [Doya, 2000, Baird, 1994], but not formally proven, while Thms 3, 4 and 5 are novel. The theoretical results presented in the main text are formally proven in the Appendix.

8.2 Related Work

Our approach builds on [Baird, 1994], who identified the collapse of Q-learning for small time steps and, as a solution, suggested the Advantage Updating algorithm, with proper scalings for the V and advantage parts depending on timescale δt; testing was only done on a quadratic-linear problem.

We expand on [Baird, 1994] in several directions. First, we modify the algorithm by using a different normalization step for A, which forgoes the need to learn the normalization itself, thanks to the parameterization (8.27). Second, we test Advantage Updating for the first time on a variety on RL environments using deep networks, establishing Deep Advantage Updating as a viable algorithm in this setting. Third, we provide formal proofs in a general setting for the collapse of Q-learning when the timescale δt tends to 0, and for the non-collapse of Advantage Updating with the proper scalings. Fourth,

we also discuss how to obtain δt-invariant exploration. Fifth, we provide stringent experimental tests of the actual robustness to changing δt.

Our study focuses on off-policy algorithms. Some on-policy algorithms, such as A3C [Mnih et al., 2016], PPO [Schulman et al., 2017] or TRPO [Schulman et al., 2015] may be time discretization invariant with specific setups. This is out of the scope of our work and would require a separate study.

[Wang et al., 2015] also use a parameterization separating the value and advantage components of the Q-function. But contrary to [Baird, 1994]'s Advantage Updating, learning is still done in a standard way on the Q-function obtained from adding these two components. Thus this approach reparameterizes Q but does not change scalings and does not result in an invariant algorithm for small δt.

The problem studied here is a continuity effect quite distinct from multiscale RL approaches: indeed the issue arises even if there is only one timescale in the environment. Arguably, a small δt can be seen as a mismatch between the algorithm's timescale and the physical system's timescale, but the collapse of the Q function to the V function is an intrinsic mathematical phenomenon arising from time continuity.

Reinforcement learning has been studied from a mathematical perspective when time and space are both continuous, in connection with optimal control and the Hamilton–Jacobi–Bellman (HJB) equation (a PDE which characterizes the value function for continuous space-time). Explicit algorithms for continuous space-time can be found in [Doya, 2000, Munos and Bourgine, 1998] (see also the references therein). [Munos and Bourgine, 1998] use a grid approach to provably solve the HJB equation when discretization tends to 0 (assuming every state in the grid is visited a large number of times). However, the resulting algorithms are impractical [Doya, 2000] for larger-dimensional problems. [Doya, 2000] focusses on algorithms specific to the continuous space-time case, including advantage updating and modelling the time derivative of the environment.

Here on the other hand we focus on generic deep RL algorithms that can handle both discrete and continuous time and space, without collapsing in continuous time, thus being robust to arbitrary timesteps.

8.3 Near Continuous-Time Reinforcement Learning

Many reinforcement learning environments are not intrinsically time-discrete, but discretizations of an underlying continuous-time environment. For instance, many simulated control environments, such as the Mujoco environments [Lillicrap et al., 2015] or OpenAI Gym classic control environments [Brockman et al., 2016], are discretizations of continuous-time control problems. In simulated environments, the time discretization is fixed by the simulator, and is often used to approximate an underlying differential equation. In this case, the timestep may correspond to the number of frames generated by second. In real world environments, sensors and actuators have a fixed time precision: cameras can only capture a fixed amount of frames per second, and physical limitations prevent actuators from responding instantaneously. The quality of these

components thus imposes a lower bound on the discretization timestep. As the timestep δt is largely a constraint imposed by computational ressources, we would expect that decreasing δt would only improve the performance of RL agents (though it might make optimization harder). RL algorithms should, at least, be resilient to a change of δt, and should remain viable when $\delta t \to 0$. Besides, designing a time discretization invariant algorithm could alleviate tedious hyperparameterization by providing better defaults for time-horizon-related parameters.

We are thus interested in the behavior of RL algorithms in discretized environments, when the discretization timestep is small. We will refer to such environments as *near-continuous environments*. A formalized view of near-continuous environments is given below, along with δt-dependent definitions of return, discount factor and value functions, that converge to well defined continous-time limits. The state-action value function is shown to collapse to the value function as δt goes to 0. Consequently there is no Q-learning in continuous time, foreshadowing problematic behavior of Q-learning with small timesteps.

8.3.1 Framework

Let $\mathcal{S} = \mathbb{R}^d$ be a set of states, and $\mathcal{A}$ be a set of actions. Consider the continuous-time *Markov Decision Process* (MDP) defined by the differential equation

$$ds_t/dt = F(s_t, a_t) \tag{8.1}$$

where $F \colon \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ describes the dynamics of the environment. The agent interacts with the environment through a deterministic policy function $\pi \colon \mathcal{S} \to \mathcal{A}$, so that $a_t = \pi(s_t)$. Actions can be discrete or continuous. For simplicity we assume here that both the dynamics and exploitation policy are deterministic; [1] the exploration policy will be random, but care must be taken to define proper random policies in continuous time, especially with discrete actions (Sec. 8.4.2).

For any timestep $\delta t > 0$, we can define an MDP $\mathcal{M}_{\delta t} = \langle \mathcal{S}, \mathcal{A}, T_{\delta t}, r_{\delta t}, \gamma_{\delta t} \rangle$ as a discretization of the continuous-time MDP with *time discretization* δt. The transition function of a state s is the state obtained when starting at $s_0 = s$ and maintaining $a_t = a$ constant for a time δt. This corresponds to an agent evolving in the continuous environment (8.1), but only making observations and choosing actions every δt. The rewards and decay factor are specified below. We call such an MDP $\mathcal{M}_{\delta t}$ *near-continuous*.

A necessary condition for robustness of an algorithm for near-continuous time MDPs is to remain viable when $\delta t \to 0$. Thus we will try to make sure the various quantities involved converge to meaningful limits when $\delta t \to 0$.

[1] We believe the results presented here hold more generally, assuming states follow a *stochastic* differential equation

$$ds = F(s, a)dt + \Sigma(s, a)dB_t \tag{8.2}$$

with B_t a multidimensional Brownian motion and Σ a covariance matrix. A formal treatment of SDEs is beyond the scope of this paper.

We give semi-formal statements below; the full statements, proofs, and technical assumptions (typically, differentiability assumptions) can be found in the supplementary material.

Return and discount factor. Suitable δt-dependent scalings of the discount factor $\gamma_{\delta t}$ and reward $r_{\delta t}$ are as follows. These definitions fit the discrete case when $\delta t = 1$, and provide well-defined, non-trivial returns and value functions when δt goes to 0.

For a continuous MDP and a continuous trajectory $\tau = (s_t, a_t)_t$, the return is defined as [Doya, 2000]

$$R(\tau) := \int_{t=0}^{\infty} \gamma^t \, r(s_t, a_t) \, dt. \tag{8.3}$$

A natural time discretization is obtained by defining the discretized return $R_{\delta t}$ of the MDP $\mathcal{M}_{\delta t}$ as

$$R_{\delta t}(\tau) := \sum_{k=0}^{\infty} \gamma^{k\delta t} \, r(s_{k\delta t}, a_{k\delta t}) \, \delta t \tag{8.4}$$

and the discretized return will correspond to the continuous-time return if we set the decay factor $\gamma_{\delta t}$ and rewards $r_{\delta t}$ of the discretized MDP $\mathcal{M}_{\delta t}$ to

$$\gamma_{\delta t} := \gamma^{\delta t}, \qquad r_{\delta t} := \delta t . r. \tag{8.5}$$

Physical time vs algorithmic time, time horizon. In near-continuous environments, there are two notions of time: the algorithmic time k (number of steps or actions taken), and the physical time t (time spent in the underlying continuous time environment), related via $t = k.\delta t$.

The time horizon is, informally, the time range over which the agent optimizes its return. As a rule of thumb, the time horizon of an agent with discount factor γ is of order $\frac{1}{1-\gamma}$ steps; beyond that, the decay factor kicks in and the influence of further rewards becomes small.

The definition (8.5) of the decay factor $\gamma_{\delta t}$ in near-continuous environments keeps the time horizon constant in *physical* time, by making $\gamma_{\delta t}$ close to 1 in algorithmic time. Indeed, physical time horizon is δt times the algorithmic time horizon, namely

$$\frac{\delta t}{1 - \gamma^{\delta t}} = -\frac{1}{\log \gamma} + O(\delta t) \approx \frac{1}{1-\gamma}, \tag{8.6}$$

which is thus stable when $\delta t \to 0$. On the other hand, if $\gamma_{\delta t}$ was left constant as δt goes to 0, the corresponding time horizon in physical time would be $\approx \frac{\delta t}{1-\gamma^{\delta t}}$ which goes to 0 when δt goes to 0: such an agent would be increasingly short-sighted as $\delta t \to 0$.

In the following, we use the suitably-scaled decay factor (8.5) both for Deep Advantage Updating and for the classical deep Q-learning baselines.

Value function. The return (8.3) leads to the following continuous-time value function

$$V^\pi(s) = \mathbb{E}_{\tau\sim\pi}\left[R(\tau) \mid s_0 = s\right] \tag{8.7}$$

$$= \mathbb{E}_{\tau\sim\pi}\left[\int_0^\infty \gamma^t\, r(s_t, a_t)\, dt \mid s_0 = s\right]. \tag{8.8}$$

Meanwhile, the value function (in the ordinary sense) of the discrete MDP $\mathcal{M}_{\delta t}$ is

$$V^\pi_{\delta t}(s) = \mathbb{E}_{\tau\sim\pi}\left[R_{\delta t}(\tau) \mid s_0 = s\right] \tag{8.9}$$

$$= \mathbb{E}_{\tau\sim\pi}\left[\sum_{k=0}^\infty \gamma^{k\delta t}\, r(s_{k\delta t}, a_{k\delta t})\, \delta t \mid s_0 = s\right] \tag{8.10}$$

which obeys the Bellman equation [2]

$$V^\pi_{\delta t}(s) = r(s, \pi(s))\, \delta t + \gamma^{\delta t}\, \mathbb{E}_{s_{(k+1)\delta t}|s_{k\delta t}=s} V^\pi_{\delta t}(s_{(k+1)\delta t}) \tag{8.11}$$

When the timestep tends to 0, one converges to the other.

Theorem 1. *Under suitable smoothness assumptions, $V^\pi_{\delta t}(s)$ converges to $V^\pi(s)$ when $\delta t \to 0$.*

8.3.2 There is No Q-Function in Continuous Time

Contrary to the value function, the action-value function Q is ill-defined for continuous-time MDPs. More precisely, the Q-function collapses to the V-function when $\delta t \to 0$. In near continuous time, the effect of individual actions on the Q-function is of order $O(\delta t)$. This will make ranking of actions difficult, especially with an approximate Q-function. This argument appears informally in [Baird, 1994]. Formally:

Theorem 2. *Under suitable smoothness assumptions, The action-value function of a near-continuous MDP is related to its value function via*

$$Q^\pi_{\delta t}(s, a) = V^\pi_{\delta t}(s) + O\left(\delta t\right) \tag{8.12}$$

when $\delta t \to 0$, for every (s, a).

As a consequence, in exact continuous time, Q^π is equal to V^π: it does not bear *any* information on the ranking of actions, and thus cannot be used to select actions with higher returns. There is no continuous-time Q-learning.

[2] If the continuous MDP follows the dynamics (8.1), the limit of the Bellman equation (8.11) for $V^\pi_{\delta t}$ when $\delta t \to 0$ is the *Hamilton–Jacobi–Bellman* equation on V^π [Doya, 2000], namely, $r + \nabla_s V^\pi \cdot F = -\log(\gamma) V^\pi$.

Proof. The discrete-time Q-function of the MDP $\mathcal{M}_{\delta t}$ satisfies the Bellman equation

$$Q^\pi_{\delta t}(s,a) = r(s,a)\,\delta t + \gamma^{\delta t}\mathbb{E}_{s'|s,a}\left[V^\pi_{\delta t}(s')\right]. \tag{8.13}$$

The dynamics (8.1) of the environment yields

$$s' = s + F(s,a)\,\delta t + o(\delta t). \tag{8.14}$$

Assuming that $V^\pi_{\delta t}$ is continuously differentiable with respect to the state, and that its derivatives are uniformly bounded, we obtain,

$$V^\pi_{\delta t}(s') = V^\pi_{\delta t}(s) + \nabla_s V^\pi_{\delta t}(s)\cdot F(s,a)\,\delta t + o(\delta t) \tag{8.15}$$
$$= V^\pi_{\delta t}(s) + O(\delta t) \tag{8.16}$$

Expanding $V^\pi_{\delta t}(s')$ into $Q^\pi_{\delta t}$ yields

$$Q^\pi_{\delta t}(s,a) = r(s,a)\,\delta t + \gamma^{\delta t}(V^\pi_{\delta t}(s) + O(\delta t)) \tag{8.17}$$
$$= O(\delta t) + (1 + O(\delta t))(V^\pi_{\delta t}(s) + O(\delta t)) \tag{8.18}$$
$$= V^\pi_{\delta t}(s) + O(\delta t). \tag{8.19}$$

which ends the proof. □

8.4 Reinforcement Learning with a Continuous-Time Limit

We now define a discrete algorithm with a well-defined continuous-time limit. It relies on three elements: defining and learning a quantity that still contains information on action rankings in the limit, using exploration methods with a meaningful limit, and scaling learning rates to induce well-behaved parameter trajectories when δt goes to 0.

8.4.1 Advantage Updating

As seen above, there is no continuous time limit to Q-learning, because Q^π becomes independent of actions and thus cannot be used to select actions. With small but nonzero δt, $Q^\pi_{\delta t}$ still depends on actions, and could still be used to choose actions. However, when approximating $Q^\pi_{\delta t}$, if the approximation error is much larger than $O(\delta t)$, this error dominates, the ranking of actions given by the approximated $Q^\pi_{\delta t}$ is likely to be erroneous.

To define an object which contains the same information on actions as $Q^\pi_{\delta t}$, but admits a learnable action-dependent limit, it is natural to define [Baird, 1994]

$$A^\pi_{\delta t}(s,a) := \frac{Q^\pi_{\delta t}(s,a) - V^\pi_{\delta t}(s)}{\delta t}, \tag{8.20}$$

a rescaled version of the advantage function, as the difference between between $Q^\pi_{\delta t}(s,a)$ and $V^\pi_{\delta t}(s)$ is of order $O(\delta t)$. This amounts to splitting Q into value and advantage, and observing that these scale very differently when $\delta t \to 0$.

Contrary to the Q-function, this rescaled advantage function converges when $\delta t \to 0$ to a non-degenerate action-dependent quantity.

Theorem 3. *Under suitable smoothness assumptions, $A^\pi_{\delta t}(s,a)$ has a limit $A^\pi(s,a)$ when $\delta t \to 0$. The limit keeps information about actions: namely, if a policy π' strictly dominates π, then $A^\pi(s,\pi'(s)) > 0$ for some state s.*

Learning A^π. The discretized Q-function rewrites as

$$Q^\pi_{\delta t}(s,a) = V^\pi_{\delta t}(s) + \delta t A^\pi_{\delta t}(s,a). \tag{8.21}$$

A natural way to approximate $V^\pi_{\delta t}$ and $A^\pi_{\delta t}$ is to apply *Sarsa* or Q-learning to a reparameterized Q-function approximator

$$Q_\Theta(s,a) := V_\theta(s) + \delta t A_\psi(s,a). \tag{8.22}$$

with $\Theta := (\theta, \psi)$. At initialization, if both V_θ and A_ψ are initialized independently of δt, this parameterization provides reasonable scaling of the contribution of actions versus states in Q. Our goal is for V_θ to approximate $V^\pi_{\delta t}$ and for A_ψ to approximate $A^\pi_{\delta t}$.

Still, this reparameterization does not, on its own, guarantee that A correctly approximates $A^\pi_{\delta t}$ if Q_Θ approximates $Q^\pi_{\delta t}$. Indeed, for any given pair (V_θ, A_ψ), the pair ($V_\theta(s) - f(s)$, $A_\psi(s,a) + f(s)/\delta t$) (for an arbitrary f) yields the exact same function Q_Θ. This new A_ψ still defines the same ranking of actions, yet this phenomenon might cause numerical problems or instability of A_ψ when $\delta t \to 0$, and prevents direct interpretation of the learned A_ψ. To enforce identifiability of A_ψ, one must enforce the consistency equation

$$V^\pi_{\delta t}(s) = Q^\pi_{\delta t}(s, \pi(s)) \tag{8.23}$$

on the approximate A_ψ and V_θ. This translates to

$$A_\psi(s, \pi(s)) = 0. \tag{8.24}$$

With this additional constraint, if $Q_\Theta = Q^\pi_{\delta t}$, then $A_\psi = A^\pi_{\delta t}$ and $V_\theta = V^\pi_{\delta t}$: indeed

$$A^\pi_{\delta t}(s,a) = \frac{Q^\pi_{\delta t}(s,a) - V^\pi_{\delta t}(s)}{\delta t} \tag{8.25}$$

$$= \frac{Q_\Theta(s,a) - Q_\Theta(s,\pi(s))}{\delta t} = A_\psi(s,a). \tag{8.26}$$

In the spirit of [Wang et al., 2015], instead of directly parameterizing A_ψ, we define a parametric function $\bar{A}_\psi$ (typically a neural network), and use $\bar{A}_\psi$ to define A_ψ as

$$A_\psi(s,a) := \bar{A}_\psi(s,a) - \bar{A}_\psi(s,\pi(s)) \tag{8.27}$$

so that A_ψ directly verifies the consistency condition.

This approach will lead to δt-robust algorithms for approximating $A^\pi_{\delta t}$, from which a ranking of actions can be derived.

8.4.2 Timestep-Invariant Exploration

To obtain a timestep-invariant RL algorithm, a timestep-invariant exploration scheme is required. For *continuous* actions, [Lillicrap et al., 2015] already introduced such a scheme, by adding an *Ornstein–Uhlenbeck* [Uhlenbeck and Ornstein, 1930] (OU) random process to the actions. Formally, this is defined as

$$\pi^{\text{explore}}(s_{k\delta t}, z_{k\delta t}) := \pi(s_{k\delta t}) + z_{k\delta t} \tag{8.28}$$

with $z_{k\delta t}$ the discretization of a continuous-time OU process,

$$dz_t = -z_t\,\kappa\,dt + \sigma\,dB_t. \tag{8.29}$$

where B_t is a brownian motion, κ a stiffness parameter and σ a noise scaling parameter. The discretized trajectories of z converge to nontrivial continuous-time trajectories, exhibiting Brownian behavior with a recall force towards 0.

This exploration can be extended to schemes of the form

$$a_{k\delta t} = \pi^{\text{explore}}_{\delta t}(s_{k\delta t}, z_{k\delta t}) \tag{8.30}$$

with $(z_{k\delta t})_{k\geq 0}$ a sequence of random variables independent from the a's and s's. A sufficient condition for this policy to admit a continuous-time limit is for the sequence $z_{k\delta t}$ to converge in law to a well-defined continuous stochastic process z_t as δt goes to 0.

Thus, for *discrete* actions we can obtain a consistent exploration scheme by taking $z_{\delta t}$ to be a discretization of an $(\#\mathcal{A})$-dimensional continuous OU process, and setting

$$\pi^{\text{explore}}(s_{k\delta t}, z_{k\delta t}) := \operatorname*{argmax}_a \left(A_\psi(s_{k\delta t}, a) + z_{k\delta t}[a]\right)$$

where $z_{k\delta t}[a]$ denotes the a-th component of $z_{k\delta t}$. Namely, we perturb the *advantage values* by a random process before selecting an action. The resulting scheme converges in continuous time to a nontrivial exploration scheme.

On the other hand, ε-greedy exploration is likely *not* to explore, i.e., to collapse to a deterministic policy, when δt goes to 0. Intuitively, with very small δt, changing the action at random every δt time step just averages out the randomness due to the law of large numbers. More precisely:

Theorem 4. *Consider a near-continuous MDP in which an agent selects an action according to an ε-greedy policy that mixes a deterministic exploitation policy π with an action taken from a noise policy $\pi^{noise}(a|s)$ with probability ε at each step. Then the agent's trajectories converge when $\delta t \to 0$ to the solutions of the* deterministic *equation*

$$ds_t/dt = (1-\varepsilon)F(s_t, \pi(s_t)) + \varepsilon \mathbb{E}_{a\sim\pi^{noise}(a|s)} F(s_t, a)$$

8.4.3 Algorithms for Deep Advantage Updating

We learn V_θ and A_ψ via suitable variants of Q-learning for continuous and discrete action spaces. Namely, the true $A^\pi_{\delta t}$ and $V^\pi_{\delta t}$ of a near-continuous MDP with greedy exploitation

Algorithm 4 Deep Advantage Updating (Continuous actions)

Input

θ, ψ and ϕ, parameters of V_θ, $\bar{A}_\psi$ and π_ϕ.
π^{explore} and $\nu_{\delta t}$ defining an exploration policy.
$\mathbf{opt}_V$, $\mathbf{opt}_A$, $\mathbf{opt}_\pi$, $\alpha^V \delta t$, $\alpha^A \delta t$ and $\alpha^\pi \delta t$, optimizers and learning rates.
$\mathcal{D}$, buffer of transitions (s, a, r, d, s'), with d the episode termination signal.
δt and γ, time discretization and discount factor.
nb_epochs number of epochs.
nb_steps, number of steps per epoch.
nb_learn, number of learning step per epoch
N, training batch size

Observe initial state s^0
$t \leftarrow 0$
for $e = 0$, **nb_epochs do**
  for $j = 1$, **nb_steps do**
    $a^t \leftarrow \pi^{\text{explore}}(s^t, \nu^t_{\delta t})$.
    Perform a^t and observe $(r^{t+1}, d^{t+1}, s^{t+1})$.
    Store $(s^t, a^t, r^{t+1}, d^{t+1}, s^{t+1})$ in $\mathcal{D}$.
    $t \leftarrow t + 1$
  end for
  for $k = 0$, **nb_learn do**
    Sample a batch of N random transitions from $\mathcal{D}$
    for i=0, N **do**

$$\delta Q^i \leftarrow \delta t\big(\bar{A}_\psi(s^i, a^i) - \bar{A}_\psi(s^i, \pi_\phi(s^i))\big) - \Big(r^i \delta t + (1 - d^i)\gamma^{\delta t} V_\theta(s'^i) - V_\theta(s^i)\Big)$$

    end for

$$\Delta\theta \leftarrow \frac{1}{N} \sum_{i=1}^{N} \frac{\delta Q^i \partial_\theta V_\theta(s^i)}{\delta t}$$

$$\Delta\psi \leftarrow \frac{1}{N} \sum_{i=1}^{N} \frac{\delta Q^i \partial_\psi \big(\bar{A}_\psi(s^i,a^i) - \bar{A}_\psi(s^i,\pi_\phi(s^i))\big)}{\delta t}$$

$$\Delta\phi \leftarrow \frac{1}{N} \sum_{i=1}^{N} \partial_a \bar{A}_\psi(s^i, \pi_\phi(s^i)) \partial_\phi \pi_\phi(s^i)$$

    Update θ with $\mathbf{opt}_V$, $\Delta\theta$ and learning rate $\alpha^V \delta t$.
    Update ψ with $\mathbf{opt}_A$, $\Delta\psi$ and learning rate $\alpha^A \delta t$.
    Update ϕ with $\mathbf{opt}_\pi$, $\Delta\phi$ and learning rate $\alpha^\pi \delta t$.
  end for
end for

policy $\pi(s) := \text{argmax}_{a'} A^\pi_{\delta t}(s, a')$ are the unique solution to the Bellman and consistency

equations

$$V^{\pi}_{\delta t}(s) + \delta t\, A^{\pi}_{\delta t}(s,a) = r\,\delta t + \gamma^{\delta t}\,\mathbb{E}_{s'} V^{\pi}_{\delta t}(s') \tag{8.31}$$

$$A^{\pi}_{\delta t}(s,\pi(s)) = 0. \tag{8.32}$$

as seen in 8.4.1. Thus V_θ and A_ψ are trained to approximately solve these equations.

Maximization over actions for π is implemented exactly for discrete actions, and, for continuous actions, approximated by a policy neural network $\pi_\phi(s)$ trained to maximize $A_\psi(s,\pi_\phi(s))$, similarly to [Lillicrap et al., 2015].

Eq. (8.32) is directly verified by A_ψ, owing to the reparametrization $A_\psi(s,a) = \bar{A}_\psi(s,a) - \bar{A}_\psi(s,\pi(s))$, described in 8.4.1. To approximately verify (8.31), the corresponding squared Bellman residual is minimized by an approximate gradient descent. The update equations when learning from a transition (s,a,r,s'), either from an exploratory trajectory or from a replay buffer [Mnih et al., 2015], are

$$\delta Q_{\delta t} \leftarrow A_\psi(s,a)\,\delta t - \Big(r\,\delta t + \gamma^{\delta t} V_\theta(s') - V_\theta(s)\Big) \tag{8.33}$$

$$\theta_{\delta t} \leftarrow \theta_{\delta t} - \eta^{V}_{\delta t}\,\partial_\theta V_\theta(s)\,\frac{\delta Q_{\delta t}}{\delta t} \tag{8.34}$$

$$\psi_{\delta t} \leftarrow \psi_{\delta t} - \eta^{A}_{\delta t}\,\partial_\psi A_\psi(s,a)\,\frac{\delta Q_{\delta t}}{\delta t}. \tag{8.35}$$

where the η's are learning rates. Appropriate scalings for the learning rates $\eta^V_{\delta t}$ and $\eta^A_{\delta t}$ in terms of δt to obtain a well defined continuous limit are derived next.

8.4.4 Scaling the Learning Rates

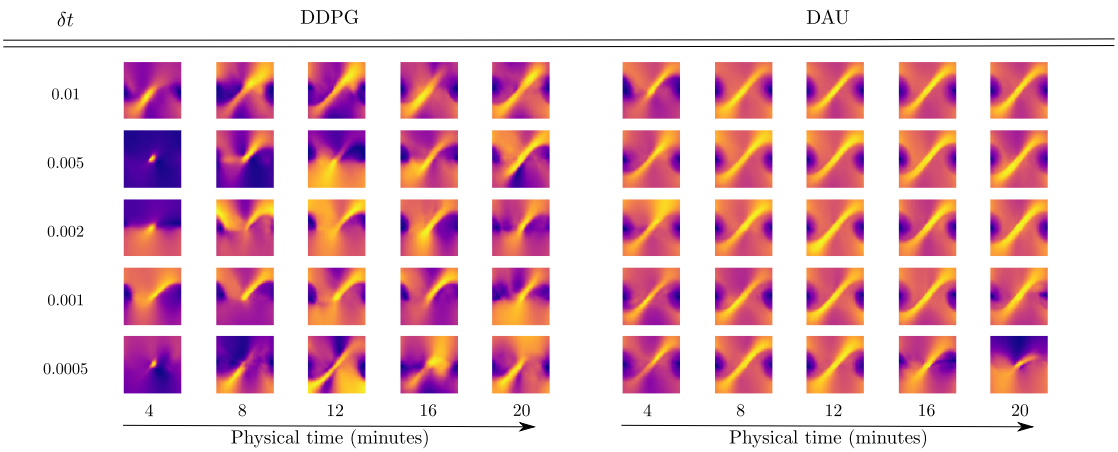


Figure 8.1: Value functions obtained by DDPG (unscaled version) and DAU at different instants in physical time of training on the pendulum swing-up environment. Each image represents the learnt value function (the x-axis is the angle, and the y-axis the angular velocity). The lighter the pixel, the higher the value.

For the algorithm to admit a continuous-time limit, the discrete-time trajectories of parameters must converge to well-defined trajectories as δt goes to 0. This in turn imposes precise conditions on the scalings of the learning rates.

Informally, in the parameter updates (8.33)–(8.35), the quantity $\delta Q_{\delta t}$ is of order $O(\delta t)$, because $s' = s + O(\delta t)$ in a near-continuous system. Therefore, $\delta Q_{\delta t}/\delta t$ is $O(1)$, so that the gradients used to update θ and ψ are $O(1)$. Therefore, if the learning rates are of order δt, one would expect the parameters θ and ψ to change by $O(\delta t)$ in each time interval δt, thus hopefully converging to smooth continuous-time trajectories. The next theorem formally confirms that learning rates of order δt are the only possibility.

Theorem 5. *Let (s_t, a_t) be some exploration trajectory in a near-continuous MDP. Set the learning rates to $\eta^V_{\delta t} = \alpha^V \delta t^\beta$ and $\eta^A = \alpha^A \delta t^\beta$ for some $\beta \geq 0$, and learn the parameters θ and ψ by iterating (8.33)–(8.35) along the trajectory (s_t, a_t). Then, when $\delta t \to 0$:*

- *If $\beta = 1$ the discrete parameter trajectories converge to continuous parameter trajectories.*
- *If $\beta > 1$ the parameters stay at their initial values.*
- *If $\beta < 1$, the parameters can reach infinity in arbitrarily small physical time.*

The resulting algorithm with suitable scalings, *Deep Advantage Updating* (DAU), is specified in Alg. 4 for continuous actions (in the Supplementary for discrete ones). With $\delta t = 1$, DAU is similar to dueling architectures [Wang et al., 2015], but weighs contributions differently for $\delta t \neq 1$.

8.5 Experiments

The experiments provided here are specifically aimed at showing that the proposed method, DAU, works efficiently over a wide range of time discretizations, without specific tuning, while standard deep Q-learning approaches do not. DAU is compared to DDPG for continuous actions and to DQN for discrete actions. As mentionned earlier, we do not study the time discretization invariance of on-policy methods (A3C, PPO, TRPO...).

In all setups, we use the algorithms described in Alg. 4 and Supplementary Alg. 1. The variants of DDPG and DQN used are described in the Supplementary, as well as all hyperparameters. We tested two different setups for DDPG and DQN. In one, we scaled the discount factor (to avoid shortsightedness with small δt), but left all other hyperparameters constant across time discretizations. In the other, we used the properly rescaled discount factor and reward from Eq. (8.5), as well as $O(\delta t)$ learning rates for RMSProp. The first variant empirically yields slightly better results, and is presented here, with the second variant in the Supplementary. For all setups, quantitative results are averaged over five runs.

Let us stress that the quantities plotted are rescaled to make comparison possible across different timesteps. For example, returns are given in terms of the discretized

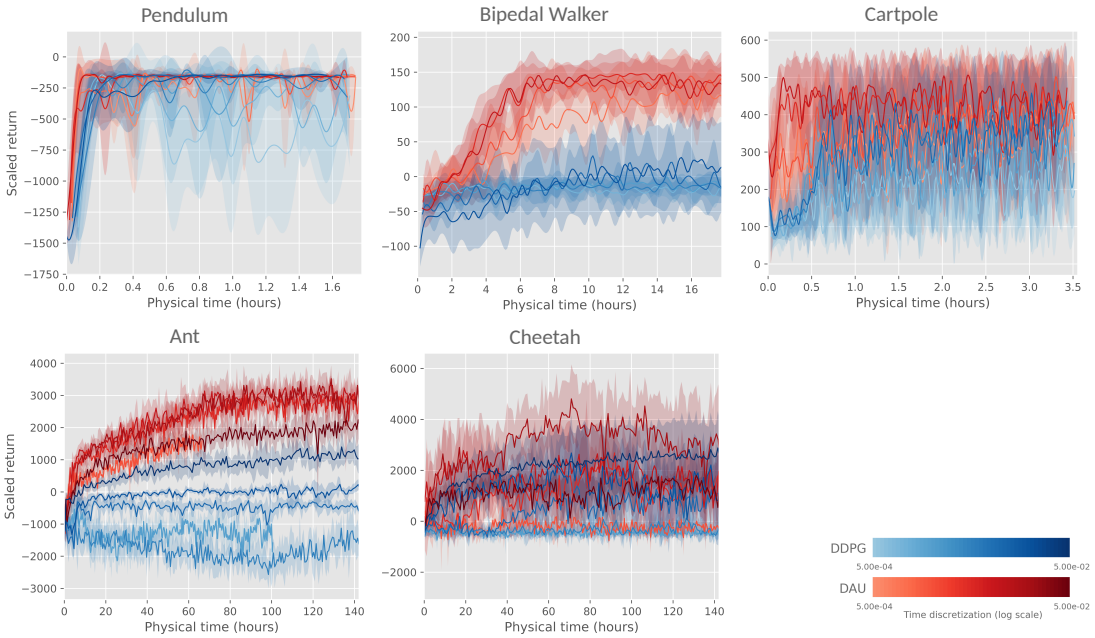


Figure 8.2: Learning curves for DAU and DDPG on classic control benchmarks for various time discretization δt: Scaled return as a function of the physical time spent in the environment.

return $R_{\delta t}$ as defined in (8.4),[3] and, most notably, time elapsed is always given in *physical* time, i.e., the amount of time that the agent spent interacting with the environment (this is not the number of steps).

Qualitative experiments: Visualizing policies and values. To provide qualitative results, and check robustness to time discretization both in terms of returns and in terms of convergence of the approximate value function and policies, we first provide results on the simple pendulum environment from the OpenAI Gym classic control suite. The state space is of dimension 2. We visualize both the learnt value and policy functions by plotting, for each point of the phase diagram $(\theta, \dot{\theta})$, its value and policy. The results are presented in Fig. 8.1 (value function) and Figs. 1, 2, 3 in Supplementary.

We plot the learnt policy at several instants in physical time during training, for various time discretizations δt, for both DAU and DDPG. With DAU, the agent's policy and value function quickly converge for every time discretization without specific tuning. On the contrary, with DDPG, learning of both value function and policy vary greatly from one discretization to another.

[3]This mostly amounts to scaling rewards by a factor δt when this scaling is not naturally done in the environment. Environment-specific details are given in the Supplementary.

Quantitative experiments. We benchmark DAU against DDPG on classic control benchmarks[4]: Pendulum, Cartpole, BipedalWalker, Ant, and Half-Cheetah environments from OpenAI Gym (Fig. 8.2). On Pendulum, Bipedal Walker and Ant, DAU is quite robust to variations of δt and displays reasonable performance in all cases. On the other hand, DDPG's performance varies with δt; performance either degrades as δt decreases (Ant, Cheetah), or the standard deviation across runs increases as learning progresses (Pendulum) for small δt. On Cartpole, noise dominates, making interpretation difficult. On Half-Cheetah, DAU is not clearly invariant to time discretization. This could be explained by the multiple suboptimal regimes that coexist in the Half-Cheetah environment (walking on the head, walking on the back), which create discontinuities in the value function (see Discussion).

8.6 Discussion

The method derived in this work is theoretically invariant to time discretization, and indeed seems to yield improved timestep robustness on various environments, e.g., simple locomotion tasks. However, on some environments there is still room for improvment. We discuss some of the issues that could explain this theoretical/practical discrepancy.

Note that Alg. 4 requires knowledge of the timestep δt. In most environments, this is readily available, or even directly set by the practitioner: depending on the environment it is given by the frame rate, the maximum frequency of actuators or observation acquisition, the timestep of a physics simulator, etc.

Smoothness of the value function. In our proofs, V^π is assumed to be continuously differentiable. This hypothesis is not always satisfied in practice. For instance, in the pendulum swing-up environment, depending on initial position and momentum, the optimal policy may need to oscillate before reaching the target state. The optimal value function is discontinuous at the boundary between states where oscillations are needed and those where they are not. This results in non-infinitesimal advantages for actions on the boundary. In such environments where a given policy has different regimes depending on the initial state, the continuous-time analysis only holds almost-everywhere.

Memory buffer size. Thm. 5 is stated for transitions sampled sequentially from a fixed trajectory. In practice, transitions are sampled from a memory replay buffer, to prevent excessive correlations. We used a fixed-size circular buffer, filled with samples from a single growing exploratory trajectory. In our experiments, the same buffer size was used for all time discretizations. Thus the physical-time freshness of samples in the buffer varies with the time discretization, and in the strictest sense using a fixed-size buffer breaks timestep invariance. A memory-intensive option would be to use a buffer of size $\frac{1}{\delta t}$ (fixed memory in physical time).

[4] We also experimented with a continuous action variant of dueling architectures as a baseline, but found that the results were not substantially different than that of vanilla DDPG.

Near-continuous reinforcement learning and RMSProp. RMSProp [Tieleman and Hinton, 2012b] divides gradient steps by the square root of a moving average estimate of the second moment of gradients. This may interact with the learning rate scaling discussed above. In deterministic environments, gradients typically scale as $O(1)$ in terms of δt, as seen in (8.35). In that case, RMSProp preconditioning has no effect on the suitable order of magnitude for learning rates. However, in near continuous *stochastic* environments (Eq. 8.2), variance of $\delta Q_{\delta t}/\delta t$ and of the gradients typically scales as $O\left(1/\delta t\right)$. With a fixed batch size, RMSProp will multiply gradients by a factor $O(\sqrt{\delta t})$. In that case, learning rates need only be scaled as $\sqrt{\delta t}$ instead of δt.

More generally, the continuous-time analysis should in principle be repeated for every component of a system. For instance, if a recurrent model is used to handle state memory or partial observability, care should be taken that the model is able to maintain memory for a non-infinitesimal physical time when $\delta t \to 0$ (see e.g. [Tallec and Ollivier, 2018]).

8.7 Conclusion

Q-learning methods have been found to fail to learn with small time steps, both theoretically and empirically. A theoretical analysis help in building a practical off-policy deep RL algorithm with better robustness to time discretization. This robustness is confirmed empirically.

Chapter 9

Conclusion

The aim of this thesis was to provide theoretical solutions and insights to existing problems in recurrent networks training and reinforcement learning. Among the core contributions introduced are the design of scalable, online, and theoretically grounded learning algorithms for Recurrent Networks. These algorithms can theoretically capture arbitrarily long term dependencencies, while still processing data in an online fashion. We provide an interpretation on the capacity of recurrent networks to learn long term dependencies by examining invariance properties. This analysis additionnally gives better initialization methods for gated networks.

In Reinforcement Learning, our core contribution is to highlight, both in theory and practice, the lack of robustness of Q-learning based approaches to changes of time discretization. We introduce corrections to standard methods to provide such robustness. Additionnally, we reimplemented [Ha and Schmidhuber, 2018], solidifying the idea that model based approaches combined with simple evolutionary methods can provide good results on challenging environments.

Among our different contributions, providing unbiased loss gradient estimates for recurrent neural networks has probably been the most influencial. The idea of approximating the unscalable derivative of the internal state of the network w.r.t. its parameters by a much smaller, scalably computable object has given rise to a serie of followups, which either try to reduce the overall variance of the approximation by providing different normalization coeficients [Cooijmans and Martens, 2019], or provide different tradeoffs between variance and computational requirements [Mujika et al., 2018]. The main limitation of UORO is still the variance of its gradient estimate, and more precisely the dependency of the variance on the number of recurrent units, which makes learning prohibitively slow for very large networks. Finding ways to limit the increase in variance for large networks while maintaining a scalable computational complexity will be key in making UORO-like approaches fully competitive with Backpropagation Through Time.

Chrono initialization, which was introduced in [Tallec and Ollivier, 2018] has become a standard benchmark initialization when working on long term dependencies, as examplified in [Thornton et al., 2019] or [Chandar et al., 2019], and allowed increased performance on other architectures than the one described in the paper, e.g. [Takamura

and Yamane, 2019]. Furthermore, the invariance analysis used in [Tallec and Ollivier, 2018] inspired part of the analysis of [Ganea et al., 2018]. In substance, both [Tallec and Ollivier, 2018] and [Tallec et al., 2019] strive towards the similar objective of providing robustness to changes in the representation of time. The success of this methodology in two substancially different fields seems to indicate that analysing invariance to time discretization of algorithm dealing with sequential data is a fruitful research direction, that can provide both theoretical insights, as well as practical benefits in many different areas.

Future research directions

Improving the training speed of the online recurrent algorithms that we designed appears as a very natural research direction extending this thesis. Recently, [Cooijmans and Martens, 2019] showed that UORO can be reinterpreted as a version of the REINFORCE [Williams, 1992] algorithm applied to an infinitesimally perturbed recurrent network dynamic. REINFORCE is a popular reinforcement learning algorithm, but is known to suffer from high variance. Adapting reinforcement learning variance mitigation techniques to UORO-like algorithms could be a promising way to speed up learning.

The goal of this study was to improve the core elements that would serve in the design of intelligent agents, that would make use of recurrent models of the world to learn more quickly and efficiently to achieve their goals. What signal to use to train such models of the world is a question of interest to us. A common approach to building world models is to train a recurrent network as a one step predictor of the environment. This amounts to trying to predict the next observation given the history of observations and actions. The question of whether this training objective is still viable when the time discretization becomes small, and more precisely the question of how compounded errors on many very small timesteps affect the predictive power of the model is a topic close to the subjects tackled in this thesis, and that we would like to address in future works.

Another subject we would like to tackle is the use of time discretization robust reinforcement learning algorithms to learn through a curriculum of time discretization, hoping to achieve faster convergence. The idea is that by starting learning at a very coarse discretization, on many problems, we may hope to find rudimentary behaviors that coarsly optimize the reward, and further refining the behavior using progressively smaller time discretizations.

Bibliography

[Arjovsky et al., 2016] Arjovsky, M., Shah, A., and Bengio, Y. (2016). Unitary evolution recurrent neural networks. In *International Conference on Machine Learning*, pages 1120–1128.

[Bahdanau et al., 2014] Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

[Baird, 1994] Baird, L. C. (1994). Reinforcement learning in continuous time: Advantage updating. In *Neural Networks, 1994. IEEE World Congress on Computational Intelligence., 1994 IEEE International Conference on*, volume 4, pages 2448–2453. IEEE.

[Bengio et al., 2013] Bengio, Y., Léonard, N., and Courville, A. (2013). Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*.

[Bengio et al., 1994] Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *Trans. Neur. Netw.*, 5(2):157–166.

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym.

[Chandar et al., 2019] Chandar, S., Sankar, C., Vorontsov, E., Kahou, S. E., and Bengio, Y. (2019). Towards non-saturating recurrent units for modelling long-term dependencies. *CoRR*, abs/1902.06704.

[Chung et al., 2016] Chung, J., Ahn, S., and Bengio, Y. (2016). Hierarchical multiscale recurrent neural networks. *arXiv preprint arXiv:1609.01704*.

[Chung et al., 2014] Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

[Cooijmans et al., 2016a] Cooijmans, T., Ballas, N., Laurent, C., and Courville, A. C. (2016a). Recurrent batch normalization. *CoRR*, abs/1603.09025.

[Cooijmans et al., 2016b] Cooijmans, T., Ballas, N., Laurent, C., Gülçehre, Ç., and Courville, A. (2016b). Recurrent batch normalization. *arXiv preprint arXiv:1603.09025*.

[Cooijmans and Martens, 2019] Cooijmans, T. and Martens, J. (2019). On the variance of unbiased online recurrent optimization. *arXiv preprint arXiv:1902.02405*.

[Doya, 2000] Doya, K. (2000). Reinforcement learning in continuous time and space. *Neural computation*, 12(1):219–245.

[Duchi et al., 2010] Duchi, J., Hazan, E., and Singer, Y. (2010). Adaptive subgradient methods for online learning and stochastic optimization. Technical Report UCB/EECS-2010-24, EECS Department, University of California, Berkeley.

[El Hihi and Bengio, 1995] El Hihi, S. and Bengio, Y. (1995). Hierarchical recurrent neural networks for long-term dependencies. In *Proceedings of the 8th International Conference on Neural Information Processing Systems*, NIPS'95, pages 493–499, Cambridge, MA, USA. MIT Press.

[Espeholt et al., 2018] Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., et al. (2018). Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*.

[Ganea et al., 2018] Ganea, O., Becigneul, G., and Hofmann, T. (2018). Hyperbolic neural networks. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31*, pages 5345–5355. Curran Associates, Inc.

[Gers and Schmidhuber, 2000] Gers, F. A. and Schmidhuber, J. (2000). Recurrent nets that time and count. In *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, volume 3, pages 189–194. IEEE.

[Gers and Schmidhuber, 2001] Gers, F. A. and Schmidhuber, J. (2001). Long short-term memory learns context free and context sensitive languages. In *Artificial Neural Nets and Genetic Algorithms*, pages 134–137. Springer.

[Gers et al., 1999] Gers, F. A., Schmidhuber, J., and Cummins, F. (1999). Learning to forget: Continual prediction with lstm.

[Gers et al., 2000] Gers, F. A., Schmidhuber, J. A., and Cummins, F. A. (2000). Learning to forget: Continual prediction with LSTM. *Neural Comput.*, 12(10):2451–2471.

[Graves, 2013] Graves, A. (2013). Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850.

[Graves et al., 2013] Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE.

[Graves et al., 2014a] Graves, A., Wayne, G., and Danihelka, I. (2014a). Neural turing machines. *arXiv preprint arXiv:1410.5401*.

[Graves et al., 2014b] Graves, A., Wayne, G., and Danihelka, I. (2014b). Neural turing machines. *arXiv preprint arXiv:1410.5401*.

[Graves et al., 2016] Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J., et al. (2016). Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471.

[Gruslys et al., 2016a] Gruslys, A., Munos, R., Danihelka, I., Lanctot, M., and Graves, A. (2016a). Memory-efficient backpropagation through time. In Lee, D. D., Sugiyama, M., von Luxburg, U., Guyon, I., and Garnett, R., editors, *NIPS*, pages 4125–4133.

[Gruslys et al., 2016b] Gruslys, A., Munos, R., Danihelka, I., Lanctot, M., and Graves, A. (2016b). Memory-efficient backpropagation through time. *CoRR*, abs/1606.03401.

[Ha and Schmidhuber, 2018] Ha, D. and Schmidhuber, J. (2018). Recurrent world models facilitate policy evolution. *CoRR*, abs/1809.01999.

[Hansen et al., 2019] Hansen, N., Akimoto, Y., and Baudis, P. (2019). CMA-ES/pycma on Github. Zenodo, DOI:10.5281/zenodo.2559634.

[Hansen and Auger, 2014] Hansen, N. and Auger, A. (2014). Evolution strategies and CMA-ES (covariance matrix adaptation). In *Genetic and Evolutionary Computation Conference, GECCO '14, Vancouver, BC, Canada, July 12-16, 2014, Companion Material Proceedings*, pages 513–534.

[Helfrich et al., 2017] Helfrich, K., Willmott, D., and Ye, Q. (2017). Orthogonal recurrent neural networks with scaled cayley transform. *arXiv preprint arXiv:1707.09520*.

[Henaff et al., 2016] Henaff, M., Szlam, A., and LeCun, Y. (2016). Recurrent orthogonal networks and long-memory tasks. In *International Conference on Machine Learning*, pages 2034–2042.

[Henderson et al., 2017] Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. (2017). Deep reinforcement learning that matters. *CoRR*, abs/1709.06560.

[Hessel et al., 2017] Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. G., and Silver, D. (2017). Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298.

[Hochreiter, 1991] Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.*, 9(8):1735–1780.

[Jaderberg et al., 2016] Jaderberg, M., Czarnecki, W. M., Osindero, S., Vinyals, O., Graves, A., and Kavukcuoglu, K. (2016). Decoupled neural interfaces using synthetic gradients. *CoRR*, abs/1608.05343.

[Jaeger, 2002] Jaeger, H. (2002). Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network" approach.

[Jaeger et al., 2007] Jaeger, H., Lukoševičius, M., Popovici, D., and Siewert, U. (2007). Optimization and Applications of Echo State Networks with Leaky-Integrator Neurons. *Neural Networks*, 20(3):335–352.

[Jing et al., 2019] Jing, L., Gulcehre, C., Peurifoy, J., Shen, Y., Tegmark, M., Soljacic, M., and Bengio, Y. (2019). Gated orthogonal recurrent units: On learning to forget. *Neural computation*, 31(4):765–783.

[Jozefowicz et al., 2015] Jozefowicz, R., Zaremba, W., and Sutskever, I. (2015). An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 2342–2350.

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

[Kingma and Welling, 2013] Kingma, D. P. and Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

[Koutnik et al., 2014] Koutnik, J., Greff, K., Gomez, F., and Schmidhuber, J. (2014). A clockwork rnn. *arXiv preprint arXiv:1402.3511*.

[Lample et al., 2016] Lample, G., Ballesteros, M., Subramanian, S., Kawakami, K., and Dyer, C. (2016). Neural architectures for named entity recognition. *arXiv preprint arXiv:1603.01360*.

[Le et al., 2015] Le, Q. V., Jaitly, N., and Hinton, G. E. (2015). A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*.

[LeCun et al., 1999] LeCun, Y., Haffner, P., Bottou, L., and Bengio, Y. (1999). Object recognition with gradient-based learning. *Shape, contour and grouping in computer vision*, pages 823–823.

[Lillicrap et al., 2015] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971.

[Lucas et al., 2018] Lucas, T., Tallec, C., Verbeek, J., and Ollivier, Y. (2018). Mixed batches and symmetric discriminators for GAN training. *CoRR*, abs/1806.07185.

[Luong et al., 2015] Luong, M.-T., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

[Maass et al., 2002] Maass, W., Natschläger, T., and Markram, H. (2002). Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Comput.*, 14(11):2531–2560.

[Mahoney, 2011] Mahoney, M. (2011). Large text compression benchmark.

[Marcus et al., 1993] Marcus, M. P., Santorini, B., and Marcinkiewicz, M. A. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

[Martens and Sutskever, 2011] Martens, J. and Sutskever, I. (2011). Learning recurrent neural networks with hessian-free optimization. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1033–1040.

[Massé, 2017] Massé, P.-Y. (2017). *Autour de l'Usage des gradients en apprentissage statistique*. PhD thesis, Université Paris Sud.

[Mhammedi et al., 2017] Mhammedi, Z., Hellicar, A., Rahman, A., and Bailey, J. (2017). Efficient orthogonal parametrisation of recurrent neural networks using householder reflections. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2401–2409. JMLR. org.

[Mikolov et al., 2012] Mikolov, T., Sutskever, I., Deoras, A., Hai-Son, L., Kombrink, S., and Černocký, J. (2012). Subword language modeling with neural networks.

[Mnih et al., 2016] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937.

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

[Movellan et al., 2002] Movellan, J. R., Mineiro, P., and Williams, R. J. (2002). A Monte Carlo EM approach for partially observable diffusion processes: Theory and applications to neural networks. *Neural Comput.*, 14(7):1507–1544.

[Mozer, 1992] Mozer, M. C. (1992). Induction of multiscale temporal structure. In *Advances in neural information processing systems*, pages 275–282.

[Mujika et al., 2018] Mujika, A., Meier, F., and Steger, A. (2018). Approximating real-time recurrent learning with random kronecker factors. In *Advances in Neural Information Processing Systems*, pages 6594–6603.

[Munos and Bourgine, 1998] Munos, R. and Bourgine, P. (1998). Reinforcement learning for continuous stochastic control problems. In *Advances in neural information processing systems*, pages 1029–1035.

[Ollivier et al., 2015] Ollivier, Y., Tallec, C., and Charpiat, G. (2015). Training recurrent networks online without backtracking. *CoRR*, abs/1507.07680.

[OpenAI, 2018a] OpenAI (2018a). Learning dexterous in-hand manipulation. *CoRR*, abs/1808.00177.

[OpenAI, 2018b] OpenAI (2018b). Openai five. `https://blog.openai.com/openai-five/`.

[Pascanu et al., 2012] Pascanu, R., Mikolov, T., and Bengio, Y. (2012). Understanding the exploding gradient problem. *CoRR*, abs/1211.5063.

[Pearlmutter, 1995] Pearlmutter, B. A. (1995). Gradient calculations for dynamic recurrent neural networks: A survey. *IEEE Transactions on Neural networks*, 6(5):1212–1228.

[Salimans et al., 2017] Salimans, T., Ho, J., Chen, X., Sidor, S., and Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.

[Saxe et al., 2013] Saxe, A. M., McClelland, J. L., and Ganguli, S. (2013). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*.

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

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

[Siegelmann and Sontag, 1995] Siegelmann, H. T. and Sontag, E. D. (1995). On the computational power of neural nets. *Journal of computer and system sciences*, 50(1):132–150.

[Silver et al., 2017] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

[Simard et al., 1991] Simard, P. Y., Victorri, B., LeCun, Y., and Denker, J. S. (1991). Tangent prop - a formalism for specifying selected invariances in an adaptive network. In Moody, J. E., Hanson, S. J., and Lippmann, R., editors, *NIPS*, pages 895–903. Morgan Kaufmann.

[Steil, 2004] Steil, J. J. (2004). Backpropagation-decorrelation: online recurrent learning with O(N) complexity. In *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, volume 2, pages 843–848 vol.2. IEEE.

[Sutskever, 2013] Sutskever, I. (2013). *Training Recurrent Neural Networks*. PhD thesis, Toronto, Ont., Canada, Canada. AAINS22066.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Introduction to reinforcement learning*, volume 135. MIT press Cambridge.

[Takamura and Yamane, 2019] Takamura, K. and Yamane, S. (2019). Improving minimal gated unit for sequential data. *CoRR*, abs/1906.00748.

[Tallec et al., 2019] Tallec, C., Blier, L., and Ollivier, Y. (2019). Making deep q-learning methods robust to time discretization. *arXiv preprint arXiv:1901.09732*.

[Tallec and Ollivier, 2017a] Tallec, C. and Ollivier, Y. (2017a). Unbiased online recurrent optimization. *arXiv preprint arXiv:1702.05043*.

[Tallec and Ollivier, 2017b] Tallec, C. and Ollivier, Y. (2017b). Unbiasing truncated backpropagation through time. *CoRR*, abs/1705.08209.

[Tallec and Ollivier, 2018] Tallec, C. and Ollivier, Y. (2018). Can recurrent neural networks warp time? *arXiv preprint arXiv:1804.11188*.

[Tesauro, 1995] Tesauro, G. (1995). Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68.

[Thornton et al., 2019] Thornton, M., Anumula, J., and Liu, S. (2019). Reducing state updates via gaussian-gated lstms. *CoRR*, abs/1901.07334.

[Tieleman and Hinton, 2012a] Tieleman, T. and Hinton, G. (2012a). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31.

[Tieleman and Hinton, 2012b] Tieleman, T. and Hinton, G. (2012b). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31.

[Uhlenbeck and Ornstein, 1930] Uhlenbeck, G. E. and Ornstein, L. S. (1930). On the theory of the Brownian motion. *Physical review*, 36(5):823.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

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

[Werbos, 1990] Werbos, P. (1990). Backpropagation through time: what does it do and how to do it. In *Proceedings of IEEE*, volume 78, pages 1550–1560.

[Weston et al., 2014] Weston, J., Chopra, S., and Bordes, A. (2014). Memory networks. *arXiv preprint arXiv:1410.3916*.

[Williams, 1992] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256.

[Williams and Zipser, 1989] Williams, R. J. and Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural Comput.*, 1(2):270–280.

[Wisdom et al., 2016a] Wisdom, S., Powers, T., Hershey, J., Le Roux, J., and Atlas, L. (2016a). Full-capacity unitary recurrent neural networks. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R., editors, *Advances in Neural Information Processing Systems 29*, pages 4880–4888. Curran Associates, Inc.

[Wisdom et al., 2016b] Wisdom, S., Powers, T., Hershey, J., Le Roux, J., and Atlas, L. (2016b). Full-capacity unitary recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 4880–4888.

[Zhang et al., 2018] Zhang, A., Wu, Y., and Pineau, J. (2018). Natural environment benchmarks for reinforcement learning. *CoRR*, abs/1811.06032.

[Zilly et al., 2016] Zilly, J. G., Srivastava, R. K., Koutník, J., and Schmidhuber, J. (2016). Recurrent highway networks. *arXiv preprint arXiv:1607.03474*.

Appendix A

Appendix to Robust Q-learning

A.1 Proofs

We now give proofs for all the results presented in the paper. Most proofs follow standard patterns from calculus and numerical schemes for differential equations, except for Theorem 8, which uses an argument specific to reinforcement learning to prove that the continuous-time advantage function contains all the necessary information for policy improvement.

The first result presented is a proof of convergence for discretized trajectories.

Lemma 1. *Let $F\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}^n$ and $\pi\colon \mathcal{S} \to \mathcal{A}$ be the dynamic and policy functions. Assume that, for any a, $s \to F(s,a)$ and $s \to F(s,\pi(s))$ are $\mathcal{C}^1$, bounded and K-lipschitz. For a given s_0, define the trajectory $(s_t)_{t\geq 0}$ as the unique solution of the differential equation*

$$\frac{ds_t}{dt} = F(s_t, \pi(s_t)). \tag{A.1}$$

For any $\delta t > 0$, define the discretized trajectory $(s^k_{\delta t})_k$ which amounts to maintaining each action for a time interval δt; it is defined by induction as $s^0_{\delta t} = s_0$, $s^{k+1}_{\delta t}$ is the value at time δt of the unique solution of

$$\frac{d\tilde{s}_t}{dt} = F(\tilde{s}_t, \pi(s^k_{\delta t})) \tag{A.2}$$

with initial point $s^k_{\delta t}$. Then, there exists $C > 0$ such that, for every $t \geq 0$

$$\|s_t - s^{\lfloor \frac{t}{\delta t} \rfloor}_{\delta t}\| \leq \delta t \frac{C}{K} e^{Kt}. \tag{A.3}$$

Therefore, discretized trajectories converge pointwise to continuous trajectories.

Proof. The proof mostly follos the classical argument for convergence of the Euler scheme for differential equations. For any k, define

$$e^k_{\delta t} = \|s^k_{\delta t} - s_{\delta t k}\|. \tag{A.4}$$

Let $\tilde{s}_t$ be the solution of Eq. (A.62) with initial state s_δ^k. This $\tilde{s}_t$ is $\mathcal{C}^2$ on $[0, \delta t]$. Consequently, the Taylor integral formula gives

$$s_{\delta t}^{k+1} = s_{\delta t}^k + F(s_{\delta t}^k, \pi(s_{\delta t}^k))\delta t + \int_0^{\delta t} (\delta t - t)\frac{d^2\tilde{s}_t}{dt^2}dt \tag{A.5}$$

$$s_{\delta t(k+1)} = s_{\delta tk} + F(s_{\delta tk}, \pi(s_{\delta tk}))\delta t + \int_0^{\delta t} (\delta t - t)\frac{d^2 s_{t+\delta tk}}{dt^2}dt. \tag{A.6}$$

Now, both $d^2 s_t/dt^2$ and $d^2\tilde{s}_t/dt^2$ are uniformly bounded, by boundedness and Lipschitzness of $s \to F(s, \pi(s))$ and $s \to F(s, \pi(s_{\delta t}^k))$. Consequently, there exists C such that

$$e_{\delta t}^{k+1} \leq \|s_{\delta t}^k - s_{\delta tk}\| + \|F(s_{\delta t}^k, \pi(s_{\delta t}^k)) - F(s_{\delta tk}, \pi(s_{\delta tk}))\|\delta t + C\delta t^2 \tag{A.7}$$

$$\leq (1 + K\delta t)e_{\delta t}^k + C\delta t^2. \tag{A.8}$$

Now, it is easy to prove by induction that

$$e_{\delta t}^k \leq (1 + K\delta t)^k (e_{\delta t}^0 + \frac{C}{K}\delta t) - \frac{C}{K}\delta t. \tag{A.9}$$

As $e_{\delta t}^0 = 0$, this translates to

$$e_{\delta t}^k \leq ((1 + K\delta t)^k - 1)\delta t\frac{C}{K} \tag{A.10}$$

$$\leq (e^{K\delta tk} - 1)\delta t\frac{C}{K}. \tag{A.11}$$

Consequently,

$$e_{\delta t}^{\lfloor t/\delta t \rfloor} \leq (e^{K(t+\delta t)} - 1)\delta t\frac{C}{K}. \tag{A.12}$$

Finally, by boundedness, of $s \to F(s, \pi(s))$, there exists C' such that

$$\|s_{\delta t\lfloor t/\delta t \rfloor} - s_t\| \leq \delta t C'. \tag{A.13}$$

Combining Eq. (A.13) with Eq. (A.12), one can find C'' such that

$$\|s_t - s_{\delta t}^{\lfloor t/\delta t \rfloor}\| \leq \delta t\frac{C''}{K}e^{Kt}. \tag{A.14}$$

□

In what follows, we assume that the continuous-time reward function $r\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is bounded, to ensure existence of V^π and $V_{\delta t}^\pi$ for all δt .

Theorem 6. *Assume that $r\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is bounded, and that $s \to r(s, \pi(s))$ is L_r-Lipschitz continuous, then for all $s \in \mathcal{S}$, one has $V_{\delta t}^\pi(s) = V^\pi(s) + o(1)$ when $\delta t \to 0$.*

Proof. Let $\tilde{s}^t_{\delta t} := s^{\lfloor t/\delta t \rfloor}_{\delta t}$. We have:

$$V^\pi_{\delta t}(s) = \int_t \gamma^t r(\tilde{s}^t_{\delta t}, \pi(\tilde{s}^t_{\delta t}))dt + O(\delta t) \tag{A.15}$$

Indeed:

$$\begin{aligned} V^\pi_{\delta t}(s) =& \sum_{k=0}^{\infty} \gamma^{k\delta t} r(s^k_{\delta t}, \pi(s^k_{\delta t}))\delta t && \text{(A.16)}\\ =& \sum_{k=0}^{\infty} \gamma^{k\delta t} \int_{u=k}^{k+1} r(\tilde{s}^{u\delta t}_{\delta t}, \pi(\tilde{s}^{u\delta t}_{\delta t}))du && \text{(A.17)}\\ =& \sum_{k=0}^{\infty} \frac{\delta t \log\gamma}{\gamma^{\delta t}-1} \int_{u=k}^{k+1} \gamma^{u\delta t} r(\tilde{s}^{u\delta t}_{\delta t}, \pi(\tilde{s}^{u\delta t}_{\delta t}))du && \text{(A.18)}\\ =& \frac{\delta t \log\gamma}{\gamma^{\delta t}-1} \int_{t=0}^{\infty} \gamma^t r(\tilde{s}^t_{\delta t}, \pi(\tilde{s}^t_{\delta t}))dt && \text{(A.19)}\\ & && \text{(A.20)} \end{aligned}$$

But:

$$\begin{aligned} \frac{\delta t \log\gamma}{\gamma^{\delta t}-1} &= \frac{\delta t \log\gamma}{\delta t \log\gamma + O(\delta t^2)} && \text{(A.21)}\\ &= 1 + O(\delta t) && \text{(A.22)}\\ & && \text{(A.23)} \end{aligned}$$

Therefore:

$$V^\pi_{\delta t}(s) = \int_t \gamma^t r(\tilde{s}^t_{\delta t}, \pi(\tilde{s}^t_{\delta t}))dt + O(\delta t) \tag{A.24}$$

We now have, for any $T > 0$,

$$\begin{aligned} |V^\pi_{\delta t}(s) - V^\pi(s)| =& \left| \int_{t=0}^{\infty} \gamma^t \left(r(\tilde{s}^t_{\delta t}, \pi(\tilde{s}^t_{\delta t})) - r(s_t, \pi(s_t)) \right) dt \right| + O(\delta t) && \text{(A.25)}\\ =& \left| \int_{t=0}^{T} \gamma^t \left(r(\tilde{s}^t_{\delta t}, \pi(\tilde{s}^t_{\delta t})) - r(s_t, \pi(s_t)) \right) dt \right| && \text{(A.26)}\\ &+ \left| \int_{t=T}^{\infty} \gamma^t \left(r(\tilde{s}^t_{\delta t}, \pi(\tilde{s}^t_{\delta t})) - r(s_t, \pi(s_t)) \right) dt \right| + O(\delta t) && \text{(A.27)} \end{aligned}$$

The second term can be bounded by the supremum of the reward:

$$\left| \int_{t=T}^{\infty} \gamma^t \left(r(\tilde{s}^t_{\delta t}, \pi(\tilde{s}^t_{\delta t})) - r(\tilde{s}_t, \pi(s_t)) \right) dt \right| \leq 2 \frac{\|r\|_\infty}{\log(\frac{1}{\gamma})} \gamma^T \tag{A.28}$$

The first term can be bounded by using Lemma. 1:

$$|\int_{t=0}^{T}\gamma^t\left(r(\tilde{s}^t_{\delta t},\pi(\tilde{s}^t_{\delta t}))-r(s_t,\pi(s_t))\right)dt| \leq \int_{t=0}^{T}\gamma^t L_r\|s_t-\tilde{s}^t_{\delta t}\|dt \tag{A.29}$$

$$\leq \int_{t=0}^{T} L_r\frac{C\delta t}{K}\exp((K+\log\gamma)t)dt \tag{A.30}$$

$$\leq \frac{L_rC}{K(K+\log\gamma)}\exp((K+\log\gamma)T)\delta t \tag{A.31}$$

Let us set $T := -\frac{1}{K}\log(\delta t)$. By plugging into Eq. (A.28), we have:

$$|\int_{t=T}^{\infty}\gamma^t\left(r(\tilde{s}^t_{\delta t},\pi(\tilde{s}^t_{\delta t}))-r(s_t,\pi(s_t))\right)dt| = O(\delta t^{-\log\gamma}) = o(1). \tag{A.32}$$

By plugging T into equation (A.31), we have:

$$|\int_{t=0}^{T}\gamma^t\left(r(\tilde{s}^t_{\delta t},\pi(\tilde{s}^t_{\delta t}))-r(s_t,\pi(s_t))\right)dt| = O(\delta t^{-\frac{\log\gamma}{K}}) = o(1), \tag{A.33}$$

yielding our result.

□

For the following proof, we further assume that both V^π and $V^\pi_{\delta t}$ are continuously differentiable, and that the gradient and Hessian of $V^\pi_{\delta t}$ w.r.t. s are uniformly bounded in both s and δt. We also assume convergence of $\partial_s V^\pi_{\delta t}(s)$ to $\partial_s V^\pi(s)$ for all s.

Theorem 7. *Under the hypothesis above, there exists $A^\pi\colon \mathcal{S}\to\mathbb{R}$ such that $A^\pi_{\delta t}$ converges pointwise to A^π as δt goes to 0. Besides,*

$$A^\pi(s,a) = r(s,a) + \partial_s V^\pi(s)F(s,a) + \log\gamma V^\pi(s). \tag{A.34}$$

Proof. Denote $\tilde{s}^t_{\delta t}(s_0)$ the evaluation at instant t of the solution of $d\tilde{s}_t/dt = F(\tilde{s}_t,\pi(s_0))$ with starting point s_0.

The Bellman equation on $Q^\pi_{\delta t}$ yields

$$Q^\pi_{\delta t}(s,a) = r(s,a)\delta t + \gamma^{\delta t}V^\pi_{\delta t}(\tilde{s}^{\delta t}_{\delta t}(s)). \tag{A.35}$$

For all s, a first-order Taylor expansion yields

$$\tilde{s}^{\delta t}_{\delta t}(s) = s + F(s,a)\delta t + O(\delta t^2) \tag{A.36}$$

where the constant in $O()$ is uniformly bounded thanks to the assumptions on the Hessian. Thus, by uniform boundedness of the Hessian of $V^\pi_{\delta t}$,[1]

$$Q^\pi_{\delta t}(s,a) = r(s,a)\delta t + (1+\ln(\gamma)\delta t + O(\delta t^2))(V^\pi_{\delta t}(s) + \delta t\partial_s V^\pi_{\delta t}(s)F(s,a) + O(\delta t^2). \tag{A.37}$$

[1] Without boundedness of the Hessian, we cannot write the second order Taylor expansion of $V^\pi_{\delta t}(\tilde{s}^{\delta t}_{\delta t}(s))$ in term of δt.

Now, this yields

$$A^{\pi}_{\delta t}(s,a) = r(s,a) + \ln(\gamma)V^{\pi}_{\delta t}(s) + \partial_s V^{\pi}_{\delta t}(s)F(s,a) + O(\delta t), \tag{A.38}$$

and using the convergence of $V^{\pi}_{\delta t}(s)$ to $V^{\pi}(s)$ (Thm. 6) and $\partial_s V^{\pi}_{\delta t}(s)$ to $\partial_s V^{\pi}(s)$ (hypothesis) yields the result with

$$A^{\pi}(s,a) = r(s,a) + \ln(\gamma)V^{\pi}(s) + \partial_s V^{\pi}(s)F(s,a). \tag{A.39}$$

□

We now show that policy improvement works with the continous time advantage function, i.e.

Theorem 8. *Let π and π' be two policies such that both $s \to r(s,\pi(s))$ and $s \to r(s,\pi'(s))$ are continuous. Assume that both V^{π} and $V^{\pi'}$ are continuously differentiable. Define the advantage function for policies π and π' as in Eq.* (A.39).

If for all s, $A^{\pi}(s,\pi'(s)) \geq 0$, then for all s, $V^{\pi}(s) \leq V^{\pi'}(s)$. Moreover, if for all s, $V^{\pi'}(s) > V^{\pi}(s)$, then there exists s' such that $A^{\pi}(s',\pi'(s')) > 0$.

Proof. Let $(s_t)_{t\geq 0}$ be a trajectory sampled from π' i.e. solution of the equation

$$ds_t/dt = F(s_t, \pi'(s_t)) \tag{A.40}$$

with initial condition $s_0 = s$.

Define

$$B(T) = \int_{t=0}^{T} \gamma^t r(s_t,\pi'(s_t))dt + \gamma^T V^{\pi}(s_T). \tag{A.41}$$

This function if continuously differentiable, and its derivative is

$$\dot{B}(T) = \gamma^T r(s_T,\pi'(s_T)) + \gamma^T \partial_s V^{\pi}(s)F(s,\pi'(s)) + \gamma^T \ln(\gamma)V^{\pi}(s_T) \tag{A.42}$$

$$= \gamma^T A^{\pi}(s_T,\pi'(s_T)) \qquad \geq 0. \tag{A.43}$$

Thus B is increasing, and $B(0) = V^{\pi}(s)$, $\lim_{T\to\infty} B(t) = V^{\pi'}(s)$. Consequently, $V^{\pi}(s) \leq V^{\pi'}(s)$. Furthermore, if $V^{\pi}(s) < V^{\pi'}(s)$, then there exists T_0 such that $\dot{B}(T_0) > 0$ (otherwise B is constant), and $A^{\pi}(s_{T_0},\pi'(s_{T_0})) > 0$.

□

Theorem 9. *Let $\mathcal{A} = \mathbb{R}^{\mathcal{A}}$ be the action space, and let $\mathcal{P}_1 = \mathbb{R}^{p_1}$ and $\mathcal{P}_2 = \mathbb{R}^{p_2}$ be parameter spaces. Let $A\colon \mathcal{P}_1 \times \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ and $V\colon \mathcal{P}_2 \times \mathcal{S} \to \mathbb{R}$ be $\mathcal{C}^2$ function approximators with bounded gradients and Hessians. Let $(a_t)_{t\geq 0}$ be a $\mathcal{C}^1$ exploratory action trajectory and $(s_t)_{t\geq 0}$ the resulting state trajectory, when starting from s_0 and following $ds_t/dt = F(s_t,a_t)$. Let $\theta^k_{\delta t}$ and $\psi^k_{\delta t}$ be the discrete parameter trajectories resulting from the gradient descent steps in the main text, with learning rates $\eta^V = \alpha^V \delta t^{\beta}$ and $\eta^A = \alpha^A \delta t^{\beta}$ for some $\beta \geq 0$. Then,*

- *If $\beta = 1$ the discrete parameter trajectories converge to continuous parameter trajectories as δt goes to 0.*
- *If $\beta > 1$, parameter trajectories become stationary as δt goes to 0.*
- *If $\beta < 1$, parameters can grow arbitrarily large after an arbitrarily small physical time when δt goes to 0.*

Proof. Let $(s_t, a_t)_{t\geq 0}$ be the trajectory on which parameters are learnt. To simplify notations, define

$$A_\psi(s,a) = \bar{A}_\psi(s,a) - \bar{A}_\psi(s,\pi(s)). \tag{A.44}$$

Define F as

$$F^\theta(\theta,\psi,s,a) = \alpha^V(r(s,a) + \ln(\gamma)V_\theta(s) + \partial_s V_\theta(s)F(s,a) - A_\psi(s,a))\partial_\theta V_\theta(s) \tag{A.45}$$

$$F^\psi(\theta,\psi,s,a) = \alpha^A(r(s,a) + \ln(\gamma)V_\theta(s) + \partial_s V_\theta(s)F(s,a) - A_\psi(s,a))\partial_\psi A_\psi(s,a). \tag{A.46}$$

From the bounded Hessians and Gradients hypothesis, V, A, $\partial_s V$, $\partial_\theta V$ and $\partial_\psi A$ are uniformly Lipschitz continuous in θ and ψ, thus F is Lipschitz continuous.

The discrete equations for parameters updates with learning rates $\alpha^V \delta t^\beta$ and $\alpha^A \delta t^\beta$ are

$$\delta Q = r(s_{k\delta t}, a_{k\delta t})\delta t + \gamma^{\delta t} V_{\theta^k_{\delta t}}(s_{(k+1)\delta t}) - V_{\theta^k_{\delta t}}(s_{k\delta t}) - A_\psi(s_{k\delta t}, a_{k\delta t}) \tag{A.47}$$

$$\theta^{k+1}_{\delta t} = \theta^k_{\delta t} + \alpha^V \delta t^\beta \frac{\delta Q}{\delta t} \partial_\theta V_{\theta^k_{\delta t}}(s_{k\delta t}) \tag{A.48}$$

$$\psi^{k+1}_{\delta t} = \psi^k_{\delta t} + \alpha^A \delta t^\beta \frac{\delta Q}{\delta t} \partial_\psi A_{\theta^k_{\delta t}}(s_{k\delta t}, a_{k\delta t}) \tag{A.49}$$

Under uniform boundedness of the Hessian of $s \mapsto V_\theta(s)$, one can show

$$\begin{pmatrix} \theta^{k+1}_{\delta t} \\ \psi^{k+1}_{\delta t} \end{pmatrix} = \begin{pmatrix} \theta^k_{\delta t} \\ \psi^k_{\delta t} \end{pmatrix} + \delta t^\beta F(\theta^k_{\delta t}, \psi^k_{\delta t}, s_{k\delta t}, a_{k\delta t}) + O(\delta t^\beta \delta t), \tag{A.50}$$

with a O independent of k. With the additional hypothesis that the gradient of $(s,a) \to \bar{A}_\psi(s,a)$ is uniformly bounded, we have

- For $\beta = 1$, a proof scheme identical to that of Thm. 1 shows that discrete trajectories converge pointwise to continuous trajectories defined by the differential equation

  $$\frac{d}{dt}\begin{pmatrix} \theta_t \\ \psi_t \end{pmatrix} = F(\theta_t, \psi_t, s_t, a_t), \tag{A.51}$$

  which admits unique solutions for all initial parameters, since F is uniformly lipschitz continuous.

- Similarly, for $\beta > 1$, the proof scheme of Thm. 1 shows that discrete trajectories converge pointwise to continuous trajectories defined by the differential equation

$$\frac{d}{dt}\begin{pmatrix}\theta_t\\ \psi_t\end{pmatrix} = 0 \tag{A.52}$$

  and thus that trajectories shrink to a single point as δt goes to 0.

We now turn to proving that when $\beta < 1$, trajectories can diverge instantly in physical time. Consider the following continuous MDP,

$$s_t = \sin(t) \tag{A.53}$$

whatever the actions, with reward 0 everywhere and $0 < \gamma < 1$. The resulting value function is $V(s) = 0$ (since there are no actions, V is independent of a policy), and the advantage function is 0. We consider the function approximator $V_\theta(s) = \theta s$ (which can represent the true value function). The update rule for θ is

$$\delta Q^k_{\delta t} = \gamma^{\delta t}\theta^k_{\delta t}\sin((k+1)\delta t) - \theta^k_{\delta t}\sin(k\delta t) \tag{A.54}$$

$$\theta^{k+1}_{\delta t} = \theta^k_{\delta t} + \alpha\delta t^\beta \frac{\gamma^{\delta t}\theta^k_{\delta t}\sin((k+1)\delta t) - \theta^k_{\delta t}\sin(k\delta t)}{\delta t}\sin(k\delta t) \tag{A.55}$$

Set $K_{\delta t} := \lfloor \delta t^{-\frac{\beta+3}{4}} \rfloor$, then for all $k \leq K_{\delta t}$, $o(k\delta t) = o(1)$ and

$$\theta^{k+1}_{\delta t} = \theta^k_{\delta t}(1 + \alpha\delta t^\beta(1 + o(1))\sin(k\delta t)) \tag{A.56}$$

$$\tag{A.57}$$

Let $\rho^k_{\delta t} := \log\theta^k_{\delta t}$. Then

$$\rho^k_{\delta t} = \rho^k_{\delta t} + \alpha k\delta t^{\beta+1} + o(k\delta t^{\beta+1}). \tag{A.58}$$

Finally,

$$\rho^{K_{\delta t}}_{\delta t} = \rho^0_{\delta t} + \alpha\frac{K_{\delta t}(K_{\delta t}+1)}{2}\delta t^\beta + o(K^2_{\delta t}\delta t^{\beta+1}) \tag{A.59}$$

$$= \rho^0_\delta + \alpha\delta t^{\frac{\beta-1}{3}} + o(\delta t^{\frac{\beta-1}{3}}) \tag{A.60}$$

$$\xrightarrow[\delta t\to 0]{} +\infty. \tag{A.61}$$

Thus parameters diverge in an infinitesimal physical time when δt goes to 0.

□

Theorem 10. *Let $F\colon \mathcal{S}\times\mathcal{A} \to \mathbb{R}^n$ be the dynamic, and $\pi\colon \mathcal{S}\times\mathcal{A}\to[0,1]$ be the policy, such that $\pi(s,\cdot)$ is a probability distribution over $\mathcal{A}$. Assume that F is C^1 with bounded derivatives, and that π is C^1 and bounded. For any $\delta t > 0$, define the discretized trajectory $(s^k_{\delta t})_k$ which amounts to sample an action from $\pi(s,\cdot)$ and maintaining each*

action for a time interval δt; it is defined by induction as $s^0_{\delta t} = s_0$, $s^{k+1}_{\delta t}$ is the value at time δt of the unique solution of

$$\frac{d\tilde{s}_t}{dt} = F(\tilde{s}_t, a_k) \tag{A.62}$$

with $a_k \sim \pi(s^k_{\delta t}, \cdot)$ and initial point $s^k_{\delta t}$.

Then the agent's trajectories converge when $\delta t \to 0$ to the solutions of the deterministic *equation:*

$$\frac{ds_t}{dt} = \mathbb{E}_{a\sim\pi(s_t,\cdot)} F(s_t, a). \tag{A.63}$$

Notably, if π is an epsilon greedy strategy that mixes a deterministic exploitation policy $\pi^{deterministic}$ with an action taken from a noise policy π^{noise} with probability ε at, the trajectory converge to the solutions of the equation:

$$ds_t/dt = (1-\varepsilon)F(s_t, \pi^{deterministic}(s_t)) + \varepsilon \mathbb{E}_{a\sim\pi^{noise}(a|s)} F(s_t, a) \tag{A.64}$$

Proof. Consider $(s_{\delta t^2})$ the random trajectory of a near-continuous MDP with time-discretization δt^2 obtained by taking at each step k an action a_k along $a_k \sim \pi(a|s^k_{\delta t^2})$ independantly. We have:

$$s^{\lfloor 1/\delta t \rfloor}_{\delta t^2} = s^0_{\delta t^2} + \sum_{k=1}^{\lfloor 1/\delta t \rfloor} s^k_{\delta t^2} - s^{k-1}_{\delta t^2} + O(\delta t^2) \tag{A.65}$$

$$= s^0_{\delta t^2} + \sum_{k=1}^{\lfloor 1/\delta t \rfloor} F(s^{k-1}_{\delta t^2}, a_{k-1})\delta t^2 + O(\delta t^2) \tag{A.66}$$

We define $f(s) := \mathbb{E}_{a\sim\pi(s)}[F(s,a)] = \int_{a\in\mathcal{A}} F(s,a)\pi(s,a)$. Since π and F are bounded and C^1, we know that f is C^1. We have:

$$s^{\lfloor 1/\delta t \rfloor}_{\delta t^2} = s^0_{\delta t^2} + \sum_{k=1}^{\lfloor 1/\delta t \rfloor} f(s^{k-1}_{\delta t^2})\delta t^2 + \sum_{k=1}^{\lfloor 1/\delta t \rfloor} (F(s^{k-1}_{\delta t^2}, a_{k-1}) - f(s^{k-1}_{\delta t^2}))\delta t^2 + O(\delta t^2) \tag{A.67}$$

$$s^{\lfloor 1/\delta t \rfloor}_{\delta t^2} = s^0_{\delta t^2} + \sum_{k=1}^{\lfloor 1/\delta t \rfloor} f(s^{k-1}_{\delta t^2})\delta t^2 + \xi + O(\delta t^2) \tag{A.68}$$

with $\xi := \delta t^2 \sum_{k=1}^{\lfloor 1/\delta t \rfloor} \left(F(s^{k-1}_{\delta t^2}, a_{k-1}) - f(s^{k-1}_{\delta t^2}) \right)$. By definition, we have $\mathbb{E}[\xi] = 0$. Moreover, by using the independance of actions and the boundness of F, there is $\sigma > 0$ such that:

$$\mathbb{E}[\|\xi\|^2] \leq \sigma^2 \delta t^3 \tag{A.69}$$

We know that f is C^1 on a compact space. Therefore, there is L_f such that f is L_f Lipschitz, and we have:

$$\| \left(\sum_{k=1}^{\lfloor 1/\delta t \rfloor} f(s_{\delta t^2}^{k-1}) \delta t \right) - f(s_{\delta t^2}^0) \| \leq \delta t L_f \sum_{k=1}^{\lfloor 1/\delta t \rfloor} \| s_{\delta t^2}^{k-1} - s_{\delta t^2}^0 \| \tag{A.70}$$

Since F is bounded, we know that $\| s_{\delta t^2}^k - s_{\delta t^2}^{k-1} \| \leq C \delta t$. Therefore:

$$\| \left(\sum_{k=1}^{\lfloor 1/\delta t \rfloor} f(s_{\delta t^2}^{k-1}) \delta t \right) - f(s_{\delta t^2}^0) \| \leq \delta t L_f C \sum_{k=1}^{\lfloor 1/\delta t \rfloor} k \delta t \tag{A.71}$$

$$= O(\delta t^2) \tag{A.72}$$

Therefore:

$$s_{\delta t^2}^{\lfloor 1/\delta t \rfloor} = s_{\delta t^2}^0 + f(s_{\delta t^2}^0) \delta t + \xi + O(\delta t^2) \tag{A.73}$$

Therefore, we have $a > 0$ such that $\| s_{\delta t^2}^{\lfloor 1/\delta t \rfloor} - s_{\delta t^2}^0 - f(s_{\delta t^2}^0) \delta t \| \leq \| \xi \| + a \delta t^2$

We define $(\tilde{s}_{\delta t})$ the deterministic near-continuous process with time discretization δt defined by $\tilde{s}_{\delta t}^{k+1} := s_{\delta t}^k + f(s_{\delta t}^k) \delta t$. We have:

$$\| s_{\delta t^2}^{(k+1)\lfloor 1/\delta t \rfloor} - \tilde{s}_{\delta t}^{k+1} \| \leq \| s_{\delta t^2}^{(k+1)\lfloor 1/\delta t \rfloor} - s_{\delta t^2}^{k\lfloor 1/\delta t \rfloor} - f(s_{\delta t^2}^{k\lfloor 1/\delta t \rfloor}) \delta t \| + \| s_{\delta t^2}^{k\lfloor 1/\delta t \rfloor} + f(s_{\delta t^2}^{k\lfloor 1/\delta t \rfloor}) \delta t - \tilde{s}_{\delta t}^{k+1} \| \tag{A.74}$$

We know that $\| s_{\delta t^2}^{(k+1)\lfloor 1/\delta t \rfloor} - s_{\delta t^2}^{k\lfloor 1/\delta t \rfloor} - f(s_{\delta t^2}^{k\lfloor 1/\delta t \rfloor}) \delta t \| \leq \| \xi_k \| + a \delta t^2$. Moreover:

$$\| s_{\delta t^2}^{k\lfloor 1/\delta t \rfloor} + f(s_{\delta t^2}^{k\lfloor 1/\delta t \rfloor}) \delta t - \tilde{s}_{\delta t}^{k+1} \| \leq \| s_{\delta t^2}^{k\lfloor 1/\delta t \rfloor} - \tilde{s}_{\delta t}^k \| + \delta t \| f(s_{\delta t^2}^{k\lfloor 1/\delta t \rfloor}) - f(\tilde{s}_{\delta t}^k) \| \tag{A.75}$$

$$\leq (1 + L_f \delta t) \| s_{\delta t^2}^{k\lfloor 1/\delta t \rfloor} - \tilde{s}_{\delta t}^k \| \tag{A.76}$$

Therefore, we have:

$$\| s_{\delta t^2}^{(k+1)\lfloor 1/\delta t \rfloor} - \tilde{s}_{\delta t}^{k+1} \| \leq \| \xi_k \| + a \delta t^2 + (1 + L_f \delta t) \| s_{\delta t^2}^{k\lfloor 1/\delta t \rfloor} - \tilde{s}_{\delta t}^k \| \tag{A.77}$$

By induction, and by taking $k = \lfloor t/\delta t \rfloor$:

$$\| s_{\delta t^2}^{k\lfloor 1/\delta t \rfloor} - \tilde{s}_{\delta t}^k \| \leq \frac{a \delta t}{L_f} \exp(L_f t) + \sum_{j=0}^{\lfloor t/\delta t \rfloor} (1 + \delta t L_f)^j \| \xi_j \| \tag{A.78}$$

Therefore, if $\varepsilon > 0$, we have :

$$\mathbb{P}\left(\|s_{\delta t^2}^{k\lfloor 1/\delta t\rfloor} - \tilde{s}_{\delta t}^k\| > \varepsilon\right) \leq \mathbb{P}\left(\sum_{j=0}^{\lfloor t/\delta t\rfloor} (1+\delta t L_f)^j \|\xi_j\| > \varepsilon - \frac{a\delta t}{L_f}\exp(L_f t)\right) \tag{A.79}$$

$$\leq \frac{\mathbb{E}\left[\sum_{j=0}^{\lfloor t/\delta t\rfloor}(1+\delta t L_f)^j\|\xi_j\|\right]}{\varepsilon - \frac{a\delta t}{L_f}\exp(L_f t)} \tag{A.80}$$

$$\leq \frac{\mathbb{E}\left[\|\xi\|\right]}{\varepsilon - \frac{a\delta t}{L_f}\exp(L_f t)}\frac{\exp(L_f t)}{L_f \delta t} \tag{A.81}$$

$$\tag{A.82}$$

But $\mathbb{E}\left[\|\xi\|\right] \leq \sqrt{\mathbb{E}\left[\|\xi\|^2\right]} \leq \sigma\delta t^{3/2}$. Therefore, we have:

$$\mathbb{P}\left(\|s_{\delta t^2}^{\lfloor t/\delta t\rfloor\lfloor 1/\delta t\rfloor} - \tilde{s}_{\delta t}^k\| > \varepsilon\right) = O(\sqrt{\delta t}) \tag{A.83}$$

Therefore, the process $t \mapsto s_{\delta t^2}^{\lfloor t/\delta t\rfloor\lfloor 1/\delta t\rfloor}$ converges in probability to $\tilde{s}$. Furthermore, by a similar argument than in Lemma 1, we know that the discretized process $\tilde{s}$ converge to the continuous process defined by $\frac{ds}{dt} = f(s_t)$. We can conclude ou result.

□

A.2 Implementation details

All the details specifying our implementation are given in this section. We first give precise pseudo code descriptions for both *Continuous Deep Advantage Updating* (Alg. 5), as well as the variants of DDPG (Alg. 6) and DQN (Alg. 7) used.

For DDPG and DQN, two different settings were experimented with:

- One with time discretization scalings, to keep the comparison fair. In this setting, the discount factor is still scaled as $\gamma^{\delta t}$, rewards are scaled as $r\delta t$, and learning rates are scaled to obtain parameter updates of order δt. As RMSprop is used for all experiments, this amounts to using a learning rate scaling as $\alpha^Q = \tilde{\alpha}^Q\delta t$, $\alpha^\pi = \tilde{\alpha}^\pi\delta t$.

- One without discretization scalings. In that case, only the discount factor is scaled as $\gamma^{\delta t}$, to prevent unfair shortsightedness. All other parameters are set with a reference $\delta t_0 = 1e-2$. For instance, for all δt's, the reward perceived is $r * \delta t_0$, and similarily for learning rates, $\alpha^Q = \tilde{\alpha}^Q\delta t_0$, $\alpha^\pi = \tilde{\alpha}^Q\delta t_0$. These scalings don't depend on the discretization, but perform decently at least for the highest discretization.

A.2.1 Global hyperparameters

The following hyperparameters are maintained constant throughout all our experiments,

Algorithm 5 Discrete DAU

Inputs:
θ and ψ, parameters of V_θ and $\bar{A}_\psi$.
π^{explore} and $\nu_{\delta t}$ defining an exploration policy.
$\mathbf{opt}_V$, $\mathbf{opt}_A$, $\alpha^V \delta t$ and $\alpha^A \delta t$ optimizers and learning rates.
$\mathcal{D}$, buffer of transitions (s, a, r, d, s'), with d the episode termination signal.
δt and γ, time discretization and discount factor.
nb_epochs number of epochs.
nb_steps, number of steps per epoch.
nb_learn, number of learning step per epoch
N, training batch size

Observe initial state s^0
$t \leftarrow 0$
for $e = 0,$ **nb_epochs do**
  for $j = 1,$ **nb_steps do**
    $a^t \leftarrow \pi^{\text{explore}}(s^t, \nu^t_{\delta t})$.
    Perform a^t and observe $(r^{t+1}, d^{t+1}, s^{t+1})$.
    Store $(s^t, a^t, r^{t+1}, d^{t+1}, s^{t+1})$ in $\mathcal{D}$.
    $t \leftarrow t + 1$
  end for
  for $k = 0,$ **nb_learn do**
    Sample a batch of N random transitions from $\mathcal{D}$
    for i=0, N **do**

$$\delta Q^i \leftarrow \delta t\big(\bar{A}_\psi(s^i, a^i) - \bar{A}_\psi(s^i, \pi_\phi(s^i))\big) - \Big(r^i \delta t + (1 - d^i)\gamma^{\delta t} V_\theta(s'^i) - V_\theta(s^i)\Big)$$

    end for

$$\Delta\theta \leftarrow \frac{1}{N} \sum_{i=1}^{N} \frac{\delta Q^i \partial_\theta V_\theta(s^i)}{\delta t}$$

$$\Delta\psi \leftarrow \frac{1}{N} \sum_{i=1}^{N} \frac{\delta Q^i \partial_\psi \left(\bar{A}_\psi(s^i, a^i) - \max_{a'} \bar{A}_\psi(s^i, a')\right)}{\delta t}$$

    Update θ with $\mathbf{opt}_1$, $\Delta\theta$ and learning rate $\alpha^V \delta t$.
    Update ψ with $\mathbf{opt}_2$, $\Delta\psi$ and learning rate $\alpha^A \delta t$.
  end for
end for

Algorithm 6 DDPG

Inputs:
ψ and ϕ, parameters of Q_ψ and π_ϕ.
ψ' and ϕ', parameters of target networks $Q_{\psi'}$ and $\pi_{\phi'}$.
π^{explore} and ν defining an exploration policy.
$\mathbf{opt}_Q$, $\mathbf{opt}_\pi$, α^Q and α^π, optimizers and learning rates.
$\mathcal{D}$, buffer of transitions (s, a, r, d, s'), with d the episode termination signal.
γ discount factor.
τ target network update factor.
nb_epochs number of epochs.
nb_steps, number of steps per epoch.

Observe initial state s^0
$t \leftarrow 0$
for $e = 0$, **nb_epochs do**
  for $j = 1$, **nb_steps do**
    $a^k \leftarrow \pi^{\text{explore}}(s^k, \nu^k)$.
    Perform a^k and observe $(r^{k+1}, d^{k+1}, s^{k+1})$.
    Store $(s^k, a^k, r^{k+1}, d^{k+1}, s^{k+1})$ in $\mathcal{D}$.
    $k \leftarrow k + 1$
  end for
  for $k = 0$, nb_learn **do**
    Sample a batch of N random transitions from $\mathcal{D}$
    $\tilde{Q}^i \leftarrow r^i + (1 - d^i)\gamma Q_{\psi'}(s'^i, \pi_{\phi'}(s'^i))$
    $\Delta\psi \leftarrow \frac{1}{N} \sum_{i=1}^{N} \left(Q^i - \tilde{Q}^i \right) \partial_\psi Q(s^i, a^i)$
    $\Delta\phi \leftarrow \frac{1}{N} \sum_{i=1}^{N} \partial_a Q_\psi(s^i, \pi_\phi(s^i)) \partial_\phi \pi_\phi(s^i)$
    Update ψ with $\mathbf{opt}_Q$, $\Delta\psi$ and learning rate α^Q.
    Update ϕ with $\mathbf{opt}_\pi$, $\Delta\phi$ and learning rate α^π.
    $\psi' \leftarrow \tau\psi' + (1 - \tau)\psi$
    $\phi' \leftarrow \tau\phi' + (1 - \tau)\phi$
  end for
end for

Algorithm 7 DQN

Inputs:
ψ parameter of Q_ψ.
ψ', parameters of target networks $Q_{\psi'}$.
π^{explore} and ν defining an exploration policy.
$\mathbf{opt}_Q$, α^Q optimizer and learning rate.
$\mathcal{D}$, buffer of transitions (s, a, r, d, s'), with d the episode termination signal.
γ discount factor.
τ target network update factor.
nb_epochs number of epochs.
nb_steps, number of steps per epoch.

Observe initial state s^0
$t \leftarrow 0$
for $e = 0,$ **nb_epochs do**
  for $j = 1,$ **nb_steps do**
    $a^k \leftarrow \pi^{\text{explore}}(s^k, \nu^k)$.
    Perform a^k and observe $(r^{k+1}, d^{k+1}, s^{k+1})$.
    Store $(s^k, a^k, r^{k+1}, d^{k+1}, s^{k+1})$ in $\mathcal{D}$.
    $k \leftarrow k + 1$
  end for
  for $k = 0,$ nb_learn **do**
    Sample a batch of N random transitions from $\mathcal{D}$
    $\tilde{Q^i} \leftarrow r^i + (1 - d^i)\gamma \max_{a'} Q_{\psi'}(s'^i, a')$
    $\Delta\psi \leftarrow \frac{1}{N} \sum_{i=1}^{N} \left(Q^i - \tilde{Q^i}\right) \partial_\psi Q(s^i, a^i)$
    Update ψ with $\mathbf{opt}_Q$, $\Delta\psi$ and learning rate α^Q.
    $\psi' \leftarrow \tau\psi' + (1 - \tau)\psi$
  end for
end for

- All networks used are of the form

```
Sequential(
    Linear(nb_inputs, 256),
    LayerNorm(256),
    ReLU(),
    Linear(256, 256),
    LayerNorm(256),
    ReLU(),
    Linear(256, nb_outputs)
).
```

  Policy networks have an additional tanh layer to constraint action range. On certain environments, network inputs are normalized by applying a mean-std normalization, with mean and standard deviations computed on each individual input features, on all previously encountered samples.

- $\mathcal{D}$ is a cyclic buffer of size 1000000.
- **nb_steps** is set to 10, and 256 environments are run in parallel to accelerate the training procedure, totalling 2560 environment interactions between learning steps.
- **nb_learn** is set to 50.
- The physical γ is set to 0.8. It is always scaled as $\gamma^{\delta t}$ (even for unscaled DQN and DDPG).
- N, the batch size is set to 256.
- RMSprop is used as an optimizer without momentum, and with $\alpha = 1 - \delta t$ (or $1 - \delta t_0$ for unscaled DDPG and DQN).
- Exploration is always performed as described in the main text. The OU process used as parameters $\kappa = 7.5$, $\sigma = 1.5$.
- Unless otherwise stated, $\alpha_1 := \tilde{\alpha}^Q = \alpha^V = \alpha^A = 0.1$, $\alpha_2 := \tilde{\alpha}^\pi = \alpha^\pi = 0.03$.
- $\tau = 0.9$

A.2.2 Environment dependent hyperparameters

We hereby list the hyperparameters used for each environment. Continuous actions environments are marked with a (C), discrete actions environments with a (D).

- **Ant (C)**: State normalization is used. Discretization range: $[0.05, 0.02, 0.01, 0.005, 0.002]$.
- **Cheetah (C)**: State normalization is used. Discretization range: $[0.05, 0.02, 0.01, 0.005, 0.002]$

- **Bipedal Walker (C)**[2]: State normalization is used, $\alpha_2 = 0.02$. Discretization range: $[0.01, 0.005, 0.002, 0.001]$.
- **Cartpole (D)**: $\alpha_2 = 0.02$, $\tau = 0$. Discretization range: $[0.01, 0.005, 0.002, 0.001, 0.0005]$.
- **Pendulum (C)**: $\alpha_2 = 0.02$, $\tau = 0$. Discretization range: $[0.01, 0.005, 0.002, 0.001, 0.0005]$.

A.3 Additional results

Additional results mentionned in the text are presented in this section.

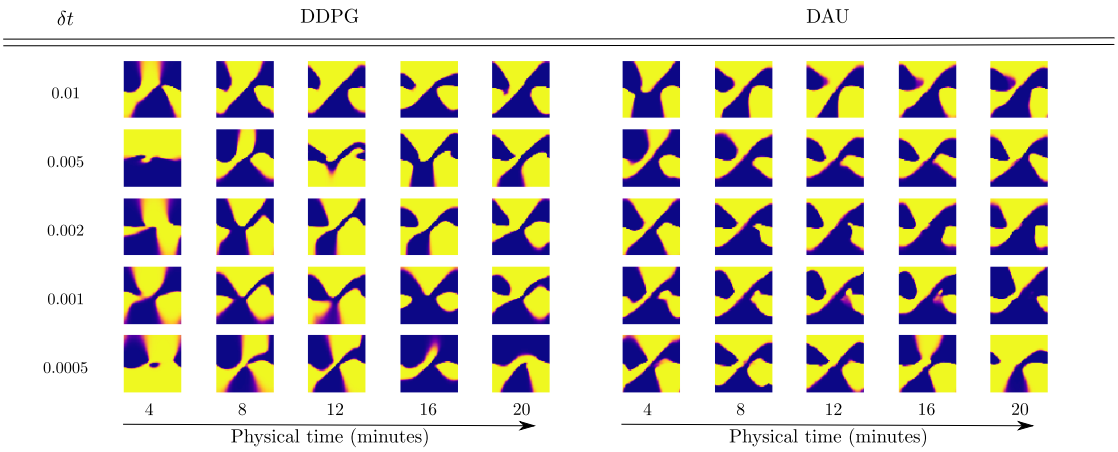


Figure A.1: Policies obtained by DDPG (unscaled version) and AU at different instants in physical time of training on the pendulum swing-up environment. Each image represents the policy learnt by the policy network, with x-axis representing angle, and y-axis angular velocity. The lighter the pixel, the closer to 1 the action, the darker, the closer to -1.

[2] The reward for Bipedal Walker is modified not to scale with δt. This does not introduce any change for the default setup.

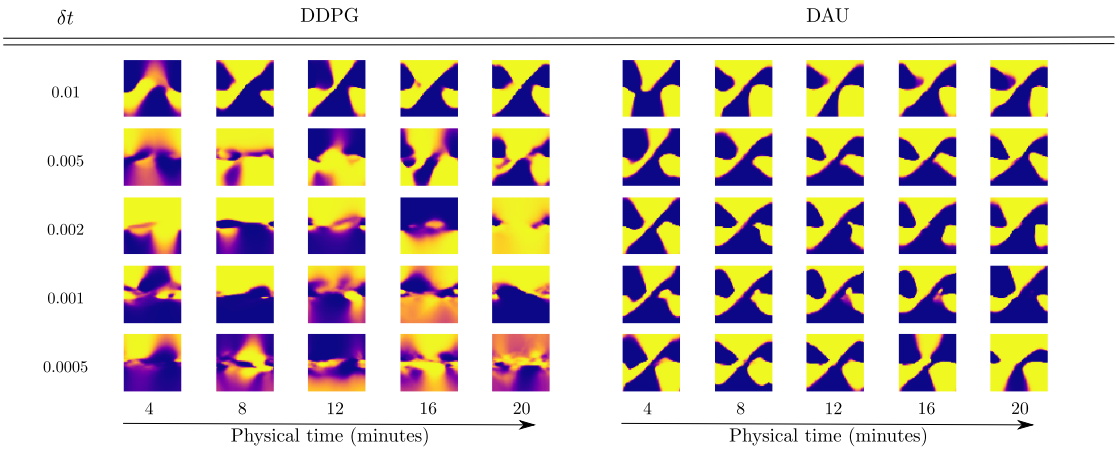


Figure A.2: Policies obtained by DDPG (scaled version) and AU at different instants in physical time of training on the pendulum swing-up environment. Each image represents the policy learnt by the policy network, with x-axis representing angle, and y-axis angular velocity. The lighter the pixel, the closer to 1 the action, the darker, the closer to -1.

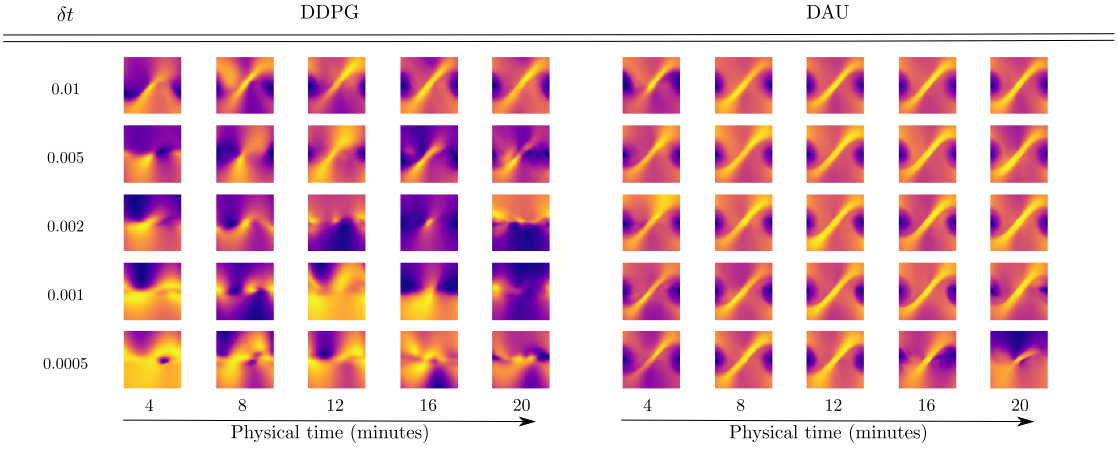


Figure A.3: Value functions obtained by DDPG (scaled version) and AU at different instants in physical time of training on the pendulum swing-up environment. Each image represents the value function learnt, with x-axis representing angle, and y-axis angular velocity. The lighter the pixel, the higher the value.

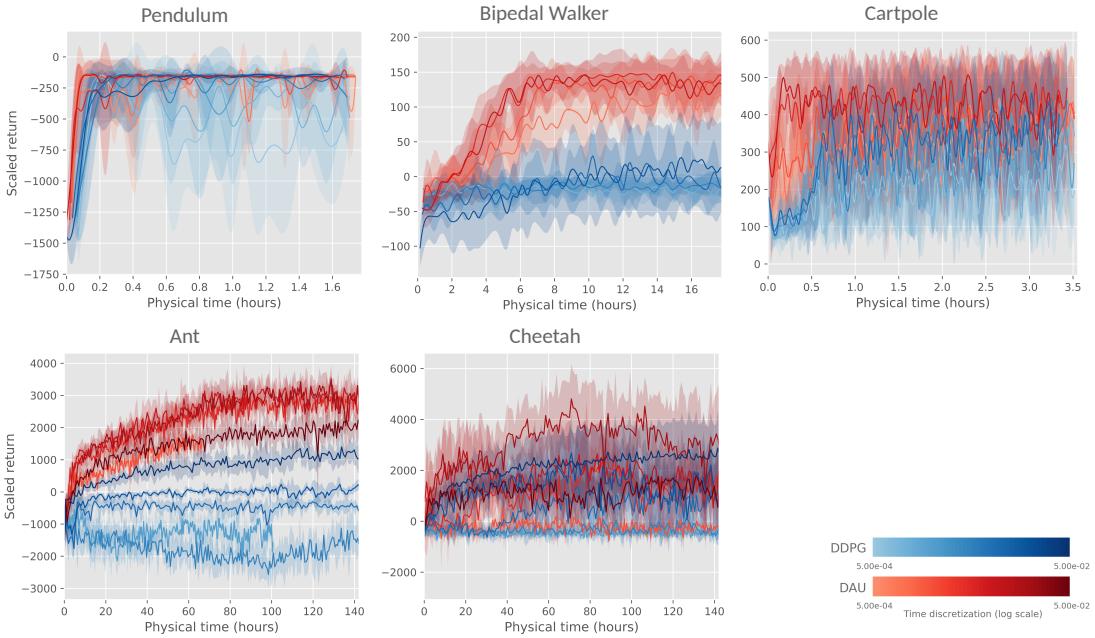


Figure A.4: Learning curves for DAU and DDPG (scaled) on classic control benchmarks for various time discretization δt: Scaled return as a function of the physical time spent in the environment.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Réseaux Récurrents et Apprentissage par Renforcement: Approches Dynamiques

Mots clés : Réseaux de Neurones Récurrents, Apprentissage par Renforcement, Apprentissage Profond

Résumé : D'un agent intelligent plongé dans le monde, nous attendons à la fois qu'il comprenne, et interagisse avec son environement. La compréhension du monde environnant requiert typiquement l'assimilation de séquences de stimulations sensorielles diverses. Interagir avec l'environnement requiert d'ˆetre capable d'adapter son comportement dans le but d'atteindre un objectif fixé, ou de maximiser une notion de récompense. Cette vision bipartite de l'interaction agent-environnement motive les deux parties de cette thèse : les réseaux de neurone récurrents sont des outils puissants pour traiter des signaux multimodaux, comme ceux résultants de l'interaction d'un agent avec son environnement, et l'apprentissage par renforcement et le domaine privilégié pour orienter le comportement d'un agent en direction d'un but. Cette thèse a pour but d'apporter des contributions théoriques et pratiques dans ces deux champs. Dans le domaine des réseaux récurrents, les contributions de cette thèse sont doubles : nous introduisons deux nouveaux algorithmes d'apprentissage de réseaux récurrents en ligne, théoriquement fondés, et passant à l'échelle. Par ailleurs, nous approfondissons les connaissances sur les réseaux récurrents à portes, en analysant leurs propriétés d'invariance. Dans le domaine de l'apprentissage par renforcement, notre contribution principale est de proposer une méthode pour robustifier les algorithmes existant par rapport à la discrétisation temporelle. Toutes ces contributions sont motivées théoriquements, et soutenues par des éléments expérimentaux.

Title : Recurrent Neural Networks and Reinforcement Learning: Dynamic Approaches

Keywords : Recurrent Neural Networks, Reinforcement Learning, Deep Learning

Abstract : An intelligent agent immerged in its environment must be able to both understand and interact with the world. Understanding the environment requires processing sequences of sensorial inputs. Interacting with the environment typically involves issuing actions, and adapting those actions to strive towards a given goal, or to maximize a notion of reward. This view of a two parts agent-environment interaction motivates the two parts of this thesis: recurrent neural networks are powerful tools to make sense of complex and diverse sequences of inputs, such as those resulting from an agent-environment interaction; reinforcement learning is the field of choice to direct the behavior of an agent towards a goal. This thesis aim is to provide theoretical and practical insights in those two domains. In the field of recurrent networks, this thesis contribution is twofold: we introduce two new, theoretically grounded and scalable learning algorithms that can be used online. Besides, we advance understanding of gated recurrent networks, by examining their invariance properties. In the field of reinforcement learning, our main contribution is to provide guidelines to design time discretization robust algorithms. All these contributions are theoretically grounded, and backed up by experimental results.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France