Department of Computer Architecture

This thesis is submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy (PhD)

Deep Learning that Scales: Leveraging Compute and Data

by

Víctor Campos Camúñez

Supervised by
Jordi Torres Viñals & Xavier Giró i Nieto

October 2020

A mi yayo, Pruden,

por enseñarme a dar siempre lo mejor de mí

Abstract

Deep learning has revolutionized the field of artificial intelligence in the past decade. Although the development of these techniques spans over several years, the recent advent of deep learning is explained by an increased availability of data and compute that have unlocked the potential of deep neural networks. They have become ubiquitous in domains such as natural language processing, computer vision, speech processing, and control, where enough training data is available. Recent years have seen continuous progress driven by ever-growing neural networks that benefited from large amounts of data and computing power.

This thesis is motivated by the observation that scale is one of the key factors driving progress in deep learning research, and aims at devising deep learning methods that scale gracefully with the available data and compute. We narrow down this scope into two main research directions. The first of them is concerned with designing hardware-aware methods which can make the most of the computing resources in current high performance computing facilities. We then study bottlenecks preventing existing methods from scaling up as more data becomes available, providing solutions that contribute towards enabling training of more complex models.

This dissertation studies the aforementioned research questions for two different learning paradigms, each with its own algorithmic and computational characteristics. The first part of this thesis studies the paradigm where the model needs to learn from a collection of examples, extracting as much information as possible from the given data. The second part is concerned with training agents that learn by interacting with a simulated environment, which introduces unique challenges such as efficient exploration and simulation.

Acknowledgements

This thesis is the product of a long journey that has shaped me as a researcher, but also as a person, and I would like to thank everyone that has been a part of it.

I am extremely thankful to Xavi Giró for introducing me to the exciting field of deep learning research. His enthusiasm and hard work have brought many students, including myself, opportunities that we had never dreamt of. I would like to thank him for making me aim high, encouraging me to pursue goals I would have never considered and for believing in me since the first day. I would also like to thank Jordi Torres for his unconditional support, and for making sure that I always had the resources I needed to develop my research. He has never hesitated to take the bumpy road with me, for which I will always be grateful. I thank Obra Social "la Caixa" for funding my doctoral studies through La Caixa - Severo Ochoa International Doctoral Fellowship program.

I would like to acknowledge all the colleagues at BSC and UPC, past and present, who have made this journey more fun and easier to bear. I wish to extend my special gratitude to Míriam, with whom I have shared the high and lows of the PhD, and has been a constant source of support and inspiration. I feel lucky to have been desk buddies with her throughout the last four years.

Besides BSC and UPC, I had the opportunity to pursue my research at DFKI, Columbia University, Salesforce Research and DeepMind, and I would like to acknowledge all the friends and colleagues I met there. I am particularly grateful to Brendan Jou for mentoring me during my first steps in research. He taught me to think critically and to conduct rigorous research, and has always been a role model for the kind of researcher I would like to become.

Esta tesis no hubiera sido posible sin Paula. Por su apoyo incondicional, celebrando los éxitos y levantándome el ánimo en los momentos de frustración. Gracias por haber adaptado tu plan de vida a mis constantes viajes, y por animarme a perseguir mis objetivos aunque eso te lo pusiera más difícil a ti. Me siento extremadamente afortunado de estar a tu lado y ojalá vivamos juntos las aventuras que están por venir.

Me gustaría concluir extendiendo mi máximo agradecimiento a mi familia, especialmente a mis padres y abuelos – Mari, Paco, Lola, Cruz, Manolo y Pruden. Por haberme transmitido los valores de la humildad, el esfuerzo y la perseverancia; por haberme apoyado y empujado a perseguir mis sueños, por difíciles y lejanos que parecieran; y porque esta tesis ha sido posible gracias a que habéis dedicado vuestra vida a brindarme las oportunidades de las que vosotros nunca gozasteis. Es tan vuestra como mía.

Contents

Abstract **ii**

Acknowledgements **iii**

Acronyms **ix**

1 Introduction **1**
1.1 Research Questions . . . 3
1.2 Major Contributions . . . 4
1.3 List of Publications . . . 5
1.4 Dissertation Outline . . . 6

2 Deep Learning **8**
2.1 Neural Networks . . . 8
2.1.1 Convolutional Neural Networks . . . 9
2.1.2 Recurrent Neural Networks . . . 10
2.2 Training Neural Networks . . . 12
2.2.1 Stochastic Gradient Descent . . . 12
2.2.2 Transfer Learning . . . 13

3 Reinforcement Learning **14**
3.1 Formal Definition . . . 15
3.2 Value Functions . . . 16
3.2.1 Definitions . . . 16
3.2.2 Optimality . . . 17
3.3 Types of Learning . . . 17
3.4 Common Approaches . . . 18
3.4.1 Value-Based . . . 18
3.4.2 Policy Gradient . . . 19
3.4.3 Direct Policy Search . . . 20

I Learning from Examples **21**

4 Distributed Training of Convolutional Neural Networks **24**

4.1 Related Work 25
4.2 Distributed Training through Data Parallelism 25
4.3 Adjective Noun Pair Detection 27
4.4 Experiments 29
4.4.1 Intra-Node Parallelism 30
4.4.2 Distributed Training 30
4.4.2.1 Throughput Analysis 31
4.4.2.2 Convergence Analysis 32
4.5 Discussion 34

5 Learning to Skip State Updates in Recurrent Neural Networks 35
5.1 Related Work 36
5.2 Model Description 37
5.2.1 Error Gradients 39
5.2.2 Limiting Computation 40
5.3 Experiments 41
5.3.1 Adding Task 41
5.3.2 Frequency Discrimination Task 43
5.3.3 MNIST Classification from a Sequence of Pixels 44
5.3.4 Sentiment Analysis on IMDB 44
5.3.5 Action Classification on UCF-101 47
5.3.6 Temporal Action Localization on Charades 47
5.4 Discussion 50

6 Robust Initialization for WeightNorm & ResNets 51
6.1 Related Work 52
6.2 Weight Normalized ReLU Networks 53
6.2.1 Forward Pass 54
6.2.2 Backward Pass 54
6.2.3 Implementation Details 56
6.3 Residual Networks 56
6.3.1 Forward Pass 57
6.3.2 Backward Pass 57
6.3.3 Implementation Details 58
6.4 Experiments 58
6.4.1 Robustness Analysis 59
6.4.2 Comparison with Batch Normalization 60
6.4.3 Initialization Method and Generalization Gap 62
6.4.4 Preliminary Reinforcement Learning Results 63
6.5 Discussion 65

II Learning from Interaction 66

7 Importance Weighted Evolution Strategies 70
7.1 Evolution Strategies 71
7.1.1 Formulation 71
7.1.2 Scalability Analysis 72

7.2 Importance Weighted Evolution Strategies 72
7.2.1 Formulation 73
7.2.2 Scalability Analysis 74
7.3 Experiments 75
7.3.1 Effect of the Number of IW Updates 75
7.3.2 Effect of the Model Size 76
7.3.3 Effect of the Learning Rate 77
7.4 Related Work 78
7.5 Discussion 79

8 Unsupervised Discovery of State-Covering Skills 80
8.1 Information-Theoretic Skill Discovery 81
8.1.1 Reverse Form of the Mutual Information 83
8.1.2 Forward Form of the Mutual Information 84
8.2 Limitations of Existing Methods 85
8.2.1 Assumptions 85
8.2.2 Reverse Form of the Mutual Information 85
8.2.3 Forward Form of the Mutual Information 87
8.2.4 Summary of Findings 88
8.3 Proposed Method 89
8.4 Experiments 91
8.5 Related Work 97
8.6 Discussion 99

9 Conclusion 101

A Qualitative Results for Skip RNN 104
A.1 Adding Task 105
A.2 Frequency Discrimination Task 106

B Choice of Mutual Information's Form for EDL 107

C Implementation Details 109
C.1 Robust Initialization for WeightNorm & ResNets 109
C.1.1 Synthetic Data 109
C.1.2 Residual Network Architecture 110
C.1.3 MNIST 111
C.1.4 CIFAR 111
C.2 Unsupervised Discovery of State-Covering Skills 113
C.2.1 Environments 113
C.2.2 RL Agents 113
C.2.3 Exploration 114
C.2.4 Skill Discovery 115

D Proofs 117

Bibliography **125**

Acronyms

A3C Asynchronous Advantage Actor-Critic

ANP Adjective Noun Pair

BN Batch Normalization

CNN Convolutional Neural Network

CPU Central Processing Unit

EDL Explore, Discover and Learn

ES Evolution Strategies

FC Fully Connected

FLOP Floating Point Operation

GPU Graphics Processing Unit

GRU Gated Recurrent Unit

IW-ES Importance Weighted Evolution Strategies

KL Kullback–Leibler

LSTM Long Short-Term Memory

MDP Markov Decision Process

MI Mutual Information

MLP Multilayer Perceptron

MSE Mean Squared Error

MVSO Multilingual Visual Sentiment Ontology

PPO Proximal Policy Optimization

RAM Random Access Memory

ReLU Rectified Linear Unit

ResNet Residual Network

RL Reinforcement Learning

RNN Recurrent Neural Network

SAC Soft Actor-Critic

SGD Stochastic Gradient Descent

SMM State Marginal Matching

UVFA Universal Value Function Approximator

VAE Variational Autoencoder

WN Weight Normalization

WRN Wide Residual Network

1 Introduction

Can machines think? Alan Turing posed this question in his famous 1950 paper, *Computing Machinery and Intelligence* [1], where he proposed the *Imitation Game* as a general method to test machine intelligence. In this test, a human evaluator would engage in a conversation through a text-based channel with two players – a machine and a human. The difficult question of whether machines can think was then reformulated into evaluating whether a machine's behavior is indistinguishable from that of a human in the imitation game. Although Turing's work was clearly concerned with machine intelligence, the term *artificial intelligence* that is widely used nowadays was not coined until six years later, when John McCarthy organized the Dartmouth Summer Research Project on Artificial Intelligence. This workshop served as a catalyst for decades of artificial intelligence research, field that has since then interleaved episodes of important investment and interest with winters produced by an excess of optimism in estimating the pace at which progress could be made.

The proposal for the Dartmouth workshop was based on the conjecture that "every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it" [2]. Inspired by this vision, an important part of the almost 70 years of artificial intelligence research has been devoted to developing expert systems that are able to mirror human reasoning. Expert systems excel at tasks that we fully understand, those for which we can formalize the underlying decision process of a solution. This enabled successful applications of rule-based systems in domains such as chemical analysis and medical diagnostics in the 1960s and 1970s. Our own understanding of the world turns out to be the main limitation of these approaches. Indeed, this was well-known from the early days of computer programs, as can be found

in the notes written by Ada Lovelace on Babbage's Analytical Engine in 1842: "The Analytical Engine has no pretensions to originate anything. It can do whatever we know how to order it to perform." [3].

The aspiration of artificial intelligence is grander than just automating solutions for problems that we fully understand, but this will require providing artificial intelligence systems with information that is beyond our knowledge. This observation has given rise to a research trend concerned with designing machines that can learn from experience. Instead of requiring a complete set of expert-designed rules, learning is possible as long as a feedback signal can be provided. This makes it a much more flexible paradigm for addressing challenging tasks. It offers a less engineering-focused view of the problem, which aims at designing general purpose methods whose quality is not limited by our own understanding of the world. The responsibility for the decision-making is no longer on us, and it is turned over to the artificial intelligence system itself.

Machine learning is a discipline within artificial intelligence that is concerned with designing systems that can learn from data. Within machine learning, the field of *deep learning* has attracted much research interest in the last decade. It studies general purpose systems, usually known as artificial neural networks, that can autonomously build a hierarchy of representations from data. These methods have contributed towards important breakthroughs in domains such as computer vision, natural language processing, speech recognition, and control. An important reason explaining the success of deep learning systems in all these seemingly unrelated fields is their ability to leverage computation. Thanks to the continuous improvements in hardware, general purpose methods that can leverage computation are ultimately the most effective – in a sense, by trading off compute for knowledge. This observation was recently formalized by Richard Sutton as the *bitter lesson*: "One thing that should be learned from the bitter lesson is the great power of general purpose methods, of methods that continue to scale with increased computation even as the available computation becomes very great." [4].

Data, compute and large neural networks are the three key components explaining the recent success of deep learning methods. The recent history of deep learning research hints at an ever-growing availability of high quality data, either in the form of datasets or simulated environments. The generalized version of Moore's law suggests a similar trend for computational power, with the cost of each unit of computation decreasing exponentially over time. In order to understand the importance of scale and computation in deep learning, let us take a closer look at the task of object recognition from images. The ImageNet dataset [5], containing over 1M images belonging to $1,000$ different object categories, used to be one of the grand challenges in artificial intelligence research. In 2012, Krizhevsky et al. [6] showed the potential of deep learning methods

for such a challenging task, outperforming competing solutions by a large margin. The quality of deep learning-based solutions has quickly improved since then, with current methods already surpassing human performance on this task [7]. The neural network by Krizhevsky et al. [6] had only eight layers and still took about two weeks to train on a machine with two GPUs, while it is now possible to use hundreds of accelerators to train more accurate models with dozens of layers in a matter of minutes [8]. Such is the pace of progress that even ImageNet has become too small for training some of the largest models currently considered by researchers, who are already experimenting with datasets containing billions of examples [9]. Similar trends have been observed in fields like natural language processing [10, 11] and reinforcement learning [12, 13].

1.1 Research Questions

Large scale compute and data were necessary conditions to unlock the potential of deep learning, but they alone do not explain the recent advances in the field. In parallel, deep learning research has kept the pace in developing methods that are able to leverage the increased amount of compute and data. In this context, the research conducted in this dissertation can be understood as laying stepping stones towards answering the following question: **How can we design deep learning methods that are able to leverage unlimited compute and data?** The scope of such a research question is broad, and we narrow our focus into two more specific areas of research.

The first direction we consider is that of **designing deep learning algorithms that can make the most of the available hardware**. Given that the increase in CPU clock speed has stagnated in recent years, specialized accelerators and distributed systems have become the de facto strategies for building machines with increased compute capabilities. This motivates the design of hardware-aware deep learning architectures and distributed training methods that can make the most of the hardware that is available to the system.

The second direction is concerned with **devising deep learning methods that can scale up with the amount of available data**. When richer datasets or environments become available, the capacity and learning capabilities of our agents should be increased accordingly to make the most of the data they are exposed to. However, there are multiple bottlenecks preventing us from scaling up existing methods naively. This opens up multiple research directions, ranging from solving low-level optimization issues to designing objectives that result in efficient learning.

We can distinguish two different computational paradigms depending on the nature of the data used to train a deep learning model. The data can be given beforehand, in the form

of a static dataset, which is the standard setting in most *supervised* and *unsupervised learning* problems. Since the data can be read from disk, the computational burden falls on the accelerator where the model is deployed. On the other hand, agents can learn from interaction with an environment, a paradigm that is often formulated through the lens of *reinforcement learning*. This setup puts much more pressure on the devices that simulate the environment, often CPUs, and its computational requirements can differ substantially from those of settings where the model is trained using a fixed dataset. This dissertation considers the aforementioned research questions in both settings. Part I is devoted to learning from fixed collections of examples, whereas Part II considers the setting where an agent learns from interaction.

1.2 Major Contributions

The technical contributions of this dissertation can be listed under each of the learning paradigms presented earlier:

Part I: Learning from Examples

[C1] We explore how to accelerate training of Convolutional Neural Networks on a distributed GPU cluster.

[C2] We introduce Skip RNN, a Recurrent Neural Network architecture that learns to skip state updates and can be trained for different computational budgets.

[C3] We propose a novel initialization strategy for weight normalized and residual networks, which improves robustness and enables training deeper networks.

Part II: Learning from Interaction

[C4] We introduce Importance Weighted Evolution Strategies, which improves the data efficiency of Evolution Strategies while preserving its scalability.

[C5] We provide theoretical analysis and empirical evidence showing that existing unsupervised skill discovery methods based on information-theoretic objectives fail at learning state-covering skills. We propose EDL, a method for optimizing the same information-theoretic objective while overcoming the limitations of previous methods.

1.3 List of Publications

All the technical contributions presented in this dissertation have been published at peer-reviewed venues.

Part I: Learning from Examples

[C1] Víctor Campos, Francesc Sastre, Maurici Yagües, Míriam Bellver, Xavier Giró-i-Nieto, and Jordi Torres. Distributed training strategies for a computer vision deep learning algorithm on a distributed GPU cluster. *Procedia Computer Science*, 2017

[C2] Víctor Campos, Brendan Jou, Xavier Giró-i-Nieto, Jordi Torres, and Shih-Fu Chang. Skip RNN: Learning to skip state updates in recurrent neural networks. In *ICLR*, 2018

[C3] Devansh Arpit*, Víctor Campos*, and Yoshua Bengio. How to initialize your network? Robust initialization for WeightNorm & ResNets. In *NeurIPS*, 2019

Part II: Learning from Interaction

[C4] Víctor Campos, Xavier Giró-i-Nieto, and Jordi Torres. Importance Weighted Evolution Strategies. In *NeurIPS Deep RL Workshop*, 2018

[C5] Víctor Campos, Alexander Trott, Caiming Xiong, Richard Socher, Xavier Giró-i-Nieto, and Jordi Torres. Explore, Discover and Learn: Unsupervised discovery of state-covering skills. In *ICML*, 2020

As a product of other research activities

[Extension of V. Campos' BSc Thesis] Víctor Campos, Brendan Jou, and Xavier Giró-i-Nieto. From pixels to sentiment: Fine-tuning CNNs for visual sentiment prediction. *Image and Vision Computing*, 2017

[X. Lin's BSc Thesis] Xunyu Lin, Víctor Campos, Xavier Giró-i-Nieto, Jordi Torres, and Cristian Canton Ferrer. Disentangling motion, foreground and background features in videos. In *CVPR Brave New Motion Representations Workshop*, 2017

[Initial results for C1] Víctor Campos, Francesc Sastre, Maurici Yagües, Jordi Torres, and Xavier Giró-i-Nieto. Scaling a convolutional neural network for classification of adjective noun pairs with tensorflow on gpu clusters. In *CCGRID*, 2017

*Equal contribution

[D. Fernández's MSc Thesis] Dèlia Fernández, Alejandro Woodward, Víctor Campos, Xavier Giró-i-Nieto, Brendan Jou, and Shih-Fu Chang. More cat than cute?: Interpretable prediction of adjective-noun pairs. In *ACM MM MUSA Workshop*, 2017

[D. Fojo's BSc Thesis] Daniel Fojo, Víctor Campos, and Xavier Giró-i-Nieto. Comparing fixed and adaptive computation time for recurrent neural networks. In *ICLR Workshop Track*, 2018

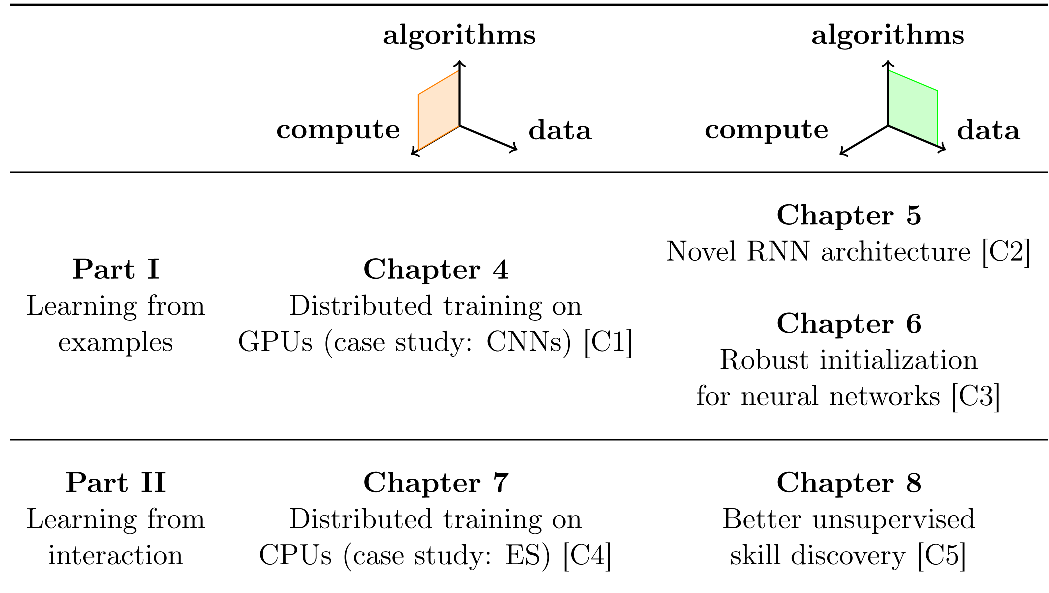
[Collaboration with GPI (UPC)] Amaia Salvador, Míriam Bellver, Víctor Campos, Manel Baradad, Ferran Marqués, Jordi Torres, and Xavier Giró-i-Nieto. Recurrent neural networks for semantic instance segmentation. In *CVPR DeepVision Workshop*, 2018

[Follow-up of V. Campos' BSc Thesis] Víctor Campos, Xavier Giró-i-Nieto, Brendan Jou, Jordi Torres, and Shih-Fu Chang. Sentiment concept embedding for visual affect recognition. In *Multimodal Behavior Analysis in the Wild*. Elsevier, 2019

1.4 Dissertation Outline

Table 1.1 illustrates the structure of this dissertation and summarizes the main contribution made in each chapter. Each part is devoted to a different learning paradigm. Within each part, we classify our contributions depending on the the axes of scale being studied. In particular, we separately study how to develop algorithms that scale gracefully as more compute and data become available.

Part I explores the paradigm of learning from examples, where the model needs to extract knowledge from a fixed set of samples. Chapter 4 studies strategies for distributed training of Convolutional Neural Networks (CNNs) on a GPU cluster, shortening training times thanks to increased compute. Given that empirical evidence plays an important role in deep learning research, fast iteration is important for both academic and industrial applications. Chapter 5 introduces a novel Recurrent Neural Network (RNN) architecture, Skip RNN, that learns to solve tasks while using only a fraction of the elements in the input sequence. This provides important computational savings at inference, i.e. when making predictions on new inputs, which are obtained by learning skipping patterns from data. Chapter 6 presents a robust initialization for neural networks that stabilizes training and provides improved generalization. This initialization scheme allows training much deeper models reliably, which is key to leveraging larger data collections.



	algorithms, compute, data	algorithms, compute, data
Part I Learning from examples	**Chapter 4** Distributed training on GPUs (case study: CNNs) [C1]	**Chapter 5** Novel RNN architecture [C2] **Chapter 6** Robust initialization for neural networks [C3]
Part II Learning from interaction	**Chapter 7** Distributed training on CPUs (case study: ES) [C4]	**Chapter 8** Better unsupervised skill discovery [C5]

Table 1.1: Thesis structure. Each part is dedicated to a different learning paradigm. For each learning paradigm, we separately study how to develop algorithms that scale gracefully as more compute and data become available.

Part II considers agents that need to learn by interacting with a simulated environment. Simulations are often performed on CPU, and evolutionary methods have been shown to scale gracefully when using hundreds of cores in a distributed setting. This usually comes at the cost of reduced data efficiency, i.e. the agent needs more interactions with the environment to solve the task, which can be troublesome when simulation is expensive. Chapter 7 extends a state of the art evolutionary approach in order to improve its data efficiency. This is achieved with a minimal impact in its scalability, and we report gains both in terms of data efficiency and wall-clock time. When enough compute is available, what agents can learn in complex environments is often limited by our own ability to define objectives and tasks. Chapter 8 studies methods that let agents set their own goals, self-supervising their learning and overcoming the limitations of handcrafted objectives. Through theoretical and empirical evidence, we show that existing methods do not let agents explore all the possibilities that are available to them in the environment. We then propose Explore, Discover and Learn (EDL), an alternative approach that overcomes these limitations and offers several advantages over existing methods.

2

Deep Learning

Computers can solve problems by running algorithms, i.e. a list of instructions that need to be carried out in order to transform inputs into the desired outputs. Numerous problems can be solved with human designed algorithms, such as sorting, encrypting messages or compressing signals. However, how can computers solve problems for which we do not know the right algorithm? Note that this set of problems includes some that are simple for people to perform, such as those related to cognition and perception, but are hard to describe formally. Assuming that we are able to compile a collection of inputs and their corresponding outputs, *machine learning* aims at designing machines that are capable of extracting (*learning*) the algorithm from the provided examples.

There exist numerous machine learning algorithms, and we refer the reader to Alpaydin [26] for an overview. The best solutions to many difficult machine learning problems are based on deep neural networks, which allow computers to learn from experience by building a hierarchy of concepts. This results on a deep computational graph with many layers, so this sub-field of machine learning is commonly referred to as *deep learning*. This chapter covers the basics of deep learning, and we refer the reader to Goodfellow et al. [27] for a deeper introduction to the field.

2.1 Neural Networks

Neural networks can be seen as flexible function approximators whose parameters are fitted in a data-driven fashion. They are built by stacking a series of simple non-linear mappings, usually referred to as *layers*. Note that a composition of linear mappings can

be reduced to a single linear mapping, so it is crucial to build neural networks off of non-linear layers. In its most generic form, a feedforward neural network with L layers can be defined recursively as follows:

$$\mathbf{h}^l = \phi\left(\mathbf{W}^l \mathbf{h}^{l-1} + \mathbf{b}^l\right) \tag{2.1}$$

where $\mathbf{h}^l \in \mathbb{R}^{n_l}$ is the output vector of the l-th layer, $\mathbf{h}^0 = \mathbf{x} \in \mathbb{R}^{n_x}$ is the input vector, ϕ is a non-linear function, and $\mathbf{W}^l \in \mathbb{R}^{n_l \times n_{l-1}}$ and $\mathbf{b}^l \in \mathbb{R}^{n_l}$ are trainable weights and biases, respectively. We will refer to each of the scalar elements in the output vector of a layer as *units* or *neurons*.

The capacity of the network to model complex mappings can be increased by adding intermediate layers between inputs and outputs, also known as *hidden layers*. The intuition behind stacking multiple layers is that it lets networks build a hierarchy of feature extractors. There exist strong theoretical results showing that a neural network with a single hidden layer can approximate any function [28]. Unfortunately, these results do not tell us how to design our neural networks to achieve such property, which might require too many neurons to be implemented in practice. For this reason, it is common to increase the capacity of neural networks through depth by adding more hidden layers [29].

The concept of *inductive bias* is crucial for understanding recent advances in deep learning. It refers to ways of incorporating prior knowledge about the task in the architecture of neural networks, so that they can learn from fewer examples and generalize more easily to unseen inputs. Similar cognitive biases have been observed in children, which let them eliminate broad swaths of the hypothesis space when learning new words [30]. In the context of neural networks, these priors shape the properties of the mappings that a model can learn, and bias them towards those that practitioners deem useful for the task at hand. Leveraging the right inductive biases has been key for scaling up neural networks and making them perform well on complex tasks involving high-dimensional inputs such as images or speech. The following subsections describe some of the most common neural network types and the inductive biases they implement.

2.1.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) implement a weight-tying scheme that induces translation equivariance[1]. Tying weights across input locations results in an operation that is akin to convolving the learned kernels and the input.

[1]The translation equivariance property implies that any shift in the input will result in the same shift in the output. This should not to be mistaken for translation invariance, which implies that any shifted version of the input will produce the same output.

CNNs are widely used for computer vision applications, where they leverage the fact that the appearance of objects is independent of their location. Moreover, since objects have a local spatial support, they usually feature small convolutional kernels that only take into consideration the neighborhood of each location. Stacking several layers increases the actual receptive field of each filter, making it possible for the output of the network to be a function of the whole input image if needed. These assumptions result in a massive reduction of the number of trainable parameters at each layer, which enables training very deep CNNs without catastrophically overfitting to the training set [31]. The use of CNNs is not limited to computer vision, as similar architectures have been used to process other modalities such as text [32] and speech [33].

2.1.2 Recurrent Neural Networks

The basic layer model described in Equation 2.1 assumes a fixed input size. Fully convolutional architectures are able to process inputs of arbitrary size, but the size of their output will change accordingly [34]. This raises the question of how to design neural network architectures that can process variable-length inputs and produce an output of constant size.

Let us consider a sequence of input vectors, $\mathbf{x} = (\mathbf{x}_1, ..., \mathbf{x}_T)$. When applied to each input vector separately, the feedforward model described in Equation 2.1 would output a sequence of activation vectors at each layer, $\mathbf{h}^l = \left(\mathbf{h}^l_1, ..., \mathbf{h}^l_T\right)$, which are agnostic to previous activations. Recurrent Neural Networks (RNNs) extend the feedforward model by adding a recurrent connection in time:

$$\mathbf{h}^l_t = \phi\left(\mathbf{W}^l \mathbf{h}^{l-1}_t + \mathbf{U}^l \mathbf{h}^l_{t-1} + \mathbf{b}^l\right) \tag{2.2}$$

where $\mathbf{h}^0 = \mathbf{x}$ denotes the input sequence, similarly to Equation 2.1. $\mathbf{U}^l \in \mathbb{R}^{n_l \times n_l}$ operates on the hidden state in the previous time step, $\mathbf{h}^l_{t-1}$, allowing information to persist. Providing models with memory and enabling them to model the temporal evolution of signals is a key factor in many sequence classification and transduction tasks where RNNs excel, such as machine translation [35], language modeling [36] or speech recognition [37].

Given the dependency on the hidden state of the previous time step, many of the computations in an RNN need to be performed sequentially. This results in a computation time that grows linearly with the input length. Chapter 5 presents a novel neural network architecture that is able to skip some of those computations, providing important savings when deployed on modern accelerators like GPUs.

As will be described in Section 2.2, neural networks are often trained using gradient-based methods. In such a setup, the multiplicative memory mechanism in Equation 2.2 might result in vanishing or exploding gradients that preclude training, motivating the design of alternative recurrent architectures.

Long Short-Term Memory (LSTM) [38]. The LSTM cell overcomes vanishing and exploding gradient issues by interacting with the memory vector in an additive fashion:

$$\mathbf{i}_t^l = \sigma\left(\mathbf{W}_i^l \mathbf{h}_t^{l-1} + \mathbf{U}_i^l \mathbf{h}_{t-1}^l + \mathbf{b}_i^l\right) \tag{2.3}$$

$$\mathbf{f}_t^l = \sigma\left(\mathbf{W}_f^l \mathbf{h}_t^{l-1} + \mathbf{U}_f^l \mathbf{h}_{t-1}^l + \mathbf{b}_f^l\right) \tag{2.4}$$

$$\mathbf{o}_t^l = \sigma\left(\mathbf{W}_o^l \mathbf{h}_t^{l-1} + \mathbf{U}_o^l \mathbf{h}_{t-1}^l + \mathbf{b}_o^l\right) \tag{2.5}$$

$$\hat{\mathbf{c}}_t^l = \tanh\left(\mathbf{W}_c^l \mathbf{h}_t^{l-1} + \mathbf{U}_c^l \mathbf{h}_{t-1}^l + \mathbf{b}_c^l\right) \tag{2.6}$$

$$\mathbf{c}_t^l = \mathbf{f}_t^l \mathbf{c}_{t-1}^l + \mathbf{i}_t^l \hat{\mathbf{c}}_t^l \tag{2.7}$$

$$\mathbf{h}_t^l = \mathbf{o}_t^l \tanh(\mathbf{c}_t^l) \tag{2.8}$$

where σ is the sigmoid function, $\mathbf{i}_t^l \in \mathbb{R}^{n_l}$ is the input gate, $\mathbf{f}_t^l \in \mathbb{R}^{n_l}$ is the forget gate, $\mathbf{o}_t^l \in \mathbb{R}^{n_l}$ is the output gate, $\mathbf{c}_t^l \in \mathbb{R}^{n_l}$ is the cell state, and $\mathbf{h}_t^l \in \mathbb{R}^{n_l}$ is the hidden state that is exposed to subsequent blocks in the computational graph. The different gates control which elements are stored, forgotten and exposed to the output depending on the current input and previous hidden state.

Gated Recurrent Unit (GRU) [39]. The GRU can be seen as a simplified version of the LSTM cell that does not keep separate cell and hidden states, and merges the forget and input gate. This results in a smaller number of trainable parameters, which generally exhibits similar performance to that of LSTMs:

$$\mathbf{z}_t^l = \sigma\left(\mathbf{W}_z^l \mathbf{h}_t^{l-1} + \mathbf{U}_z \mathbf{h}_{t-1}^l + \mathbf{b}_z^l\right) \tag{2.9}$$

$$\mathbf{r}_t^l = \sigma\left(\mathbf{W}_r^l \mathbf{h}_t^{l-1} + \mathbf{U}_r \mathbf{h}_{t-1}^l + \mathbf{b}_r^l\right) \tag{2.10}$$

$$\hat{\mathbf{h}}_t^l = \tanh\left(\mathbf{W}_h^l \mathbf{h}_t^{l-1} + \mathbf{r}_t^l \mathbf{U}_h \mathbf{h}_{t-1}^l + \mathbf{b}_h^l\right) \tag{2.11}$$

$$\mathbf{h}_t^l = (1 - \mathbf{z}_t^l)\mathbf{h}_{t-1}^l + \mathbf{z}_t^l \hat{\mathbf{h}}_t^l \tag{2.12}$$

where $\mathbf{z}_t^l \in \mathbb{R}^{n_h}$ plays a similar role to the input and forget gates in the LSTM cell, and $\mathbf{r}_t^l \in \mathbb{R}^{n_h}$ is the reset gate that allows erasing information in the hidden state.

2.2 Training Neural Networks

This section discusses how to optimize the parameters of a neural network to perform some given task. Let us assume that we have access to a dataset of input and desired output tuples $\mathcal{D} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{N}$, and we would like to train our network f_θ parameterized by θ. Following notation in Equation 2.1, $\theta = \{(\mathbf{W}^l, \mathbf{b}^l)\}_{l=1}^{L}$. We will aim at minimizing some loss or cost function $\mathcal{L}$ between the outputs of our model and the ground truth values. Common choices for this loss function are the Mean Squared Error for regression tasks or the cross-entropy loss for classification problems. We can then frame the optimization task as

$$\theta^* = \operatorname*{argmin}_{\theta} \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}} \left[\mathcal{L}\left(f_\theta(\mathbf{x}), \mathbf{y} \right) \right] \tag{2.13}$$

2.2.1 Stochastic Gradient Descent

The gradient of a function points towards its direction of maximum increase. When the loss function is differentiable with respect to the parameters of the model, we can follow the opposite direction of the gradient in order to update the parameters and find a point with smaller loss value:

$$\theta_{i+1} \leftarrow \theta_i - \alpha \nabla_{\theta_i} \mathcal{L}\left(f_{\theta_i}(\mathbf{x}), \mathbf{y} \right), \; (\mathbf{x}, \mathbf{y}) \sim \mathcal{D} \tag{2.14}$$

where the learning rate α determines the magnitude of the step taken. This method is known as Stochastic Gradient Descent (SGD), as it replaces the actual gradient that would be computed over the whole dataset with a single sample estimate. In practice, a batch of samples is commonly used in order to reduce the variance of the gradient estimate, which incurs into a negligible computational overhead when leveraging the parallelization capabilities of modern accelerators such as GPUs. Chapter 4 studies methods to accelerate training of neural networks through distributed computation.

In multi-layered networks, the gradients can be propagated through all layers by applying the chain rule. This process is known as the backpropagation algorithm [40]. The optimization task can become difficult due to the high dimensionality of the parameter space and the noisy and potentially non-stationary gradients. Numerous techniques have been proposed to improve the convergence properties of the original update rule, which range from including momentum in the updates [41] to re-scaling gradients based on adaptive estimation of their moments [42, 43].

The computation made when training neural networks with SGD can be decomposed in two main steps: forward and backward passes through the net. The forward pass computes the outputs for a batch of data, and an error with respect to the desired result is calculated. In the backward pass, such error is backpropagated through the neural network in order to compute gradients with respect to every parameter. These gradients are then used to update the weights of the model following Equation 2.14. These steps are repeated until a termination condition is met, e.g. when the loss function plateaus or after a fixed number of updates.

SGD is an iterative algorithm for updating the current set of parameters, but it does not define how to set the *initial* value of such parameters. Parameter initialization turns out to be crucial for proper gradient-based learning, as poor initialization schemes can lead to vanishing or exploding gradients problems. Chapter 6 studies parameter initialization schemes that are well-suited for gradient-based optimization of neural networks.

2.2.2 Transfer Learning

Humans are able to leverage prior knowledge, experience, and skills when faced with new tasks or situations. On the other hand, most of our machine learning systems start from scratch every time they are tasked with solving a new problem. This generally results in inefficient learning, forcing practitioners to collect a huge number of annotated examples for the task at hand.

A particularly successful strategy for reusing knowledge in neural networks consists in initializing some of their parameters using those from a pre-trained model. The intuition behind this approach is that deep neural networks build a hierarchy of feature extractors, and most of the levels in the hierarchy might be similar for related tasks. For instance, one can initialize the backbone of an object localization network with the parameters of an object recognition model to boost performance when the number of annotated examples is scarce [44]. Similarly, the features extracted by bidirectional language models trained on a huge corpora have been found to transfer to a plethora of natural language processing tasks [10]. Depending on the size of the available training set, one can choose to freeze the pre-trained parameters or fine-tune them. We will follow this practice in several chapters of this thesis in order to accelerate training of our models.

3
Reinforcement Learning

An important part of our knowledge of the world is acquired through interaction, without an external teacher telling us what the outcomes of every single action we take will be. This contrasts with some of the assumptions made in Chapter 2, where we considered learning from a collection of training examples for which an expert has provided the right outputs – a paradigm known as *supervised learning*. *Reinforcement Learning* (RL) is a computational approach to goal-directed learning from interaction that does not rely on expert supervision. Note that this setting also differs from *unsupervised learning*, which is generally about finding patterns and hidden structure in unlabelled data.

The RL problem can be described by the loop depicted in Figure 3.1, where at each time step the agent observes the state of the environment and acts accordingly. The consequences of such action are evaluated and returned to the agent in the form of a scalar reward and the updated state. The goal of the agent is to discover behaviors that maximize the rewards obtained from the environment. Defining proper reward functions is crucial in order to obtain the desired behaviors, but this task can become very challenging in complex environments. Chapter 8 presents methods that enable agents to set their own reward functions and acquire knowledge autonomously.

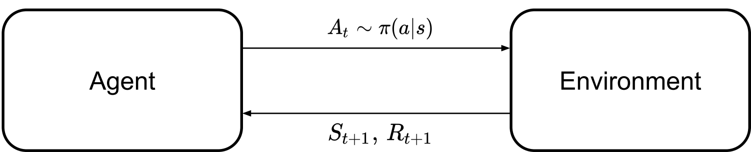


Figure 3.1: Illustration of the RL setting. An agent interacts with an environment, trying to produce actions that lead to high rewards.

This chapter provides a high-level overview of basic concepts in RL. All approaches used in this thesis are model-free, meaning that they do not rely on a model of the environment in order to maximize reward. For this reason, the remaining of this chapter is devoted to explaining the basic concepts of model-free methods. We note that there exist model-based methods that plan using a model of the environment, which can be either given or learned from experience. We refer the reader to Sutton and Barto [45] for a deeper introduction to the field.

3.1 Formal Definition

The RL problem is formalized as a Markov Decision Process (MDP), $\mathcal{M} \equiv (\mathcal{S}, \mathcal{A}, p, \mathcal{R})$, where $\mathcal{S}$ is the state space and $\mathcal{A}$ is the action space; $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, \infty)$ represents the probability density of the next state given the current state and action; and $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ represents a scalar reward function. We will consider an episodic setting where the agent interacts with the environment in discrete time steps, generating trajectories of states, actions, and rewards:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, \dots \tag{3.1}$$

where we use R_{t+1} to denote the reward obtained by performing action A_t in state S_t. An initial state $S_0 \sim p(s_0)$ is sampled at the beginning of each episode. After the agent performs an action A_t, the environment transitions to a new state $S_{t+1} \sim p(S_{t+1}|S_t, A_t)$. It is important to note that all states in an MDP must fulfill the Markov property:

$$p(S_{t+1}|S_t, A_t) = p(S_{t+1}|S_t, A_t, \dots, S_0, A_0) \tag{3.2}$$

which implies that the future is conditionally independent of the past, and it depends only on the current state and action. This should not be seen as a limitation of the framework but as a condition on how states for each task should be defined, i.e. the state must encode all information about the past that is useful for the future.

The outcome of solving the RL problem is a policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ that maps states to actions. Such mapping can be deterministic or stochastic[1]. The goal is finding the optimal policy that maximizes the expected sum of future discounted returns when deployed in the

[1] Stochastic policies might be optimal in some situations, e.g. when facing an opponent that could otherwise exploit the determinism of the policy. Deterministic policies are often preferred when some actions might lead to catastrophic outcomes, e.g. in robotics.

environment:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \, \mathbb{E}_{s_{t+1} \sim p(s_{t+1}|s_t, a_t), a_t \sim \pi(a|s), s_0 \sim p(s_0)} \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \right] \tag{3.3}$$

where $\gamma \in [0, 1]$ is the discount factor that controls how strongly future rewards are taken into account. There are different interpretations for the meaning of the discount factor, and why an agent should prioritize immediate rewards over delayed ones. Here we highlight the most pragmatic view, and note that the infinite sum in Equation 3.3 diverges when $\gamma = 1$ (i.e. in the undiscounted case).

3.2 Value Functions

Value functions are useful measures that quantify the goodness of states and actions, and most state of the art RL methods rely on value function estimates. This section covers the definition of these measures and their relationship to the optimal policy.

3.2.1 Definitions

Before diving into the definition of different value functions, let us define the return as the discounted sum of future rewards:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.4}$$

We can define different types of value functions by building off of the definition of return. For brevity, we will use $\mathbb{E}_\pi$ to denote expectation over trajectories produced when following policy π. Note that this operator integrates over action distributions induced by π as well as over the stochastic transition dynamics of the environment.

State-value function. It measures the expected return one would obtain by following policy π from state s onward:

$$V_\pi(s) = \mathbb{E}_\pi \left[G_t | S_t = s \right] \tag{3.5}$$

Action-value function. It measures the expected return one would obtain by taking action a at state s and following policy π afterwards:

$$Q_\pi(s, a) = \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \tag{3.6}$$

These two value functions are tightly related, as the state-value function is recovered when computing the expectation over Q values under policy π:

$$V_\pi(s) = \sum_{a \in \mathcal{A}} Q_\pi(s,a)\pi(a|s) \tag{3.7}$$

Advantage function. When all rewards have the same sign, it might be difficult to assess the quality of an action by estimating its Q value. The advantage function measures the goodness of an action with respect to the expected quality over all valid actions:

$$A_\pi(s,a) = Q_\pi(s,a) - V(s) \tag{3.8}$$

Intuitively, the sign of $A(s,a)$ tells us whether an action is better or worse than average for each state.

3.2.2 Optimality

We can define optimal value functions over the set of all valid policies:

$$V^*(s) = \max_\pi V_\pi(s) \tag{3.9}$$

$$Q^*(s,a) = \max_\pi Q_\pi(s,a) \tag{3.10}$$

These are useful definitions, as they let us define the optimal policy introduced in Equation 3.3 in terms of value functions:

$$\pi^*(s) = \operatorname*{argmax}_\pi V_\pi(s) = \operatorname*{argmax}_\pi Q_\pi(s,a) \tag{3.11}$$

where $V_{\pi^*}(s) = V^*(s)$ and $Q_{\pi^*}(s,a) = Q^*(s,a)$. These definitions are leveraged by some methods, which estimate the optimal value function and then derive the optimal policy following Equation 3.11.

3.3 Types of Learning

Training RL agents consists in iterating over two phases: collecting experience and learning from it. This section is concerned with the distribution of the data we are going to learn from, and discussion on what to learn from the gathered experience is deferred to the following sections.

In the most general setting, we would like to learn about some target policy π using data collected from an arbitrary behavior policy μ. We can distinguish between two different scenarios depending on the relationship between π and μ, namely on-policy and off-policy learning.

On-policy learning. The behavior and target policy are the same, i.e. $\pi = \mu$. This setup simplifies the mathematical formulation, as there is no mismatch between data distributions. Since the collected data needs to be discarded after every update to the target policy, on-policy learning is often data inefficient.

Off-policy learning. The behavior policy is different from the target policy, i.e. $\pi \neq \mu$. Note that this is the case even when using data collected by a slightly outdated version of the target policy. All value functions and objectives we discussed so far compute expectations with respect to the target policy π, forcing us to account for the mismatch in the data distribution (e.g. through importance sampling). This setting is more generic and data efficient than on-policy learning, but these advantages come at the cost of a more complex and potentially unstable formulation.

3.4 Common Approaches

We will now discuss some of the major families of model-free RL methods. Within each of these families, one can derive on-policy and off-policy variants with different properties and requirements. The goal of this section is not to give an extensive overview of all existing approaches, but to provide a comprehensive introduction to the most common ways of parameterizing agents. We refer interested readers to Sutton and Barto [45] for detailed descriptions of commonly used methods.

3.4.1 Value-Based

As their name suggests, these methods are based on value function estimation. There need not be an explicit policy, which can be implicitly derived from the value function estimate. Assuming access to an initial estimate of $Q(s, a)$, which can be random, these methods repeatedly apply the following steps:

Policy improvement. A new policy is derived from the action-value function estimate, $\pi(s) \leftarrow \text{argmax}_a Q(s, a)$. If the action space is discrete and its cardinality is low, this can be achieved through exhaustive search. Otherwise, one can train an approximate sampler that predicts which actions would achieve the maximum value at each state [46].

Policy evaluation. The action-value function estimate $Q(s, a)$ is updated using data collected with the behavior policy. There exist multiple ways of defining the targets for the estimates, e.g. using Monte Carlo over complete episodes or bootstrapping with current estimates in a temporal-difference learning fashion.

There exist proofs demonstrating that value-based methods converge to the optimal value function $Q^*(s, a)$ under certain assumptions, from which the optimal policy can be derived.

3.4.2 Policy Gradient

These methods directly optimize the parameters of an explicit policy, $\pi_\theta(a|s)$, to maximize the objective in Equation 3.3. Let us rewrite the objective in terms of the policy parameters:

$$\mathcal{J}(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=1}^{\infty} \gamma^{t-1} R_t\right] = \mathbb{E}_{\pi_\theta}\left[G_0\right] \tag{3.12}$$

We can now frame the optimization problem:

$$\theta^* = \operatorname*{argmax}_\theta \mathcal{J}(\theta) \tag{3.13}$$

We can use the Policy Gradient Theorem [45, Section 13.2] to derive the gradient of the objective $\mathcal{J}(\theta)$ with respect to the policy parameters θ:

$$\nabla_\theta \mathcal{J}(\theta) = \mathbb{E}_{\pi_\theta}\left[G_0 \nabla_\theta \log \pi(a|s)\right] \tag{3.14}$$

which provides a strategy for optimizing θ by gradient ascent, using any of the methods discussed in Section 2.2.1. The implications of this result is that the log-likelihood of actions should be increased proportionally to the rewards they lead to.

Equation 3.14 presents the most basic form of the policy gradient update rule, which provides an unbiased but high variance direction of improvement, and there exist many ways improvements to it. For instance, one could replace G_0 with G_t in Equation 3.12 to account for the fact that current actions cannot change the past, or subtract a learned baseline to reduce variance. A common practice for trading off bias and variance in the policy gradient update consists in using a learned value estimate, giving rise to *actor-critic* algorithms. These methods bootstrap with a learned value function estimate after a few steps instead of using full Monte Carlo estimates. For instance:

$$G_t \approx \sum_{k=1}^{N} \gamma^{k-1} R_{t+k} + \gamma^N V(s_{t+N+1}) \tag{3.15}$$

where the intuition is that Monte Carlo returns are unbiased and high variance, whereas the learned estimate is biased but has a low variance. Bootstrapping also allows updating the policy more often, as the return estimate does not rely on full episodes, which generally leads to faster convergence.

These are just some of the multiple variants of the policy gradient update rule that can be found in the literature. We refer the interested reader to Schulman et al. [47] for a detailed overview of these variants.

3.4.3 Direct Policy Search

Derivative-free methods, also known as zero-order methods, optimize the objective by evaluating it only at some positions of the parameter space. Each iteration of these methods generally consists of three steps: (1) generating a list of candidate solutions, (2) evaluating the loss function on the candidates, and (3) returning the best solution. The families of genetic algorithms and evolution strategies are well known examples of derivative-free methods, and we refer the reader to Ha [48] for a comprehensive introduction to the field. Direct policy search methods apply this strategy to RL tasks by directly searching in parameter space for the vector of weights θ that maximizes the returns obtained by a policy π_θ.

Direct policy search methods have recently achieved impressive results in RL benchmarks while offering almost perfect scalability to thousands of cores [49, 50]. Their scalability make them very appealing when short iteration times are needed, which compensates for their reduced data efficiency when simulating the environment is cheap. We will present improvements to a state of the art evolutionary approach in Chapter 7.

Part I

Learning from Examples

Introduction

Methods based on deep neural networks that are trained from examples have established the state-of-the-art in multiple domains and tasks such as computer vision [6], machine translation [51] and speech generation [52]. The power of these methods stems from their ability to learn arbitrary input to output mappings from example pairs. This mapping, which is initially random, is progressively improved by comparing the predictions made by the model with the correct outputs. Learning complex and accurate mappings with these self-correcting systems requires from many iterations and examples. This explains why, despite their development spans over many decades [53], their potential has been only recently unlocked thanks to the creation of large-scale datasets [5, 54] and the increased computational power of specific accelerators such as GPUs.

Even with the use of specific hardware devices, training these algorithms is so computationally intensive that it can take days, or even weeks, to converge on a single machine. The pace of advances in machine learning is frequently upper bounded by the time taken to train models, and shortening training times has become a crucial challenge both for research and industrial applications. Even though hardware manufacturers continuously provide improvements in computational power [55], the community has turned to distributed solutions for further reducing training times [56] and training larger models [57]. However, accelerating an algorithm by distributing it across several computing devices is not always a trivial task. The communication overhead precludes the distribution of some methods beyond a reduced number of machines [58], and sometimes parallel training can even hinder the final performance of the model when done naively [59]. This motivates research efforts towards developing algorithms that are well suited for parallel training, from both learning and computational standpoints. Chapter 4 presents strategies for accelerating training of neural networks on a homogeneous GPU cluster. We obtain important speedups when leveraging a large number of GPUs, enabling faster iteration over research ideas and hypotheses.

Neural networks are often trained on high performance computing facilities, leveraging distributed training techniques for shortening training times and training larger models.

Large models are generally more accurate, but their computational demands might preclude their deployment on devices with limited computational power. Devising strategies for managing the trade-off between computational requirements and model accuracy is crucial for deploying deep learning models on devices with varying computational powers. In Chapter 5, we address this problem in the context of Recurrent Neural Networks (RNNs) that process sequential inputs. We introduce a novel RNN architecture that not only maximizes accuracy on the task at hand, but attempts to do so while skipping as many elements in the input sequence as possible. The skipping patterns are automatically discovered by the model, without explicit supervision, based on the training data and the limitations imposed on the available computation. By making the neural network aware of its own computational requirements, we are able to train a variety of models for different computational budgets with little impact in its final accuracy.

Increased compute and larger datasets alone are not enough for training very large and accurate models. Naively increasing the model depth generally leads to unstable training dynamics caused by exploding or vanishing signals within the network. Modern architectures composed of dozens of layers base their success on a series of building blocks that have been developed over the years, such as initialization schemes [7, 60], normalization techniques [61, 62], residual connections [63], and advanced optimizers [42, 43]. Most of these advances provide solutions for problems that are only revealed with scale, highlighting that low-level algorithmic improvements are key when it comes to developing large-scale deep learning models. Motivated by this observation, Chapter 6 introduces a novel initialization scheme for weight normalized and residual networks, two important building blocks in the deep learning toolbox. We show that our initialization strategy provides more stable training dynamics, which ultimately results in stronger generalization, and enables training much deeper networks with hundreds of layers.

4

Distributed Training of Convolutional Neural Networks

Víctor Campos, Francesc Sastre, Maurici Yagües, Míriam Bellver, Xavier Giró-i-Nieto, and Jordi Torres. Distributed training strategies for a computer vision deep learning algorithm on a distributed GPU cluster. *Procedia Computer Science*, 2017

Increasing the scale of deep learning models with respect to the number of training examples and the number of model parameters can drastically improve their accuracy. These improvements come at the cost of increased computational demands, which are often hard to meet when using a single machine, and distributed computation has emerged as a useful strategy towards training larger models on more examples [56]. However, from both learning and high performance computing standpoints, the best strategy for distributing training of neural networks is strongly influenced by the underlying hardware configuration. The main contribution of this chapter is an empirical evaluation of distributed training strategies on a homogeneous GPU cluster. We start by studying how to take advantage of the fact that nodes are equipped with multiple GPUs. We then compare different strategies for distributing the training process, and show that this choice can have a strong impact in the hardware utilization due to the homogeneous nature of the cluster. Finally, we analyze the impact of distributed training on the learning progress and the final quality of the model.

4.1 Related Work

The massive number of convolutions and matrix multiplications in neural networks has led to GPU implementations with CUDA [64] and efficient task-specific primitives using cuDNN [65]. Early deep learning frameworks such as Caffe [66] provided fast and easy access to such primitives, but were initially designed for single machine operation, without support for distributed environments. Efforts towards distributing the former frameworks with traditional high performance computing tools resulted in projects such as SparkNet [67] or Theano-MPI [68]. Native support for distributed settings is included in more recent frameworks such as TensorFlow [69], MXNet [70] and PyTorch [71]. However, scaling the training algorithms from a single machine environment to a distributed setting poses two main challenges. From the computing performance standpoint, the main goal is optimizing the resource utilization. From the learning side, the final accuracy of the model should not suffer a drop when compared to its single machine counterpart.

We can distinguish two main approaches to train large neural networks using multiple GPUs, namely *model parallelism* and *data parallelism* [72]. Model parallelism splits layers in the neural network among different GPUs, i.e. each GPU operates over the same batch of input data, but applying different operations on them. This strategy is mostly used for models with a large number of parameters that may not fully fit in a single GPU. Data parallelism places a replica of the model on each GPU, which then operates on a different batch of data. Since model replicas share parameters, this method is equivalent to virtually increasing the memory of a single GPU so that it can fit larger batches. Unlike model parallelism, data parallelism only introduces one synchronization point regardless of the number of GPUs, thus reducing communication overhead and making it more suitable for current neural network architectures. Balancing the load between GPUs is straightforward in this paradigm, while it would require from careful tuning for each specific model architecture and number of GPUs in a model parallelism approach. For these reasons, we will consider multi-GPU data parallelism for both single machine and distributed settings.

4.2 Distributed Training through Data Parallelism

The distributed data parallelism paradigm generally considers two types of nodes: *parameter servers* and *workers* [58]. Parameter server nodes store and update the model parameters [69]. Worker nodes hold replicas of the model, each operating on a separate batch of data. Each worker loops over three steps: (1) receiving updated model parameters from the parameter servers, (2) performing forward and backward passes through

the model to compute gradients over a batch of data, and (3) communicating gradients to the parameter servers in order to update the model. Figure 4.1 depicts the information flow in the distributed data parallel setting with multiple parameter servers and nodes. We can distinguish different flavours of data parallelism depending on how the communication between workers and parameter servers is handled.

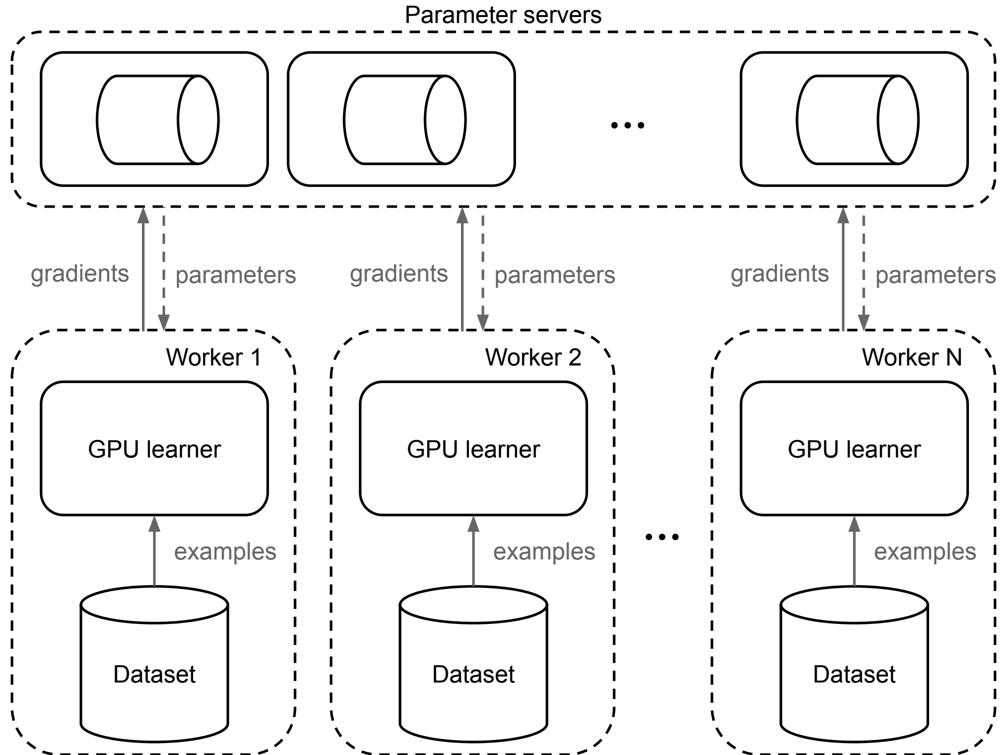


Figure 4.1: Distributed data parallel setting with multiple parameter servers and nodes. Each worker performs forward and backward passes through the model on independent batches of data, and communicates the gradients to the parameter servers. The latter aggregate gradients from the different workers, update the model and broadcast the most recent set of parameters.

Synchronous mode. Parameter servers wait until all worker nodes have computed the gradients with respect to their data batches. Once the gradients are received by the parameter servers, they are applied to the current weights and the updated model is sent back to all the worker nodes. The speed of the system will be determined by the slowest node, as no updates are performed until all worker nodes finish the computation. Some clusters might suffer from unbalanced network speeds, e.g. when shared with other users, which might hamper the training process. On the other hand, aggregating gradients over larger batch sizes might reduce the variance of the updates and provide faster convergence. Chen et al. [58] trade off hardware efficiency for resilience by using backup workers, i.e. launching $M > N$ workers but computing updates only over the fastest N workers at each iteration.

Asynchronous mode. Model parameters are updated using gradients received from every worker individually. This prevents workers from waiting until all results are available to the parameter servers, increasing the throughput of the system. The price to pay for such throughput increase is *gradient staleness*, as workers will generally compute updates on slightly outdated versions of the model[1]. This introduces bias in gradient estimates, which in practice results in models trained with asynchronous updates needing a larger number of updates to reach a given loss target than their synchronous counterparts. Too large a number of asynchronous workers might create a bottleneck on the parameter servers side, which now need to update and communicate weights more frequently. This issue can be alleviated by increasing the number of parameter servers as the number of workers grows.

Mixed mode. The pace at which pairs of nodes can communicate might differ depending on the topology of the cluster. The mixed mode aims at taking advantage of this by performing synchronous aggregation of gradients over subsets of workers, and then performing asynchronous updates in the parameter server side. This offers the advantages of reduced gradient variance of the synchronous mode, with the high throughput achieved by asynchronous updates. Moreover, aggregating gradients over subsets of workers reduces the communication burden on the parameter servers and the level of gradient staleness.

Others. For completeness, we note that there exist improvements on the traditional Stochastic Gradient Descent algorithm that can be applied to distributed settings [73–75]. We instead focus on scaling problems from single node to distributed settings with minimal modifications to the underlying training algorithm.

4.3 Adjective Noun Pair Detection

We will consider the task of Adjective Noun Pair (ANP) detection from as images as a case study for analyzing distributed training of Convolutional Neural Networks (CNNs). This is an important task within Affective Computing [76], a field that has recently garnered much research attention. Machines that are able to understand and convey subjectivity and affect would lead to a better human-computer interaction that is key in some fields such as robotics or medicine. Despite the success in some constrained environments such as emotional understanding of facial expressions [77], automated affect understanding in unconstrained domains remains an open challenge which is still far from other tasks where machines are approaching or have even surpassed human performance.

[1] In a setting with N asynchronous workers, each worker will compute gradients on a model replica that is on average $N-1$ updates behind the latest version.

The inherent complexity to emotions, i.e. a high intensity, but relatively brief experience, onset by a stimuli [78, 79], and sentiment, i.e. an attitude, disposition or opinion towards a certain topic [80], is reflected in categories that suffer from a large intra-class variance. This challenge has been addressed with the creation of Visual Sentiment Ontologies [81, 82], consisting of a large-scale collection of ANPs, which focus on emotions expressed by content owners in the images. These concepts, while exhibiting a reduced intra-class variance as compared to emotions or sentiments, still convey strong affective connotations and can be used as mid-level representations for visual affect related tasks. The noun component in an ANP can be understood to ground the visual appearance of the entity, whereas the adjective polarizes the content towards a positive or negative sentiment, or emotion [82]. These properties try to bridge the *affective gap* between low level image features and high level affective semantics, which goes far beyond recognizing the main object in an image. Whereas a traditional object classification algorithm may recognize a *baby* in an image, a finer-grained classification such as *happy baby* or *crying baby* is usually needed to fully understand the affective content being conveyed. Capturing the sophisticated differences between ANPs poses a challenging task that benefits from leveraging large-scale annotated datasets by means of high learning capacity models [83].

Jou et al. [82] built a Multilingual Visual Sentiment Ontology (MVSO) with over 156,000 ANPs from 12 different languages, extending the Visual Sentiment Ontology [81]. In order to guarantee a link between emotions and the concepts in the ontology, emotion keywords from a well-known emotion model derived from psychology studies, the *Plutchik's Wheel of Emotions* [78], were used to query the Flickr API[2] and retrieve a large corpus of images with related tags and metadata. After a data-driven filtering of ANP candidates, visual examples for these concepts were retrieved by querying the Flickr API for images containing them either in their tags or metadata. The resulting MVSO dataset contains over 15M images annotated with sentiment-biased ANPs. MVSO presents a major challenge when training visual concept detectors: there is no human-level supervision for the ground truth, so that the annotations have to be considered as *weak labels*. This issue can be mitigated by using a restricted set of the English partition of MVSO, namely the tag-restricted subset [84], that contains over 1.2M images belonging to 1,200 classes where the associated ANP was found in the image tags and are more likely to have reliable annotations.

The high level of abstraction of ANPs makes their detection a challenging task. This problem was originally addressed with hand-crafted features and Support Vector Machines [85], which have been surpassed by CNNs [82–84]. Motivated by its importance for Affective Computing applications, as well as the strong performance shown by CNNs

[2]https://www.flickr.com/services/api

on this task, we will study distributed training of CNNs in the context of ANP detection from images.

4.4 Experiments

We run experiments on the bullx R421-E4 servers of the Minotauro[3] supercomputer at the Barcelona Supercomputing Center. These 39 nodes form a homogeneous GPU cluster with a peak performance above 250TFLOPs. Each node is equipped with two NVIDIA Kepler K80 dual-GPU cards, two Intel Xeon E5-2630 8-core processors, and 128GB of RAM. Inter-node communication is performed trough a 56Gb/s InfiniBand network. The model is implemented with TensorFlow [69], running on CUDA 7.5 and using cuDNN 5.1.3 primitives for improved performance. The training process is submitted through the Slurm Workload Manager, which schedules and assigns resources for each job in the cluster. Task distribution and communication between nodes within each job is handled by Greasy [86], which is able to schedule and run a list of tasks in parallel using the resources assigned to each job.

All experiments consider the ResNet50 architecture [63]. It contains 50 layers of trainable parameters mapping a $224 \times 224 \times 3$ input image to a 1,200-dimensional vector representing a probability distribution over the ANP classes in the tag-restricted subset of MVSO. The model contains over 25×10^6 single-precision floating-point parameters, which are involved in over 4×10^9 floating-point operations and are tuned during training. It is important to notice that computationally demanding models can benefit the most from distributed training, as the added communication overhead will be small when compared to the time spent parallelizing computation.

Models are trained to minimize the cross-entropy loss between the output of the model and the the real class distribution. This cost function is minimized using the RMSProp [42] optimizer with a learning rate of 0.1, decay of 0.9, $\epsilon = 1.0$, and a weight decay rate of 10^{-4}. Each GPU processes 32 images at a time. To prevent overfitting, data augmentation consisting in random crops and/or horizontal flips is asynchronously performed on CPU while previous batches are processed by the GPUs. The weights of the network are initialized using a model pre-trained on ImageNet [5], practice that has been proven beneficial even when training on large-scale datasets [87].

[3] https://www.bsc.es/marenostrum/minotauro

4.4.1 Intra-Node Parallelism

We first study the scalability of data parallelism with synchronous model updates on a single multi-GPU machine. Due to the dual nature of the NVIDIA K80 cards, up to four model replicas can be deployed per node. The variables in the computation graph are stored in RAM in order to ensure proper weight sharing between model replicas. Each GPU performs the forward and backward passes through the network on independent batches of data. Gradients computed for each replica are averaged before performing the weights update, step that becomes the synchronization point in the graph.

The evolution of the throughput as a function of the number of GPUs is reported in Figure 4.2. We observe that the speedup is almost linear when increasing the number of GPUs, confirming the adequacy of synchronous updates for intra-node parallelism.

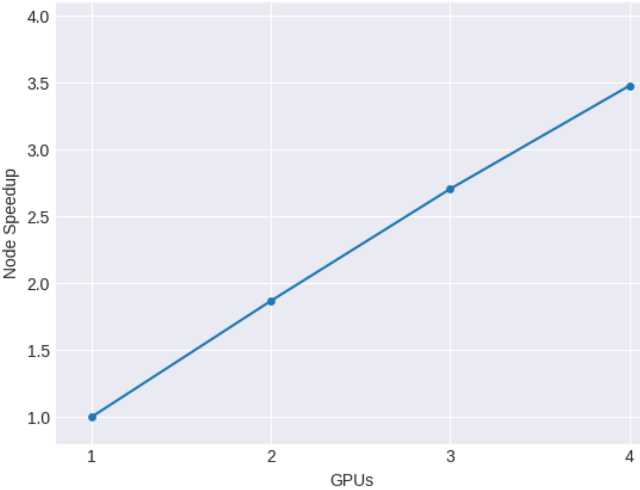


Figure 4.2: Throughput increase when parallelizing training over different number of GPUs within a single node. Throughput is measured in processed images per second.

4.4.2 Distributed Training

Based on the results for the single-node scenario, we adopt a mixed distributed training approach. Gradients over the four model replicas in each node are aggregated synchronously, resulting in an effective batch size of 128 samples per worker. The results from different nodes are then aggregated asynchronously at the parameter servers. This strategy differs from that of previous works, where each worker is defined as a single GPU [58, 88], and offers two main advantages: (1) communication overhead is reduced, as only a single collection of gradients needs to be exchanged through the network for each set of four model replicas, and (2) each worker has a larger effective batch size, providing better gradient estimates and allowing the use of larger learning rates for faster convergence.

4.4.2.1 Throughput Analysis

Previous studies on distributed training with TensorFlow tend to use different node configurations for worker and parameter server tasks [58, 69]. Given that a parameter server only stores and updates the model and there is no need for GPU computations, CPU-only machines are used for this task. Worker jobs need to perform forward and backward passes over the model, which can be greatly accelerated on GPU. Since the cluster used in our experiments is homogeneous, placing parameter servers and workers in different nodes would result in under-utilization of GPU resources. This section studies the impact of sharing resources between parameter servers and workers instead of using dedicated machines for the parameter servers.

We start by running distributed training experiments with 16 GPUs. These resources are split across four asynchronous workers, each averaging gradients over four models replicas. A round robin strategy is followed when splitting model variables across parameter servers, so we always consider an odd number of such nodes to obtain a balanced partition. Table 4.1 reports the speedups obtained under different settings, and the trade-off between throughput and resource utilization when using dedicated parameter server nodes in a homogeneous cluster. Using three dedicated parameter servers provides the largest speedups, but does not compensate for the increased hardware requirements.

Nodes	**Configuration**	**Throughput**	**Speedup**	**Efficiency**
1	–	124.18 img/sec	-	-
4	4W, 1PS	292.22 img/sec	2.35	0.58
4	4W, 3PS	383.09 img/sec	3.09	0.77
7	4W, 3PS	396.62 img/sec	3.19	0.46

Table 4.1: Scalability analysis for a varying number of nodes and parameter servers (PS), given a fixed number of workers (W). Using dedicated nodes for the parameter servers slightly improves the throughput, but involves a much larger resource utilization. Efficiency is computed as the ratio between speedup and the number of nodes, providing a more accurate measure of resource utilization.

Results suggest that sharing resources across workers and parameter servers might be the most efficient strategy for homogeneous GPU clusters. To further evaluate this hypothesis, we run experiments where we vary the number of workers and parameter servers for a given number of nodes. Figure 4.3 confirms that sharing resources across parameter servers and workers provides important throughput gains.

A useful way for assessing the scalability of a distributed method consists in measuring how throughput changes as the amount of resources increases. We measure this for the setting where all nodes are fully utilized by sharing resources between parameter servers and workers. We run experiments with N nodes and workers, with $N \in \{1, 2, 3, 4\}$.

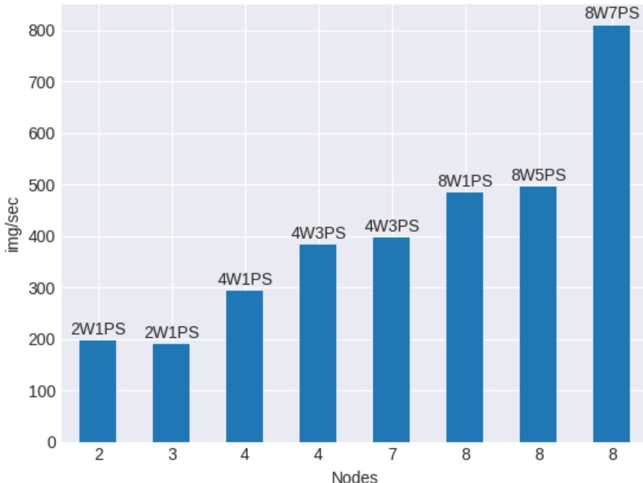


Figure 4.3: Throughput comparison between different distributed setups, where we vary the number of nodes, workers (W), and parameter servers (PS). Setting the proper number of parameter servers is key to maximizing throughput.

The best configuration in Figure 4.3 is adopted for each value of N, which corresponds to launching $N-1$ parameter servers. Figure 4.4 confirms that the throughput scales gracefully with the amount of resources.

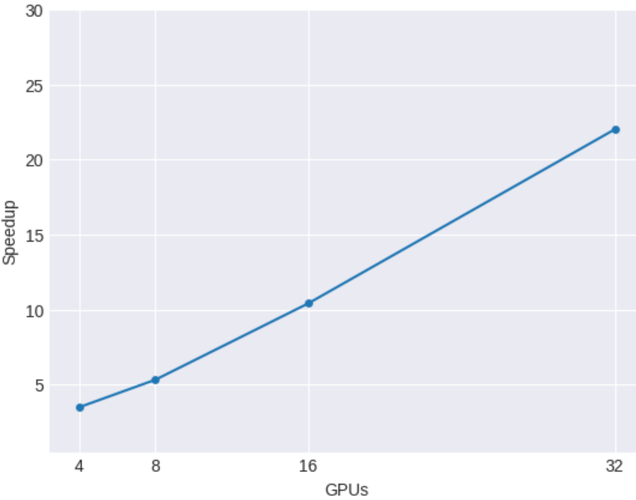


Figure 4.4: Throughput increase in the distributed setting, computed with respect to the single-GPU scenario. We employ nodes with four GPUs each.

4.4.2.2 Convergence Analysis

There are two mains aspects to take into consideration when studying the impact of distributed training on the learning process. The time required to reach a target loss value on the training set, which is the metric that is being optimized, is a good proxy for measuring the actual speedup on the training process. On the other hand, the

Workers	GPUs	Accuracy	Time (h)	Speedup
1	4	0.228	106.43	1.00
2	8	0.217	62.78	1.69
4	16	0.202	37.99	2.80
8	32	0.217	22.50	4.73

Table 4.2: Accuracy on the validation set for different numbers of asynchronous workers. We pick the highest validation accuracy over the whole training run, and report the time elapsed to achieve it. Setups with more nodes achieve higher throughputs, but also require a larger number of iterations to converge.

final accuracy achieved by the model will determine whether distributing training across several machines has an impact on the capabilities for finding the desired minima of the cost function.

We start by analyzing the time required to reach some target loss value. Despite the throughput increase is close to linear with respect to the number of nodes, Figure 4.5 shows that the rate at which the loss decreases does not improve so gracefully as additional nodes are used. This result is explained by the gradient staleness problem, which increases linearly with the number of nodes in asynchronous schemes.

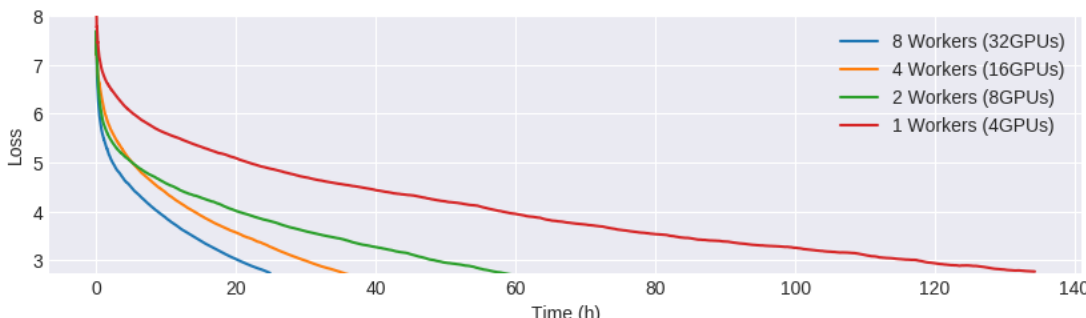


Figure 4.5: Train loss evolution for the different distributed configurations. The more nodes, the faster a target loss value is reached.

The final accuracy rates on the validation set for different distributed configurations is reported in Table 4.2. None of them is able to reach the final accuracy of the model trained in a single node, confirming that the stale gradients have a negative impact on the local optima to which training converges. Another possible reason for the performance drop might be related to the lack of hyperparameter tuning, which were not optimized for each configuration due to the cost of running distributed experiments. An important observation is that balancing the throughput across workers was critical in order to achieve a successful learning process. Workers that are constantly lagging behind aggravate the stale gradients problem and destabilize training.

4.5 Discussion

Distributed training strategies for deep learning architectures will become more important as the size of datasets increases. They allow researchers to receive earlier feedback on their ideas and increase the pace at which algorithms are developed, thus understanding the best practices to distribute training of these models is a key research area. We studied how to adapt the training algorithm to the available hardware resources in order to accelerate the training of a CNN on a homogeneous GPU cluster. First, we showed how close to linear speedups can be achieved through intra-node parallelism. Based on these results, we developed a mixed approach where this efficiency is leveraged and the amount of inter-node communication is reduced as compared to a pure asynchronous policy. When properly tuning the number of parameter servers for each configuration, this method yields an important speedup in the number of samples per second processed by the system even for the setup with the minimum hardware overhead.

In spite of the good scalability demonstrated in terms of throughput, configurations with more nodes require from additional training steps to reach the same target loss value, although the increased throughput compensates this issue and still reduces the training time considerably. This drawback becomes more important when increasing the number of nodes, and our results suggest that different strategies should be employed for highly distributed settings with dozens of nodes.

Creating tools that provide insight on the performance of each individual component can help with detecting bottlenecks and pushing even further the scalability of the system. We believe that insights from other domains within high performance computing might help in improving the scalability of distributed training strategies. Improving the performance of the synchronous setting is an important research direction, as it overcomes the stale gradients problem but is currently limited by the performance of the slowest worker. Developing a better understanding of the role of the batch size in neural network optimization and generalization [89] is also key in order to scale up approaches that rely on data parallelism.

5

Learning to Skip State Updates in Recurrent Neural Networks

Víctor Campos, Brendan Jou, Xavier Giró-i-Nieto, Jordi Torres, and Shih-Fu Chang. Skip RNN: Learning to skip state updates in recurrent neural networks. In *ICLR*, 2018

Recurrent Neural Networks (RNNs) have become the standard approach for practitioners when addressing machine learning tasks involving sequential data. Such success has been enabled by the appearance of larger datasets, more powerful computing resources and improved architectures and training algorithms. Gated units, such as the Long Short-Term Memory (LSTM) [38] and the Gated Recurrent Unit (GRU) [39], were designed to deal with the vanishing gradients problem commonly found in RNNs [90]. These architectures have been popularized, in part, due to their impressive results on a variety of tasks in machine translation [35], language modeling [36] and speech recognition [37].

Some of the main challenges of RNNs are in their training and deployment when dealing with long sequences, due to their inherently sequential behaviour. These challenges include throughput degradation, slower convergence during training and memory leakage, even for gated architectures [91]. Sequence shortening techniques, which can be seen as a sort of conditional computation [92–94] in time, can alleviate these issues. The most common approaches, such as cropping discrete signals or reducing the sampling rate in continuous ones, are based on heuristics and can be suboptimal. In contrast, we propose a model that is able to learn which samples (i.e., elements in the input sequence) need to be used in order to solve the target task. Consider a video understanding task as an

example: scenes with large motion may benefit from high frame rates, whereas only a few frames are needed to capture the semantics of a mostly static scene.

The main contribution of this chapter is a novel modification for existing RNN architectures that allows them to skip state updates, decreasing the number of sequential operations performed, without requiring any additional supervision signal. This model, called Skip RNN, adaptively determines whether the state needs to be updated or copied to the next time step. We show how the network can be encouraged to perform fewer state updates by adding a penalization term during training, allowing us to train models under different computation budgets. The proposed modification can generally be integrated with any RNN, and we show implementations with well-known RNNs, namely LSTM and GRU. The resulting models show promising results on a series of sequence modeling tasks. In particular, we evaluate the proposed Skip RNN architecture on six sequence learning problems: an adding task, sine wave frequency discrimination, digit classification, sentiment analysis in movie reviews, action classification in video, and temporal action localization in video.

5.1 Related Work

Conditional computation has been shown to gradually increase model capacity without proportional increases in computational cost by exploiting certain computation paths for each input [57, 92, 95–97]. This idea has been extended in the temporal domain, such as in learning how many times an input needs to be "pondered" before moving to the next one [98], or designing RNN architectures whose number of layers depend on the input data [99]. Other works have addressed time-dependent computation in RNNs by updating only a fraction of the hidden states based on the current hidden state and input [100], or following periodic patterns [91, 101]. However, due to the inherently sequential nature of RNNs and the parallel computation capabilities of modern hardware, reducing the size of the matrices involved in the computations performed at each time step generally has not accelerated inference as dramatically as hoped. The proposed Skip RNN model can be seen as form of conditional computation in time, where the computation associated to the RNN updates may or may not be executed at every time step. This idea is related to the UPDATE and COPY operations in hierarchical multiscale RNNs [99], but applied to the whole stack of RNN layers at the same time. This difference is key to allowing our approach to skip input samples, effectively reducing sequential computation and shielding the hidden state over longer time lags. Learning whether to update or copy the hidden state through time steps can be seen as a learnable

Zoneout mask [102] which is shared between all the units in the hidden state. Similarly, it can be interpreted as an input-dependent recurrent version of stochastic depth [103].

Selecting parts of the input signal is similar in spirit to the hard attention mechanisms that have been applied to image regions [104], where only some patches of the input image are attended in order to generate captions [105] or detect objects [106]. Our model can be understood as generating a hard temporal attention mask on-the-fly given previously seen samples, deciding which time steps should be attended and operating on a subset of input samples. Subsampling input sequences has been explored for visual storylines generation [107], although jointly optimizing the RNN weights and the subsampling mechanism is often computationally infeasible and an Expectation-Maximization algorithm is used instead. Similar research has been conducted for video analysis tasks, discovering minimally needed evidence for event recognition [108] and training agents that decide which frames need to be observed in order to localize actions in time [109, 110]. Motivated by the advantages of training recurrent models on shorter subsequences, efforts have been conducted on learning differentiable subsampling mechanisms [111], although the computational complexity of the proposed method precludes its application to long input sequences. In contrast, our proposed method can be trained with backpropagation and does not degrade the complexity of the baseline RNNs.

Accelerating inference in RNNs is difficult due to their inherently sequential nature, leading to the design of Quasi-Recurrent Neural Networks [112] and Simple Recurrent Units [113], which relax the temporal dependency between consecutive steps. With the goal of speeding up RNN inference, LSTM-Jump [114] augments an LSTM cell with a classification layer that will decide how many steps to jump between RNN updates. Despite its promising results on text tasks, the model needs to be trained with REINFORCE [115], which requires defining a reasonable reward signal. Determining these rewards is non-trivial and may not necessarily generalize across tasks. Moreover, the number of tokens read between jumps, the maximum jump distance and the number of jumps allowed all need to be chosen in advance. These hyperparameters define a reduced set of subsequences that the model can sample, instead of allowing the network to learn any arbitrary sampling scheme. Unlike LSTM-Jump, our proposed approach is differentiable, thus not requiring any modifications to the loss function and simplifying the optimization process, and is not limited to a predefined set of sample selection patterns.

5.2 Model Description

An RNN takes an input sequence $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_T)$ and generates a state sequence $\mathbf{s} = (\mathbf{s}_1, \dots, \mathbf{s}_T)$ by iteratively applying a parametric state transition model S from

$t = 1$ to T:

$$\mathbf{s}_t = \mathcal{S}(\mathbf{s}_{t-1}, \mathbf{x}_t) \tag{5.1}$$

We augment the network with a binary *state update gate*, $u_t \in \{0, 1\}$, selecting whether the state of the RNN will be updated ($u_t = 1$) or copied from the previous time step ($u_t = 0$). At every time step t, the probability $\tilde{u}_{t+1} \in [0, 1]$ of performing a state update at $t + 1$ is emitted. The resulting architecture is depicted in Figure 5.1 and can be characterized as follows:

$$u_t = f_{\text{binarize}}(\tilde{u}_t) \tag{5.2}$$

$$\mathbf{s}_t = u_t \cdot \mathcal{S}(\mathbf{s}_{t-1}, \mathbf{x}_t) + (1 - u_t) \cdot \mathbf{s}_{t-1} \tag{5.3}$$

$$\Delta\tilde{u}_t = \sigma(\mathbf{W}_u \mathbf{s}_t + b_u) \tag{5.4}$$

$$\tilde{u}_{t+1} = u_t \cdot \Delta\tilde{u}_t + (1 - u_t) \cdot (\tilde{u}_t + \min(\Delta\tilde{u}_t, 1 - \tilde{u}_t)) \tag{5.5}$$

where $\mathbf{W}_u$ is a weights vector, b_u is a scalar bias, σ is the sigmoid function and $f_{\text{binarize}} : [0, 1] \rightarrow \{0, 1\}$ binarizes the input value. Should the network be composed of several layers, some columns of $\mathbf{W}_u$ can be fixed to 0 so that $\Delta\tilde{u}_t$ depends only on the states of a subset of layers (c.f. Sections 5.3.5 and 5.3.6 for examples with two layers). We implement f_{binarize} as a deterministic step function $u_t = \text{round}(\tilde{u}_t)$, although a stochastic sampling from a Bernoulli distribution $u_t \sim \text{Bernoulli}(\tilde{u}_t)$ would be possible as well.

The model formulation encodes the observation that the likelihood of requesting a new input to update the state increases with the number of consecutively skipped samples. Whenever a state update is omitted, the pre-activation of the state update gate for the following time step, $\tilde{u}_{t+1}$, is incremented by $\Delta\tilde{u}_t$. On the other hand, if a state update is performed, the accumulated value is flushed and $\tilde{u}_{t+1} = \Delta\tilde{u}_t$.

The number of skipped time steps can be computed ahead of time. For the particular formulation used in this work, where f_{binarize} is implemented by means of a rounding function, the number of skipped samples after performing a state update at time step t is given by:

$$N_{\text{skip}}(t) = \min\{n : n \cdot \Delta\tilde{u}_t \geq 0.5\} - 1 \tag{5.6}$$

where $n \in \mathbb{Z}^+$. This enables more efficient implementations where no computation at all is performed whenever $u_t = 0$. These computational savings are possible because $\Delta\tilde{u}_t = \sigma(\mathbf{W}_u \mathbf{s}_t + b_u) = \sigma(\mathbf{W}_u \mathbf{s}_{t-1} + b_u) = \Delta\tilde{u}_{t-1}$ when $u_t = 0$ and there is no need to evaluate it again, as depicted in Figure 5.1d.

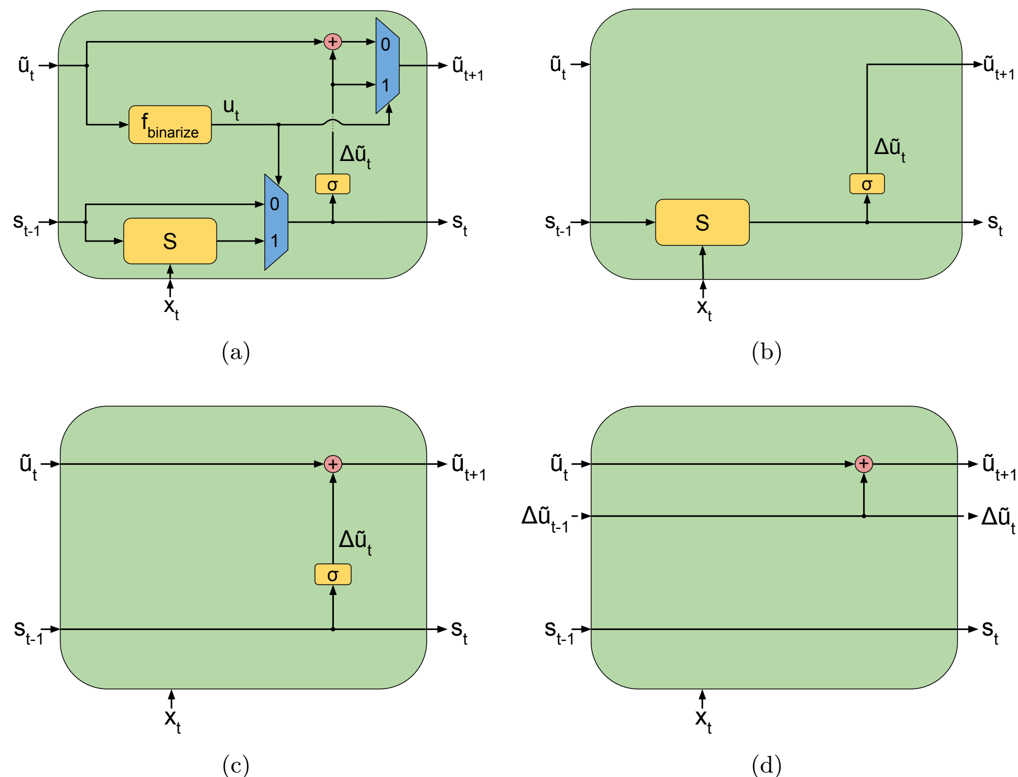


Figure 5.1: Model architecture of the proposed Skip RNN. **(a)** Complete Skip RNN architecture, where the computation graph at time step t is conditioned on u_t. **(b)** Architecture when the state is updated, i.e. $u_t = 1$. **(c)** Architecture when the update step is skipped and the previous state is copied, i.e. $u_t = 0$. **(d)** In practice, redundant computation is avoided by propagating $\Delta\tilde{u}_t$ between time steps when $u_t = 0$.

There are several advantages in reducing the number of RNN updates. From the computational standpoint, fewer updates translates into fewer required sequential operations to process an input signal, leading to faster inference and reduced energy consumption. Unlike some other models that aim to reduce the average number of operations per step [91, 100], ours enables skipping steps completely. Replacing RNN updates with copy operations increases the memory of the network and its ability to model long term dependencies even for gated units, since the exponential memory decay observed in LSTM and GRU [91] is alleviated. During training, gradients are propagated through fewer updating time steps, providing faster convergence in some tasks involving long sequences. Moreover, the proposed model is orthogonal to recent advances in RNNs and could be used in conjunction with such techniques, e.g. normalization [116, 117], regularization [36, 102], variable computation [91, 100] or even external memory [118, 119].

5.2.1 Error Gradients

The whole model is differentiable except for f_{binarize}, which outputs binary values. A common method for optimizing functions involving discrete variables is REINFORCE [115],

although several estimators have been proposed for the particular case of neurons with binary outputs [92]. We select the straight-through estimator [92, 120], which consists of approximating the step function by the identity when computing gradients during the backward pass:

$$\frac{\partial f_{\text{binarize}}(\mathbf{x})}{\partial \mathbf{x}} = 1 \tag{5.7}$$

This yields a biased estimator that has proven more efficient than other unbiased but high-variance estimators such as REINFORCE [92] and has been successfully applied in different works [99, 121]. By using the straight-through estimator as the backward pass for f_{binarize}, all the model parameters can be trained to minimize the target loss function with standard backpropagation and without defining any additional supervision or reward signal.

5.2.2 Limiting Computation

The Skip RNN is able to learn when to update or copy the state without explicit information about which samples are useful to solve the task at hand. However, a different operating point on the trade-off between performance and number of processed samples may be required depending on the application, e.g. one may be willing to sacrifice a few accuracy points in order to run faster on machines with a low computational power, or to reduce energy impact on portable devices. The proposed model can be encouraged to perform fewer state updates through additional loss terms, a common practice in neural networks with dynamically allocated computation [95, 97, 98, 100]. In particular, we consider a *cost per sample* condition

$$L_{\text{budget}} = \lambda \cdot \sum_{t=1}^{T} u_t, \tag{5.8}$$

where L_{budget} is the cost associated to a single sequence, λ is the cost per sample and T is the sequence length. This formulation bears a similarity to weight decay regularization, where the network is encouraged to slowly converge towards a solution where the norm of the weights is small. Similarly, in this case the network is encouraged to converge toward a solution where fewer state updates are required.

Although the above budget formulation is extensively studied in our experiments, other budget loss terms can be used depending on the application. For instance, a specific number of samples may be encouraged by applying a L_1 or L_2 loss between the target value and the number of updates per sequence, $\sum_{t=1}^{T} u_t$.

5.3 Experiments

In the following section, we investigate the advantages of adding this state skipping capability to two common RNN architectures, LSTM and GRU, for a variety of tasks. In addition to the evaluation metric for each task, we report the number of RNN state updates (i.e., the number of elements in the input sequence used by the model) and the number of Floating Point Operations (FLOPs) as measures of the computational load for each model. Since skipping an RNN update results in ignoring its corresponding input, we will refer to the number of updates and the number of used samples (i.e. elements in a sequence) interchangeably. With the goal of studying the effect of skipping state updates on the learning capability of the networks, we also introduce a baseline which skips a state update with probability p_{skip}. We tune the skipping probability to obtain models that perform a similar number of state updates to the Skip RNN models.

Training is performed with Adam [43], learning rate of 10^{-4}, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$ on batches of 256. Gradient clipping [122] with a threshold of 1 is applied to all trainable variables. Bias b_u in Equation 5.4 is initialized to 1, so that all samples are used at the beginning of training[1]. The initial hidden state $\mathbf{s}_0$ is learned during training, whereas $\tilde{u}_0$ is set to a constant value of 1 in order to force the first update at $t = 1$.

Experiments are implemented with TensorFlow [69] and run on a single NVIDIA K80 GPU. Code is available at https://github.com/imatge-upc/skiprnn-2017-telecombcn.

5.3.1 Adding Task

We revisit one of the original LSTM tasks [38], where the network is given a sequence of *(value, marker)* tuples. The desired output is the addition of only two values that are marked with a 1, whereas those marked with a 0 need to be ignored. We follow the experimental setup by Neil et al. [91], where the first marker is randomly placed among the first 10% of samples (drawn with uniform probability) and the second one is placed among the last half of samples (drawn with uniform probability). This marker distribution yields sequences where at least 40% of the samples are distractors and provide no useful information at all. However, it is worth noting that in this task the risk of missing a marker is very large as compared to the benefits of working on shorter subsequences.

[1] In practice, forcing the network to use all samples at the beginning of training improves its robustness against random initializations of its weights and increases the reproducibility of the presented experiments. A similar behavior was observed in other augmented RNN architectures such as Neural Stacks [123].

Model	**Task solved**	**State updates**	**Inference FLOPs**
LSTM	Yes	$100.0\% \pm 0.0\%$	2.46×10^6
LSTM, $p_{\text{skip}} = 0.2$	No	$80.0\% \pm 0.1\%$	1.97×10^6
LSTM, $p_{\text{skip}} = 0.5$	No	$50.1\% \pm 0.1\%$	1.23×10^6
Skip LSTM, $\lambda = 0$	Yes	$81.1\% \pm 3.6\%$	2.00×10^6
Skip LSTM, $\lambda = 10^{-5}$	Yes	$53.9\% \pm 2.1\%$	1.33×10^6
GRU	Yes	$100.0\% \pm 0.0\%$	1.85×10^6
GRU, $p_{\text{skip}} = 0.02$	No	$98.0\% \pm 0.0\%$	1.81×10^6
GRU, $p_{\text{skip}} = 0.5$	No	$49.9\% \pm 0.6\%$	9.25×10^5
Skip GRU, $\lambda = 0$	Yes	$97.9\% \pm 3.2\%$	1.81×10^6
Skip GRU, $\lambda = 10^{-5}$	Yes	$50.7\% \pm 2.6\%$	9.40×10^5

Table 5.1: Results for the adding task, displayed as *mean* ± *std* over four different runs. We consider different values for the cost per sample, λ, in Equation 5.8. The task is considered to be solved if the MSE is at least two orders of magnitude below the variance of the output distribution.

We train RNN models with 110 units each on sequences of length 50, where the values are uniformly drawn from $\mathcal{U}(-0.5, 0.5)$. The final RNN state is fed to a fully connected layer that regresses the scalar output. The model is trained to minimize the Mean Squared Error (MSE) between the output and the ground truth. We consider that a model is able to solve the task when its MSE on a held-out set of examples is at least two orders of magnitude below the variance of the output distribution. This criterion is a stricter version of the one followed by Hochreiter and Schmidhuber [38].

While all models learn to solve the task, results in Table 5.1 show that Skip RNN models are able to do so with roughly half of the updates of their corresponding counterparts. We observed that the models using fewer updates never miss any marker, since the penalization in terms of MSE would be very large (see Appendix A.1 for examples). This is confirmed by the poor performance of the baselines that randomly skip state updates, which are not able to solve the tasks even when the skipping probability is low. Skip RNN models learn to skip most of the samples in the 40% of the sequence where there are no markers. Moreover, most updates are skipped once the second marker is found, since all the relevant information in the sequence has already been seen. This last pattern provides evidence that the proposed models effectively learn whether to update or copy the hidden state based on the input sequence, as opposed to learning biases in the dataset only. As a downside, Skip RNN models show some difficulties skipping a large number of updates at once, probably due to the cumulative nature of $\tilde{u}_t$.

5.3.2 Frequency Discrimination Task

In this experiment, the network is trained to classify between sinusoids whose period is in range $T \sim \mathcal{U}(5,6)$ milliseconds and those whose period is in range $T \sim \{(1,5) \cup (6,100)\}$ milliseconds [91]. Every sine wave with period T has a random phase shift drawn from $\mathcal{U}(0,T)$. At every time step, the input to the network is a single scalar representing the amplitude of the signal. Since sinusoids are continuous signals, this tasks allows to study whether Skip RNNs converge to the same solutions when their parameters are fixed but the sampling period is changed. We study two different sampling periods, $T_s = \{0.5, 1\}$ milliseconds, for each set of hyperparameters.

We train RNNs with 110 units each on input signals of 100 milliseconds. Batches are stratified, containing the same number of samples for each class, yielding a 50% chance accuracy. The last state of the RNN is fed into a 2-way classifier and trained with cross-entropy loss. We consider that a model is able to solve the task when it achieves an accuracy over 99% on a held-out set of examples.

Table 5.2 summarizes results for this task. When no cost per sample is set ($\lambda = 0$), the number of updates differ under different sampling conditions. We attribute this behavior to the potentially large number of local minima in the cost function, since there are numerous subsampling patterns for which the task can be successfully solved and we are not explicitly encouraging the network to converge to a particular solution. On the other hand, when $\lambda > 0$ Skip RNN models with the same cost per sample use roughly the same number of input samples even when the sampling frequency is doubled. This is a desirable property, since solutions are robust to oversampled input signals. Qualitative results can be found in Appendix A.2.

Model	$\mathbf{T_s = 1ms}$ **(length 100)**		$\mathbf{T_s = 0.5ms}$ **(length 200)**	
	Task solved	**State updates**	**Task solved**	**State updates**
LSTM	Yes	100.0 ± 0.00	Yes	200.0 ± 0.00
Skip LSTM, $\lambda = 0$	Yes	55.5 ± 16.9	Yes	147.9 ± 27.0
Skip LSTM, $\lambda = 10^{-5}$	Yes	47.4 ± 14.1	Yes	50.7 ± 16.8
Skip LSTM, $\lambda = 10^{-4}$	Yes	12.7 ± 0.5	Yes	19.9 ± 1.5
GRU	Yes	100.0 ± 0.00	Yes	200.0 ± 0.00
Skip GRU, $\lambda = 0$	Yes	73.7 ± 17.9	Yes	167.0 ± 18.3
Skip GRU, $\lambda = 10^{-5}$	Yes	51.9 ± 10.2	Yes	54.2 ± 4.4
Skip GRU, $\lambda = 10^{-4}$	Yes	23.5 ± 6.2	Yes	22.5 ± 2.1

Table 5.2: Results for the frequency discrimination task, displayed as *mean* $\pm$ *std* over four different runs. We consider different values for the cost per sample, λ, in Equation 5.8. The task is considered to be solved if the classification accuracy is over 99%. Models with the same cost per sample ($\lambda > 0$) converge to a similar number of used samples under different sampling conditions.

5.3.3 MNIST Classification from a Sequence of Pixels

The MNIST handwritten digits classification benchmark [53] is traditionally addressed with Convolutional Neural Networks (CNNs) that efficiently exploit spatial dependencies through weight sharing. By flattening the 28×28 images into 784-d vectors, however, it can be reformulated as a challenging task for RNNs where long term dependencies need to be leveraged [124]. We follow the standard data split and set aside 5,000 training samples for validation purposes. After processing all pixels with an RNN with 110 units, the last hidden state is fed into a linear classifier predicting the digit class. All models are trained for 600 epochs to minimize cross-entropy loss.

Table 5.3 summarizes classification results on the test set after 600 epochs of training. Skip RNNs are not only able to solve the task using fewer updates than their counterparts, but also show a lower variance across runs and train faster (see Figure 5.2). We hypothesize that skipping updates make the Skip RNNs work on shorter subsequences, simplifying the optimization process and allowing the networks to capture long term dependencies more easily. A similar behavior was observed for Phased LSTM, where increasing the sparsity of cell updates accelerates training for very long sequences [91]. However, the drop in performance observed in the models where the state updates are skipped randomly suggests that learning which samples to use is a key component in the performance of Skip RNN.

The performance of RNN models on this task can be boosted through techniques like recurrent batch normalization [116] or recurrent skip coefficients [125]. Cooijmans et al. [116] show how an LSTM with specific weight initialization schemes for improved gradient flow [124, 126] can reach accuracy rates of up to 0.989. Note that these techniques are orthogonal to skipping state updates and Skip RNN models could benefit from them as well.

Sequences of pixels can be reshaped back into 2D images, allowing to visualize the samples used by the RNNs as a sort of hard visual attention model [105]. Examples depicted in Figure 5.3 show how the model learns to skip pixels that are not discriminative, such as the padding regions in the top and bottom of images. Similarly to the qualitative results for the adding task (Section 5.3.1), attended samples vary depending on the particular input being given to the network.

5.3.4 Sentiment Analysis on IMDB

The IMDB dataset [127] contains 25,000 training and 25,000 testing movie reviews annotated into two classes, *positive* and *negative* sentiment, with an approximate average

Model	Accuracy	State updates	Inference FLOPs
LSTM	0.910 ± 0.045	784.00 ± 0.00	3.83×10^7
LSTM, $p_{\text{skip}} = 0.5$	0.893 ± 0.003	392.03 ± 0.05	1.91×10^7
Skip LSTM, $\lambda = 10^{-4}$	0.973 ± 0.002	379.38 ± 33.09	1.86×10^7
GRU	0.968 ± 0.013	784.00 ± 0.00	2.87×10^7
GRU, $p_{\text{skip}} = 0.5$	0.912 ± 0.004	391.86 ± 0.14	1.44×10^7
Skip GRU, $\lambda = 10^{-4}$	0.976 ± 0.003	392.62 ± 26.48	1.44×10^7
TANH-RNN [124]	0.350	784.00	–
iRNN [124]	0.970	784.00	–
uRNN [126]	0.951	784.00	–
sTANH-RNN [125]	0.981	784.00	–
LSTM [116]	0.989	784.00	–
BN-LSTM [116]	0.990	784.00	–

Table 5.3: Accuracy, used samples and average FLOPs per sequence at inference on the test set of MNIST after 600 epochs of training. Results are displayed as *mean* $\pm$ *std* over four different runs.

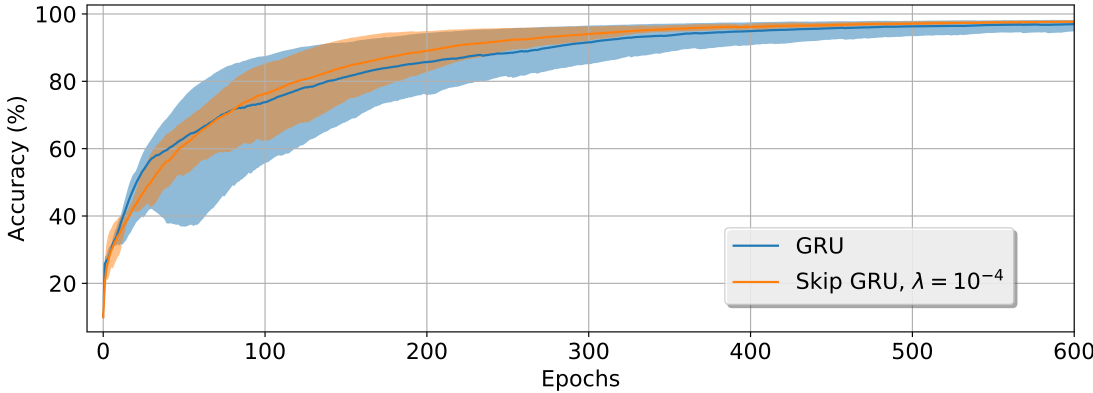


Figure 5.2: Accuracy evolution during training on the validation set of MNIST. The Skip GRU exhibits lower variance and faster convergence than the baseline GRU. A similar behavior is observed for LSTM and Skip LSTM, but omitted for clarity. Shading shows maximum and minimum over 4 runs, while dark lines indicate the mean.

length of 240 words per review. We set aside 15% of training data for validation purposes. Words are embedded into 300-d vector representations before being fed to an RNN with 128 units. The embedding matrix is initialized using pre-trained word2vec[2] embeddings [128] when available, or random vectors drawn from $\mathcal{U}(-0.25, 0.25)$ otherwise [32]. Dropout with rate 0.2 is applied between the last RNN state and the classification layer in order to reduce overfitting. We evaluate the models on sequences of length 200 and 400 by cropping longer sequences and padding shorter ones [114].

Results on the test are reported in Table 5.4. In a task where it is hard to predict which input tokens will be discriminative, the Skip RNN models are able to achieve similar

[2] `https://code.google.com/archive/p/word2vec/`

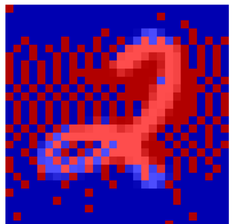
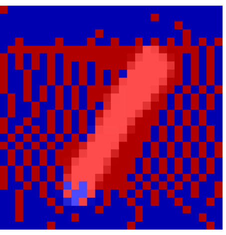
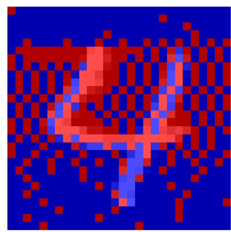
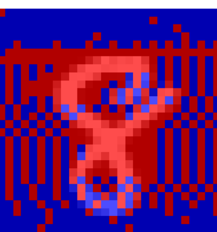





Figure 5.3: Sample usage examples for the Skip LSTM with $\lambda = 10^{-4}$ on the test set of MNIST. Red pixels are used, whereas blue ones are skipped.

accuracy rates to the baseline models while reducing the number of required updates. These results highlight the trade-off between accuracy and the available computational budget, since a larger cost per sample results in lower accuracies. However, allowing the network to select which samples to use instead of cropping sequences at a given length boosts performance, as observed for the Skip LSTM (length 400, $\lambda = 10^{-4}$), which achieves a higher accuracy than the baseline LSTM (length 200) while seeing roughly the same number of words per review. A similar behavior can be seen for the Skip RNN models with $\lambda = 10^{-3}$, where allowing them to select words from longer reviews boosts classification accuracy while using a comparable number of tokens per sequence.

In order to reduce overfitting of large models, Miyato et al. [129] leverage additional unlabeled data through adversarial training and achieve a state of the art accuracy of 0.941 on IMDB. For an extended analysis on how different experimental setups affect the performance of RNNs on this task, we refer the reader to Longpre et al. [130].

Model	**Length 200**		**Length 400**	
	Accuracy	**State updates**	**Accuracy**	**State updates**
LSTM	0.843 ± 0.003	200.00 ± 0.00	0.868 ± 0.004	400.00 ± 0.00
Skip LSTM, $\lambda = 0$	0.844 ± 0.004	196.75 ± 5.63	0.866 ± 0.004	369.70 ± 19.35
Skip LSTM, $\lambda = 10^{-5}$	0.846 ± 0.004	197.15 ± 3.16	0.865 ± 0.001	380.62 ± 18.20
Skip LSTM, $\lambda = 10^{-4}$	0.837 ± 0.006	164.65 ± 8.67	0.862 ± 0.003	186.30 ± 25.72
Skip LSTM, $\lambda = 10^{-3}$	0.811 ± 0.007	73.85 ± 1.90	0.836 ± 0.007	84.22 ± 1.98
GRU	0.845 ± 0.006	200.00 ± 0.00	0.862 ± 0.003	400.00 ± 0.00
Skip GRU, $\lambda = 0$	0.848 ± 0.002	200.00 ± 0.00	0.866 ± 0.002	399.02 ± 1.69
Skip GRU, $\lambda = 10^{-5}$	0.842 ± 0.005	199.25 ± 1.30	0.862 ± 0.008	398.00 ± 2.06
Skip GRU, $\lambda = 10^{-4}$	0.834 ± 0.006	180.97 ± 8.90	0.853 ± 0.011	314.30 ± 2.82
Skip GRU, $\lambda = 10^{-3}$	0.800 ± 0.007	106.15 ± 37.92	0.814 ± 0.005	99.12 ± 2.69

Table 5.4: Accuracy and used samples on the test set of IMDB for different sequence lengths. Results are displayed as *mean* $\pm$ *std* over four different runs. We consider different values for the cost per sample, λ, in Equation 5.8.

5.3.5 Action Classification on UCF-101

One popular approach to video analysis tasks is to extract frame-level features with a CNN and modeling temporal dynamics with an RNN [131, 132]. Videos are commonly recorded at high sampling rates, generating long sequences with strong temporal redundancies that are challenging for RNNs. Moreover, processing frames with a CNN is computationally expensive and may become prohibitive for high frame rates. These issues have been alleviated in previous works by using short clips [131] or by downsampling the original data in order to cover long temporal spans without increasing the sequence length excessively [132]. Instead of addressing the long sequence problem at the input data level, we let the network learn which frames need to be used.

UCF-101 [133] is a dataset containing 13,320 trimmed videos belonging to 101 different action categories. We use 10 seconds of video sampled at 25fps, cropping longer ones and padding shorter examples with empty frames. Activations in the Global Average Pooling layer from a ResNet-50 [63] CNN pretrained on the ImageNet dataset [5] are used as frame-level features, which are fed into two stacked RNN layers with 512 units each. The weights in the CNN are not tuned during training to reduce overfitting. The hidden state in the last RNN layer is used to compute the update probability for the Skip RNN models.

We evaluate the different models on the first split of UCF-101 and report results in Table 5.5. Skip RNN models do not only improve the classification accuracy with respect to the baseline, but require very few updates to do so, possibly due to the low motion between consecutive frames resulting in frame-level features with high temporal redundancy [134]. Moreover, Figure 5.4 shows how models performing fewer updates converge faster thanks to the gradients being preserved during longer spans when training with backpropagation through time.

Non-recurrent architectures for video action recognition that have achieved high performance on UCF-101 comprise CNNs with spatiotemporal kernels [135] or two-stream CNNs [136]. Carreira and Zisserman [137] show the benefits of expanding 2D CNN filters into 3D and pretraining on larger datasets, obtaining an accuracy of 0.845 when using RGB data only and 0.934 when incorporating optical flow information.

5.3.6 Temporal Action Localization on Charades

Charades [138] is a dataset containing 9,848 videos annotated for 157 action classes in a per-frame fashion. Frames are encoded using *fc7* features from the RGB stream of a

Model	Accuracy	State updates	Inference FLOPs
LSTM	0.671	250.0	9.52×10^{11}
Skip LSTM, $\lambda = 0$	0.749	138.9	5.29×10^{11}
Skip LSTM, $\lambda = 10^{-5}$	0.757	24.2	9.21×10^{10}
Skip LSTM, $\lambda = 10^{-4}$	0.790	7.6	2.89×10^{10}
GRU	0.791	250.0	9.51×10^{11}
Skip GRU, $\lambda = 0$	0.796	124.2	4.73×10^{11}
Skip GRU, $\lambda = 10^{-5}$	0.792	29.7	1.13×10^{11}
Skip GRU, $\lambda = 10^{-4}$	0.793	23.7	9.02×10^{10}
I3D (RGB) [137]	0.845	–	–
Two-stream I3D [137]	0.934	–	–

Table 5.5: Accuracy, used samples and average FLOPs per sequence at inference on the validation set of UCF-101 (split 1). We consider different values for the cost per sample, λ, in Equation 5.8.

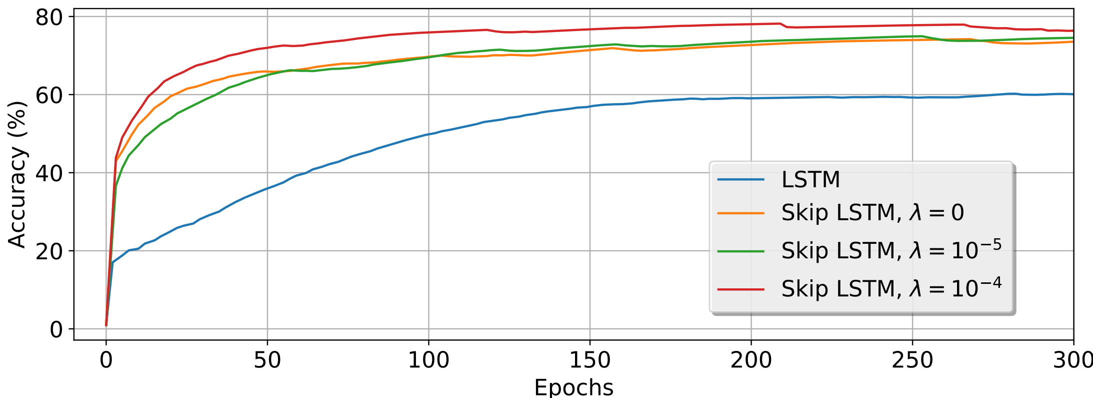


Figure 5.4: Accuracy evolution during the first 300 training epochs on the validation set of UCF-101 (split 1). Skip LSTM models converge much faster than the baseline LSTM.

Two-Stream CNN provided by the organizers of the challenge[3], extracted at 6fps. The encoded frames are fed into two stacked RNN layers with 256 units each and the hidden state in the last RNN layer is used to compute the update probability for the Skip RNN models. Since each frame may be annotated with zero or more classes, the networks are trained to minimize element-wise binary cross-entropy at every time step. Unlike the previous sequence tagging tasks, this setup allows us to evaluate the performance of Skip RNN on a task where the output is a sequence as well.

Evaluation is performed following the setup by Sigurdsson et al. [139], but evaluating on 100 equally spaced frames instead of 25, and results are reported in Table 5.6. It is surprising that the GRU baselines that randomly skip state updates perform on par with their Skip GRU counterparts for low skipping probabilities. We hypothesize several reasons for this behavior, which was not observed in previous experiments: (1) there is

[3] http://vuchallenge.org/charades.html

Model	**mAP (%)**	**State updates**	**Inference FLOPs**
LSTM	8.40	172.9 ± 47.4	2.65×10^{12}
LSTM, $p_{\text{skip}} = 0.75$	8.11	43.3 ± 13.2	6.63×10^{11}
LSTM, $p_{\text{skip}} = 0.90$	7.21	17.2 ± 6.1	2.65×10^{11}
Skip LSTM, $\lambda = 0$	8.32	172.9 ± 47.4	2.65×10^{12}
Skip LSTM, $\lambda = 10^{-4}$	8.61	172.9 ± 47.4	2.65×10^{12}
Skip LSTM, $\lambda = 10^{-3}$	8.32	41.9 ± 11.3	6.41×10^{11}
Skip LSTM, $\lambda = 10^{-2}$	7.86	17.4 ± 4.4	2.66×10^{11}
GRU	8.70	172.9 ± 47.4	2.65×10^{12}
GRU, $p_{\text{skip}} = 0.10$	8.94	155.6 ± 42.9	2.39×10^{12}
GRU, $p_{\text{skip}} = 0.40$	8.81	103.6 ± 29.3	1.06×10^{12}
GRU, $p_{\text{skip}} = 0.70$	8.42	51.9 ± 15.4	7.95×10^{11}
GRU, $p_{\text{skip}} = 0.90$	7.09	17.3 ± 6.3	2.65×10^{11}
Skip GRU, $\lambda = 0$	8.94	159.9 ± 46.9	2.45×10^{12}
Skip GRU, $\lambda = 10^{-4}$	8.76	100.8 ± 28.1	1.54×10^{12}
Skip GRU, $\lambda = 10^{-3}$	8.68	54.2 ± 16.2	8.29×10^{11}
Skip GRU, $\lambda = 10^{-2}$	7.95	18.4 ± 5.1	2.82×10^{11}

Table 5.6: Mean Average Precision (mAP), used samples and average FLOPs per sequence at inference on the validation set of Charades. We consider different values for the cost per sample, λ, in Equation 5.8. The number of state updates is displayed as *mean* $\pm$ *std* over all the videos in the validation set.

a supervision signal at every time step, (2) and the inputs and outputs are strongly correlated in consecutive frames. On the other hand, Skip RNN models clearly outperform the random methods when fewer updates are allowed. Note that this setup is far more challenging because of the longer time spans between updates, so properly distributing the state updates along the sequence is key to the performance of the models.

Skip GRU tends to perform fewer state updates than Skip LSTM when the cost per sample is low or none. This behavior is the opposite of the one observed in the adding task (Section 5.3.1), which may be related to the observation that determining the best performing gated unit depends on the task at hand [140]. Indeed, GRU models consistently outperform LSTM ones on this task. This mismatch in the number of used samples is not observed for large values of λ, as both Skip LSTM and Skip GRU converge to a comparable number of used samples.

A previous work reports better action localization performance by integrating RGB and optical flow information as an input to an LSTM, reaching 9.60% mAP [139]. This boost in performance comes at the cost of roughly doubling the number of FLOPs and memory footprint of the CNN encoder, plus requiring the extraction of flow information during a preprocessing step. Interestingly, our model learns which frames need to be attended from RGB data and without having access to explicit motion information.

5.4 Discussion

We presented Skip RNN as an extension to existing recurrent architectures enabling them to skip state updates thereby reducing the number of sequential operations in the computation graph. Unlike other approaches, all parameters in Skip RNN are trained with backpropagation. Experiments conducted with LSTMs and GRUs showed that Skip RNNs can match or in some cases even outperform the baseline models while relaxing their computational requirements. Skip RNNs provide faster and more stable training for long sequences and complex models, owing to gradients being backpropagated through fewer time steps resulting in a simpler optimization task. Moreover, the introduced computational savings are better suited for modern hardware than those methods that reduce the amount of computation required at each time step [91, 99, 101].

The presented results motivate several new research directions toward designing efficient RNN architectures. Introducing stochasticity in neural network training has proven beneficial for generalization [102, 141], which could be achieved by replacing the deterministic rounding operation with stochastic sampling. We showed that the addition of a loss term penalizing the number of updates is important in the performance of Skip RNN and allows flexibility to specialize to tasks of varying budget requirements, e.g. the cost can be increased at each time step to encourage the network to emit a decision earlier [142], or the number of updates can be strictly bounded and enforced. Finally, understanding and analyzing the patterns followed by the model when deciding whether to update or copy the RNN state may provide insight for developing more efficient architectures.

6

Robust Initialization for WeightNorm & ResNets

Devansh Arpit*, Víctor Campos*, and Yoshua Bengio. How to initialize your network? Robust initialization for WeightNorm & ResNets. In *NeurIPS*, 2019

Parameter initialization is an important aspect of deep network optimization and plays a crucial role in determining the quality of the final model. In order for deep networks to learn successfully using gradient descent based methods, information must flow smoothly in both forward and backward directions [60, 63, 143, 144]. Too large or too small parameter scale leads to information exploding or vanishing across hidden layers in both directions. This could lead to loss being stuck at initialization or quickly diverging at the beginning of training. Beyond these characteristics near the point of initialization itself, we argue that the choice of initialization also has an impact on the final generalization performance. This non-trivial relationship between initialization and final performance emerges because *good* initializations allow the use of *larger* learning rates which have been shown in existing literature to correlate with better generalization [145–147].

Weight normalization [62] accelerates convergence of Stochastic Gradient Descent (SGD) optimization by re-parameterizing weight vectors in neural networks. However, previous works have not studied initialization strategies for weight normalization and it is a common practice to use initialization schemes designed for un-normalized networks as a proxy. We study initialization conditions for weight normalized networks with Rectified

*Equal contribution

Linear Unit (ReLU) non-linearities [148], and propose a new initialization strategy for both plain and residual architectures.

The main contribution of this chapter is the theoretical derivation of a novel initialization strategy for weight normalized ReLU networks, with and without residual connections, that prevents information flow from exploding/vanishing in forward and backward pass. Extensive experimental evaluation shows that the proposed initialization increases robustness to network depth, choice of hyperparameters and seed. When combining the proposed initialization with learning rate warmup, we are able to use learning rates as large as the ones used with batch normalization [61] and significantly reduce the generalization gap between weight and batch normalized networks reported in the literature [149, 150]. Further analysis reveals that our proposal initializes networks in regions of the parameter space that have low curvature, thus allowing the use of large learning rates which are known to correlate with better generalization [145–147].

6.1 Related Work

Weight Normalization. Previous works have considered re-parameterizations that normalize weights in neural networks as means to accelerate convergence. In Arpit et al. [151], the pre- and post-activations are scaled/summed with constants depending on the activation function, ensuring that the hidden activations have zero mean and unit variance, especially at initialization. However, their work makes assumptions on the distribution of input and pre-activations of the hidden layers in order to make these guarantees. Weight normalization [62] is a simpler alternative, and the authors propose to use a data-dependent initialization [152, 153] for the introduced re-parameterization. This operation improves the flow of information, but its dependence on statistics computed from a batch of data may make it sensitive to the samples used to estimate the initial values.

Residual Network Architectures. Residual Networks (ResNets) [63] have become a cornerstone of deep learning due to their state-of-the-art performance in various applications. However, when using residual networks with weight normalization instead of batch normalization [61], they have been shown to have significantly worse generalization performance. For instance, Gitman and Ginsburg [149] and Shang et al. [150] have shown that ResNets with weight normalization suffer from severe over-fitting and have concluded that batch normalization has an implicit regularization effect.

Initialization strategies. There exists extensive literature on initialization schemes for un-normalized plain networks (c.f. He et al. [7], Glorot and Bengio [60], Saxe et al.

[154], Poole et al. [155], Pennington et al. [156, 157], to name some of the most prominent ones). Similarly, previous works have studied initialization strategies for un-normalized ResNets [143, 158, 159], but they lack large scale experiments demonstrating the effectiveness of the proposed approaches and consider a simplified ResNet setup where shortcut connections are ignored, even though they play an important role [160]. Zhang et al. [161] propose an initialization scheme for un-normalized ResNets which involves initializing the different types of layers individually using carefully designed schemes. They provide large scale experiments on various datasets, and show that the generalization gap between batch normalized ResNets and un-normalized ResNets can be reduced when using their initialization along with additional domain-specific regularization techniques like cutout [162] and mixup [163]. All the aforementioned works consider un-normalized networks and, to the best of our knowledge, there has been no formal analysis of initialization strategies for weight normalized networks that allow a smooth flow of information in the forward and backward pass.

6.2 Weight Normalized ReLU Networks

We derive initialization schemes for weight normalized networks with ReLU activation function in the asymptotic setting where network width tends to infinity, similarly to previous analysis for un-normalized networks [60, 63]. We define an L layer weight normalized ReLU network $f_\theta(\mathbf{x}) = \mathbf{h}^L$ recursively, where the lth hidden layer's activation is given by:

$$\begin{aligned} \mathbf{h}^l &:= \text{ReLU}(\mathbf{a}^l) \\ \mathbf{a}^l &:= \mathbf{g}^l \odot \hat{\mathbf{W}}^l \mathbf{h}^{l-1} + \mathbf{b}^l \quad l \in \{1, 2, \cdots L\} \end{aligned} \tag{6.1}$$

where $\mathbf{a}^l$ are the pre-activations, $\mathbf{h}^l \in \mathbb{R}^{n_l}$ are the hidden activations, $\mathbf{h}^o = \mathbf{x}$ is the input to the network, $\mathbf{W}^l \in \mathbb{R}^{n_l \times n_{l-1}}$ are the weight matrices, $\mathbf{b} \in \mathbb{R}^{n_l}$ are the bias vectors, and $\mathbf{g}^l \in \mathbb{R}^{n_l}$ is a scale factor. We denote the set of all learnable parameters as $\theta = \{(\mathbf{W}^l, \mathbf{g}^l, \mathbf{b}^l)\}_{l=1}^L$. Notation $\hat{\mathbf{W}}^l$ implies that each row vector of $\hat{\mathbf{W}}^l$ has unit norm:

$$\hat{\mathbf{W}}_i^l = \frac{\mathbf{W}_i^l}{\|\mathbf{W}_i^l\|_2} \quad \forall i \tag{6.2}$$

thus $\mathbf{g}_i^l$ controls the norm of each weight vector, whereas $\hat{\mathbf{W}}_i^l$ controls its direction. Finally, we will make use of the notion $\mathcal{L}(f_\theta(\mathbf{x}), \mathbf{y})$ to represent a differentiable loss function over the output of the network.

6.2.1 Forward Pass

We first study the forward pass and derive an initialization scheme such that for any given input, the norm of the hidden activation of any layer and the input norm are asymptotically equal. Failure to do so prevents training to begin, as studied by Hanin and Rolnick [143] for vanilla deep feedforward networks. The theorem below shows that a normalized linear transformation followed by ReLU non-linearity is a norm preserving transform in expectation when proper scaling is used.

Theorem 1. *Let* $\mathbf{v} = \text{ReLU}\left(\sqrt{2n/m} \cdot \hat{\mathbf{R}}\mathbf{u}\right)$*, where* $\mathbf{u} \in \mathbb{R}^n$ *and* $\hat{\mathbf{R}} \in \mathbb{R}^{m \times n}$*. If* $\mathbf{R}_i \overset{i.i.d.}{\sim} P$ *where* P *is any isotropic distribution in* $\mathbb{R}^n$*, or alternatively* $\hat{\mathbf{R}}$ *is a randomly generated matrix with orthogonal rows, then for any fixed vector* $\mathbf{u}$*,* $\mathbb{E}[\|\mathbf{v}\|^2] = K_n \cdot \|\mathbf{u}\|^2$*, where*

$$K_n = \begin{cases} \frac{2S_{n-1}}{S_n} \cdot \left(\frac{2}{3} \cdot \frac{4}{5} \cdots \frac{n-2}{n-1}\right) & \text{if } n \text{ is even} \\ \frac{2S_{n-1}}{S_n} \cdot \left(\frac{1}{2} \cdot \frac{3}{4} \cdots \frac{n-2}{n-1}\right) \cdot \frac{\pi}{2} & \text{otherwise} \end{cases} \tag{6.3}$$

and S_n *is the surface area of a unit* n*-dimensional sphere.*

The constant K_n seems hard to evaluate analytically, but remarkably, we empirically find that $K_n = 1$ for all integers $n > 1$. Thus applying the above theorem to Eq. 6.1 implies that every hidden layer in a weight normalized ReLU network is norm preserving for an infinitely wide network if the elements of $\mathbf{g}^l$ are initialized with $\sqrt{2n_{l-1}/n_l}$. Therefore, we can recursively apply the above argument to each layer in a normalized deep ReLU network starting from the input to the last layer and have that the network output norm is approximately equal to the input norm, i.e. $\|f_\theta(\mathbf{x})\| \approx \|\mathbf{x}\|$. Figure 6.1 (top left) shows a synthetic experiment with a 20 layer weight normalized Multilayer Perceptron (MLP) that empirically confirms the above theory. Details for this experiment can be found in Appendix C.1.1.

6.2.2 Backward Pass

The goal of studying the backward pass is to derive conditions for which gradients do not explode nor vanish, which is essential for gradient descent based training. Therefore, we are interested in the value of $\|\frac{\partial \mathcal{L}(f_\theta(\mathbf{x}),\mathbf{y})}{\partial \mathbf{a}^l}\|$ for different layers, which are indexed by l. To prevent exploding/vanishing gradients, the value of this term should be similar for all layers. We begin by writing the recursive relation between the value of this derivative

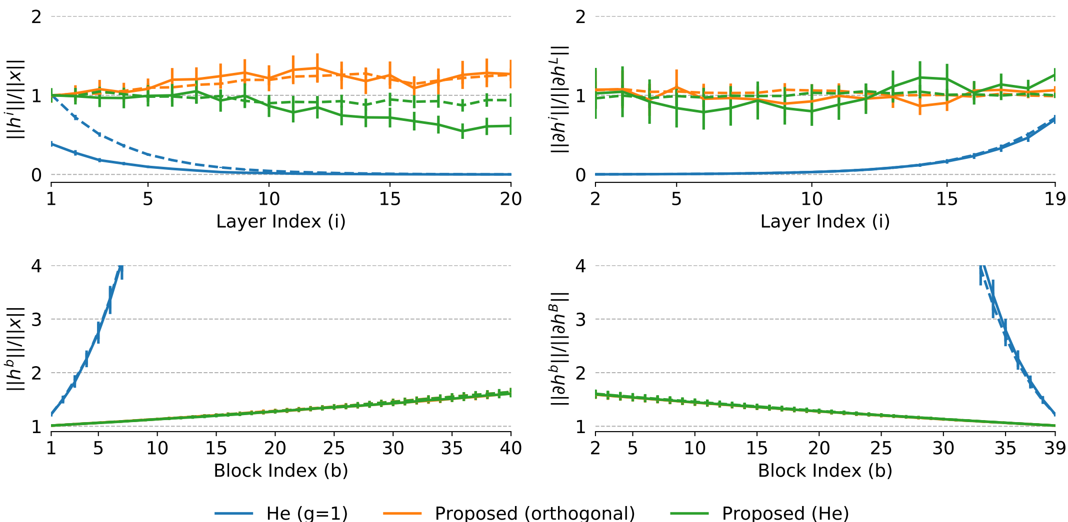


Figure 6.1: Experiments on weight normalized networks using synthetic data to confirm theoretical predictions. **Top:** feed forward networks. **Bottom:** residual networks. We report results for networks of width $\sim \mathcal{U}(150, 250)$ (solid lines) and width $\sim \mathcal{U}(950, 1050)$ (dashed lines). The proposed initialization prevents explosion/vanishing of the norm of hidden activations (left) and gradients (right) across layers at initialization. For ResNets, norm growth is $\mathcal{O}(1)$ for an arbitrary depth network. Naively initializing $\mathbf{g} = \mathbf{1}$ results in vanishing/exploding signals.

for consecutive layers:

$$\frac{\partial \mathcal{L}(f_\theta(\mathbf{x}), \mathbf{y})}{\partial \mathbf{a}^l} = \frac{\partial \mathbf{a}^{l+1}}{\partial \mathbf{a}^l} \cdot \frac{\partial \mathcal{L}(f_\theta(\mathbf{x}), \mathbf{y})}{\partial \mathbf{a}^{l+1}} \tag{6.4}$$

$$= \mathbf{g}^{l+1} \odot \mathbb{1}(\mathbf{a}^l) \odot \left(\hat{\mathbf{W}}^{l+1^T} \frac{\partial \mathcal{L}(f_\theta(\mathbf{x}), \mathbf{y})}{\partial \mathbf{a}^{l+1}} \right) \tag{6.5}$$

We note that conditioned on a fixed $\mathbf{h}^{l-1}$, each dimension of $\mathbb{1}(\mathbf{a}^l)$ in the above equation follows an i.i.d. sampling from Bernoulli distribution with probability 0.5 at initialization. This is formalized in Lemma 1 in Appendix D. We now consider the following theorem,

Theorem 2. *Let $\mathbf{v} = \sqrt{2} \cdot \mathbf{z} \odot \left(\hat{\mathbf{R}}^T \mathbf{u}\right)$, where $\mathbf{u} \in \mathbb{R}^m$, $\mathbf{R} \in \mathbb{R}^{m \times n}$ and $\mathbf{z} \in \mathbb{R}^n$. If each $\mathbf{R}_i \overset{i.i.d.}{\sim} P$ where P is any isotropic distribution in $\mathbb{R}^n$ or alternatively $\hat{\mathbf{R}}$ is a randomly generated matrix with orthogonal rows and $\mathbf{z}_i \overset{i.i.d.}{\sim}$ Bernoulli(0.5), then for any fixed vector $\mathbf{u}$, $\mathbb{E}[\|\mathbf{v}\|^2] = \|\mathbf{u}\|^2$.*

In order to apply the above theorem to Eq. 6.5, we assume that $\mathbf{u} := \frac{\partial \mathcal{L}(f_\theta(\mathbf{x}),\mathbf{y})}{\partial \mathbf{a}^{l+1}}$ is independent of the other terms, similar to He et al. [63]. This simplifies the analysis by allowing us to treat $\frac{\partial \mathcal{L}(f_\theta(\mathbf{x}),\mathbf{y})}{\partial \mathbf{a}^{l+1}}$ as fixed and take expectation w.r.t. the other terms, over $\mathbf{W}^l$ and $\mathbf{W}^{l+1}$. Thus $\|\frac{\partial \mathcal{L}(f_\theta(\mathbf{x}),\mathbf{y})}{\partial \mathbf{a}^l}\| = \|\frac{\partial \mathcal{L}(f_\theta(\mathbf{x}),\mathbf{y})}{\partial \mathbf{a}^{l+1}}\| \; \forall l$ if we initialize $\mathbf{g}^l = \sqrt{2} \cdot \mathbf{1}$. This also shows that $\frac{\partial \mathbf{a}^{l+1}}{\partial \mathbf{a}^l}$ is a norm preserving transform. Hence applying this theorem recursively to Eq. 6.5 for all l yields that $\|\frac{\partial \mathcal{L}(f_\theta(\mathbf{x}),\mathbf{y})}{\partial \mathbf{a}^l}\| \approx \frac{\partial \mathcal{L}(f_\theta(\mathbf{x}),\mathbf{y})}{\partial \mathbf{a}_L} \; \forall l$ thereby avoiding gradient explosion/vanishment. Note that the above result is strictly better for orthogonal weight

matrices compared with other isotropic distributions (see proof). Figure 6.1 (top right) shows a synthetic experiment with a 20 layer weight normalized MLP to confirm the above theory. The details for this experiment are provided in Appendix C.1.1.

We also point out that the $\sqrt{2}$ factor that appears in theorems 1 and 2 is due to the presence of ReLU activation. In the absence of ReLU, this factor should be 1. We will use this fact in the next section with the ResNet architecture.

6.2.3 Implementation Details

There is a discrepancy between the initialization required by the forward and backward pass. We tested both, as well as combinations of them, in our preliminary experiments. We found the one proposed for the forward pass to be superior, and therefore propose to initialize weight matrices $\mathbf{W}^l$ to be orthogonal[1], $\mathbf{b}^l = \mathbf{0}$, and $\mathbf{g}^l = \sqrt{2n_{l-1}/n_l} \cdot \mathbf{1}$, where n_{l-1} and n_l represent the fan-in and fan-out of the lth layer respectively. Our results apply to both fully-connected and convolutional[2] networks.

6.3 Residual Networks

Similar to the previous section, we derive an initialization strategy for ResNets in the infinite width setting. We define a residual network $\mathcal{R}(\{F_b(\cdot)\}_{b=0}^{B-1}, \theta, \alpha)$ with B residual blocks and parameters θ whose output is denoted as $f_\theta(\cdot) = \mathbf{h}^B$, and the hidden states are defined recursively:

$$\mathbf{h}^{b+1} := \mathbf{h}^b + \alpha F_b(\mathbf{h}^b) \quad b \in \{0, 1, \ldots, B-1\} \tag{6.6}$$

where $\mathbf{h}^0 = \mathbf{x}$ is the input, $\mathbf{h}^b$ denotes the hidden representation after applying b residual blocks and α is a scalar that scales the output of the b-th residual blocks. The b-th $\in \{0, 1, \ldots, B-1\}$ residual block $F_b(\cdot)$ is a feed-forward ReLU network. We discuss how to deal with shortcut connections during initialization separately. We use the notation $<\cdot, \cdot>$ to denote dot product between the argument vectors.

---

[1] We note that Saxe et al. [154] propose to initialize weights of un-normalized deep ReLU networks to be orthogonal with scale $\sqrt{2}$. Our derivation and proposal is for weight normalized ReLU networks where we study both Gaussian and orthogonal initialization and show the latter is superior.

[2] For convolutional layers with kernel size k and c channels, we define $n_{l-1} = k^2 c_{l-1}$ and $n_l = k^2 c_l$ [7].

6.3.1 Forward Pass

Here we derive an initialization strategy for residual networks that prevents information in the forward pass from exploding/vanishing independent of the number of residual blocks, assuming that each residual block is initialized such that it preserves information in the forward pass.

Theorem 3. *Let* $\mathcal{R}(\{F_b(\cdot)\}_{b=0}^{B-1}, \theta, \alpha)$ *be a residual network with output* $f_\theta(\cdot)$. *Assume that each residual block* $F_b(\cdot)$ $(\forall b)$ *is designed such that at initialization,* $\|F_b(\mathbf{h})\| = \|\mathbf{h}\|$ *for any input* $\mathbf{h}$ *to the residual block, and* $<\mathbf{u}, F_b(\mathbf{u})> \approx 0$. *If we set* $\alpha = 1/\sqrt{B}$, *then* $\exists c \in [\sqrt{2}, \sqrt{e}]$, *such that*

$$\|f_\theta(\mathbf{x})\| \approx c \cdot \|\mathbf{x}\| \tag{6.7}$$

The assumption $<\mathbf{u}, F_b(\mathbf{u})> \approx 0$ is reasonable because at initialization, $F_b(\mathbf{u})$ is a random transformation in a high dimensional space which will likely rotate a vector to be orthogonal to itself. To understand the rationale behind the second assumption, $\|F_b(\mathbf{h})\| = \|\mathbf{h}\|$, recall that $F_b(\cdot)$ is essentially a non-residual network. Therefore we can initialize each such block using the scheme developed in Section 6.2 which due to Theorem 1 (see discussion below it) will guarantee that the norm of the output of $F_b(\cdot)$ equals the norm of the input to the block. Figure 6.1 (bottom left) shows a synthetic experiment with a 40 block weight normalized ResNet to confirm the above theory. The ratio of norms of output to input lies in $[\sqrt{2}, \sqrt{e}]$ independent of the number of residual blocks exactly as predicted by the theory. The details for this experiment can be found in Appendix C.1.1.

6.3.2 Backward Pass

We now study the backward pass for residual networks.

Theorem 4. *Let* $\mathcal{R}(\{F_b(\cdot)\}_{b=0}^{B-1}, \theta, \alpha)$ *be a residual network with output* $f_\theta(\cdot)$. *Assume that each residual block* $F_b(\cdot)$ $(\forall b)$ *is designed such that at initialization,* $\|\frac{\partial F_b(\mathbf{h}^b)}{\partial \mathbf{h}^b}\mathbf{u}\| = \|\mathbf{u}\|$ *for any fixed input* $\mathbf{u}$ *of appropriate dimensions, and* $<\frac{\partial \mathcal{L}}{\partial \mathbf{h}^b}, \frac{\partial \mathbf{F}_{b-1}}{\partial \mathbf{h}^{b-1}} \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{h}_b}> \approx 0$. *If* $\alpha = \frac{1}{\sqrt{B}}$, *then* $\exists c \in [\sqrt{2}, \sqrt{e}]$, *such that*

$$\|\frac{\partial \mathcal{L}}{\partial \mathbf{h}^1}\| \approx c \cdot \|\frac{\partial \mathcal{L}}{\partial \mathbf{h}^B}\| \tag{6.8}$$

The above theorem shows that scaling the output of the residual block with $1/\sqrt{B}$ prevents explosion/vanishing of gradients irrespective of the number of residual blocks.

The rationale behind the assumptions is similar to that given for the forward pass above. Figure 6.1 (bottom right) shows a synthetic experiment with a 40 block weight normalized ResNet to confirm the above theory. Once again, the ratio of norms of gradient w.r.t. input to output lies in $[\sqrt{2}, \sqrt{e}]$ independent of the number of residual blocks exactly as predicted by the theory. The details can be found in Appendix C.1.1.

6.3.3 Implementation Details

A ResNet often has K stages [63], where each stage is characterized by one shortcut connection and B_k residual blocks, leading to a total of $\sum_{k=1}^{K} B_k$ blocks. In order to account for shortcut connections, we need to ensure that the input and output of each stage in a ResNet are at the same scale; the same argument applies during the backward pass. To achieve this, we scale the output of the residual blocks in each stage using the total number of residual blocks in that stage. Then Theorems 3 and 4 treat each stage of the network as a ResNet and normalize the flow of information in both directions to be independent of the number of residual blocks.

We consider ResNets with shortcut connections and architecture design similar to that proposed by He et al. [63] with the exception that our residual block structure is Conv $\rightarrow$ ReLU $\rightarrow$ Conv, similar to $B(3,3)$ blocks in Wide Residual Networks (WRNs) [164]. More generally, our residual block design principle is $D \times$ [Conv $\rightarrow$ ReLU $\rightarrow$]Conv, where $D \in \mathbb{Z}$. We refer the reader to the Appendix C.1.2 for a more detailed description of the architecture. Weights of all layers in the network are initialized to be orthogonal and biases are set to zero. The gain parameter of weight normalization is initialized to be $\mathbf{g} = \sqrt{\gamma \cdot \text{fan-in}/\text{fan-out}} \cdot \mathbf{1}$. We set $\gamma = 1/B_k$ for the last convolutional layer of each residual block in the k-th stage (i.e., α in Equation 6.6 is absorbed into the gain parameter $\mathbf{g}$). For the rest of layers we follow the strategy derived in Section 6.2, with $\gamma = 2$ when the layer is followed by ReLU, and $\gamma = 1$ otherwise.

6.4 Experiments

We study the impact of initialization on weight normalized networks across a wide variety of configurations. Among others, we compare against the data-dependent initialization proposed by Salimans and Kingma [62], which initializes $\mathbf{g}$ and $\mathbf{b}$ so that all pre-activations in the network have zero mean and unit variance based on estimates collected from a single minibatch of data.

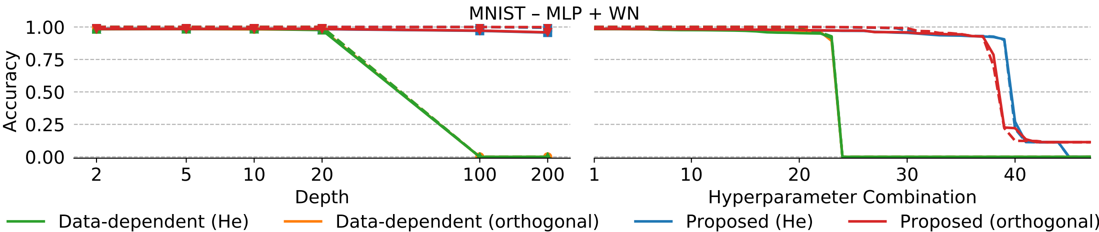


Figure 6.2: Results for MLPs on MNIST. Dashed lines denote train accuracy, and solid lines denote test accuracy. The accuracy of diverged runs is set to 0. **Left:** Accuracy as a function of depth. A held-out validation set is used to select the best model for each configuration. **Right:** Accuracy for each job in our hyperparameter sweep, depicting robustness to hyperparameter configurations.

Experiments are implemented with PyTorch [71]. Code is publicly available at `https://github.com/victorcampos7/weightnorm-init`. We refer the reader to Appendix C.1 for detailed description of the hyperparameter settings for each experiment.

6.4.1 Robustness Analysis

The difficulty of training due to exploding and vanishing gradients increases with network depth. In practice, depth often complicates the search for hyperparameters that enable successful optimization, if any. This section presents a thorough evaluation of the impact of initialization on different network architectures for increasing depths, as well as their robustness to hyperparameter configurations. We benchmark fully-connected networks on MNIST [165], whereas CIFAR-10 [166] is considered for convolutional and residual networks. We tune hyperparameters individually for each network depth and initialization strategy on a set of held-out examples, and report results on the test set. We refer the reader to Appendix C.1 for a detailed description of the considered hyperparameters.

Fully-connected networks. Results in Figure 6.2 (left) show that the data-dependent initialization can be used to train networks of up to depth 20, but training diverges for deeper nets even when using very small learning rates, e.g. 10^{-5}. On the other hand, we managed to successfully train very deep networks with up to 200 layers using the proposed initialization. When analyzing all runs in the grid search, we observe that the proposed initialization is more robust to the particular choice of hyperparameters (Figure 6.2, right). In particular, the proposed initialization allows using learning rates up to 10$\times$ larger for most depths.

Convolutional networks. We adopt a similar architecture to that in Xiao et al. [31], where all layers have 3×3 kernels and a fixed width. The two first layers use a stride of 2 in order to reduce the memory footprint. Results are depicted in Figure 6.3 (left) and show a similar trend to that observed for fully-connected nets, with the data-dependent initialization failing at optimizing very deep networks.

Residual networks. We construct residual networks of varying depths by controlling the number of residual blocks per stage in the WRN architecture with $k = 1$. Training networks with thousands of layers is computationally intensive, so we measure the test accuracy after a single epoch of training [161]. We consider two additional baselines for these experiments: (1) the default initialization in PyTorch[3], which initializes $g_i = \|\mathbf{W}_i\|_2$, and (2) a modification of the initialization proposed by Hanin and Rolnick [143] to fairly adapt it to weight normalized multi-stage ResNets. For the k-th stage with B_k blocks, the stage-wise Hanin scheme initializes the gain of the last convolutional layer in each block as $\mathbf{g} = 0.9^b \mathbf{1}$, where $b \in \{1, \ldots, B_k\}$ refers to the block number within a stage. All other parameters are initialized in a way identical to our proposal, so that information across the layers within residual blocks remains preserved. We report results over 5 random seeds for each configuration in Figure 6.3 (right), which shows that the proposed initialization achieves similar accuracy rates across the wide range of evaluated depths. PyTorch's default initialization diverges for most depths, and the data-dependent baseline converges significantly slower for deeper networks due to the small learning rates used in order to avoid divergence. Despite the stage-wise Hanin strategy and the proposed initialization achieve similar accuracy rates, we were able to use an order of magnitude larger learning rates with the latter, which denotes an increased robustness against hyperparameter configurations.

To further evaluate the robustness of each initialization strategy, we train WRN-40-10 networks for 3 epochs with different learning rates, with and without learning rate warmup [59]. We repeat each experiment 20 times using different random seeds, and report the percentage of runs that successfully completed all 3 epochs without diverging in Figure 6.4. We observed that learning rate warmup greatly improved the range of learning rates that work well for all initializations, but the proposed strategy manages to train more robustly across all tested configurations.

6.4.2 Comparison with Batch Normalization

Existing literature has pointed towards an implicit regularization effect of batch normalization [167], which prevented weight normalized models from matching the final performance of batch normalized ones [149]. On the other hand, previous works have shown that larger learning rates facilitate finding wider minima which correlate with better generalization performance [89, 145–147], and the proposed initialization and learning rate warmup have proven very effective in stabilizing training for high learning rates. This section aims at evaluating the final performance of weight normalized networks trained with high learning rates, and compare them with batch normalized networks.

[3] https://pytorch.org/docs/stable/_modules/torch/nn/utils/weight_norm.html

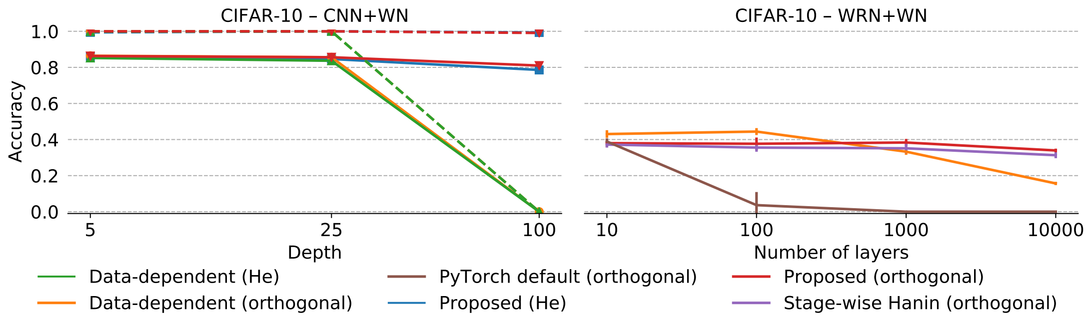


Figure 6.3: Accuracy as a function of depth on CIFAR-10 for CNNs (**left**), and WRNs (**right**). Dashed lines denote train accuracy, and solid lines denote validation accuracy. Note that WRNs are trained for a single epoch due to the computational burden of training extremely deep networks.

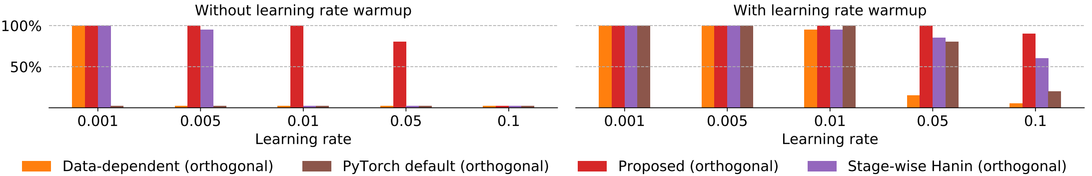


Figure 6.4: Robustness to seed of different initialization schemes on WRN-40-10. We launch 20 training runs for every configuration, and measure the percentage of runs that reach epoch 3 without diverging. Weight normalized ResNets benefit from learning rate warmup, which enables the usage of higher learning rates. The proposed initialization is the most robust scheme across all configurations.

We evaluate models on CIFAR-10 and CIFAR-100. We set aside 10% of the training data for hyperparameter tuning, whereas some previous works use the test set for such purpose [63, 164]. This difference in the experimental setup explains why the achieved error rates are slightly larger than those reported in the literature. For each architecture we use the default hyperparameters for batch normalized networks, and tune only the initial learning rate for weight normalized models.

Results in Table 6.1 show that the proposed initialization scheme, when combined with learning rate warmup, allows weight normalized residual networks to achieve comparable error rates to their batch normalized counterparts. We note that previous works reported a large generalization gap between weight and batch normalized networks [149, 150]. The only architecture for which the batch normalized variant achieves a superior performance is WRN-40-10, for which the weight normalized version is not able to completely fit the training set before reaching the epoch limit. This phenomena is different to the generalization gap reported in previous works, and might be caused by sub-optimal learning rate schedules that were tailored for networks with batch normalization.

Dataset	Architecture	Method	Test Error (%)
CIFAR-10	ResNet-56	WN w/ datadep init	9.19 ± 0.24
		WN w/ proposed init	7.87 ± 0.14
		WN w/ proposed init + warmup	7.20 ± 0.12
		BN (He et al. [63])	6.97
	ResNet-110	WN w/ datadep init	9.33 ± 0.10
		WN w/ proposed init	7.71 ± 0.14
		WN w/ proposed init + warmup	6.69 ± 0.11
		WN (Shang et al. [150])	7.46
		BN (He et al. [63])	6.61 ± 0.16
	WRN-40-10	WN w/ datadep init + cutout	6.10 ± 0.23
		WN w/ proposed init + cutout	4.74 ± 0.14
		WN w/ proposed init + cutout + warmup	4.75 ± 0.08
		BN w/ orthogonal init + cutout	3.53 ± 0.38
CIFAR-100	ResNet-164	WN w/ datadep init + cutout	30.26 ± 0.51
		WN w/ proposed init + cutout	27.30 ± 0.49
		WN w/ proposed init + cutout + warmup	25.31 ± 0.26
		BN w/ orthogonal init + cutout	25.52 ± 0.17

Table 6.1: Comparison between Weight Normalization (WN) with proposed initialization and Batch Normalization (BN). Results are reported as *mean* ± *std* over 5 runs.

6.4.3 Initialization Method and Generalization Gap

The motivation behind designing good parameter initialization is mainly for better optimization at the beginning of training, and it is not apparent why our initialization is able to reduce the generalization gap between weight normalized and batch normalized networks [149, 150]. On this note we point out that a number of papers have shown how using SGD with larger learning rates facilitate finding wider minima which correlate with better generalization performance [89, 145–147]. Additionally, it is often not possible to use large learning rates with weight normalization with traditional initializations. Therefore we believe that the use of large learning rate allowed by our initialization played an important role in this aspect. In order to understand why our initialization allows using large learning rates compared with existing ones, we compute the (log) spectral norm of the Hessian at initialization (using Power method) for the various initialization methods considered in our experiments using 10% of the training samples. They are shown in Table 6.2. We find that the local curvature (spectral norm) is smallest for the proposed initialization. These results are complementary to the seed robustness experiment shown in Figure 6.4.

Dataset	Model	PyTorch default	Data-dependent	Stage-wise Hanin	Proposed
CIFAR-10	WRN-40-10	4.68 ± 0.60	3.01 ± 0.02	7.14 ± 0.72	1.31 ± 0.12
CIFAR-100	ResNet-164	9.56 ± 0.54	2.68 ± 0.09	N/A	1.56 ± 0.18

Table 6.2: Log (base 10) spectral norm of Hessian at initialization for different initializations. Smaller values imply lower curvature. *N/A* means that the computation diverged. The proposed strategy initializes at a point with lowest curvature, which explains why larger learning rates can be used.

6.4.4 Preliminary Reinforcement Learning Results

Despite its tremendous success in supervised learning applications, batch normalization is seldom used in Reinforcement Learning (RL), as the online nature of some of the methods and the strong correlation between consecutive batches hinder its performance. These properties suggest the need for normalization techniques like weight normalization, which are able to accelerate and stabilize training of neural networks without relying on minibatch statistics.

We consider the Asynchronous Advantage Actor-Critic (A3C) algorithm [168], which maintains a policy and a value function estimate which are updated asynchronously by different workers collecting experience in parallel. Updates are estimated based on n-step returns from each worker, resulting in highly correlated batches of n samples, whose impact is mitigated through the asynchronous nature of updates. This setup is not well suited for batch normalization and, to the best of our knowledge, no prior work has successfully applied it to this type of algorithm.

We evaluate agents using Atari environments in the Arcade Learning Environment [169]. Our initial experiments with the deep residual architecture introduced by Espeholt et al. [13] show that adding weight normalization improves convergence speed and robustness to hyperparameter configurations across different environments. However, we did not observe important differences between initialization schemes for these weight normalized models. Despite being significantly deeper than previous architectures used in RL, this model is still relatively shallow for supervised learning standards, and we observed in our computer vision experiments that performance differences arise for deeper architectures or high learning rates. The latter is known to cause catastrophic performance degradation in deep RL due to excessively large policy updates [170], so we opt for building a much deeper residual network with 100 layers. Collecting experience with such a deep policy is a very slow process even when using GPU workers. Given this computational burden, we use hyperparameters tuned in initial experiments for the deep network introduced by

Espeholt et al. [13], and report initial results in the game of Pong[4] in Figure 6.5.

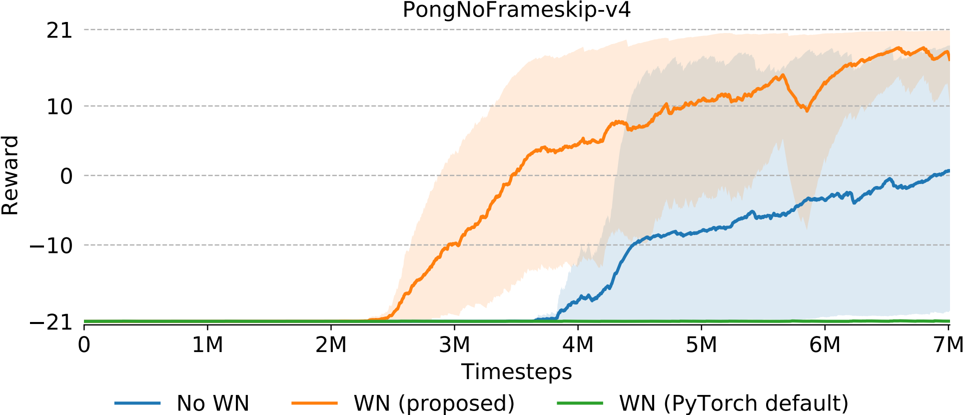


Figure 6.5: Learning progress in Pong. Shading shows maximum and minimum over 3 random seeds, while dark lines indicate the mean. Weight normalization with the proposed initialization improves convergence speed and reduces variance across seeds. These results highlight the importance of initialization in weight normalized networks, as using the default initialization in PyTorch prevents training to start.

We observe that the weight normalized policy with the proposed initialization manages to solve the task much faster than the un-normalized architecture. Perhaps surprisingly, the weight normalized policy with the sub-optimal initialization is not able to solve the environment in the given timestep budget, and it performs even worse than the un-normalized policy. These results highlight the importance of proper initialization even when using normalization techniques.

The deep network architecture considered in this experiment is excessively complex for the considered task, which can be solved with much smaller networks. However, with the development of ever complex environments [171] and distributed learning algorithms that can take advantage of massive computational resources [13], recent results have shown that RL can benefit from techniques that have found success in the supervised learning community, such as deeper residual networks [13, 172]. The aforementioned findings suggest that RL applications could benefit from techniques that help training very deep networks robustly in the future.

[4]Collecting 7M timesteps of experience took approximately 10h on a single GPU shared by 6 workers. Even though this amount of experience is enough to solve Pong, A3C usually needs many more interactions to learn competitive policies in more complex environments.

6.5 Discussion

Weight normalization is frequently used in different network architectures due to its simplicity. However, the lack of existing theory on parameter initialization of weight normalized networks has led practitioners to arbitrarily pick existing initializations designed for un-normalized networks. To address this issue, we derived parameter initialization schemes for weight normalized networks, with and without residual connections, that avoid explosion/vanishment of information in the forward and backward pass. To the best of our knowledge, no prior work had formally studied this setting. Through thorough empirical evaluation, we showed that the proposed initialization increases robustness to network depth, choice of hyperparameters and seed compared to existing initialization methods that are not designed specifically for weight normalized networks. We found that the proposed scheme initializes networks in low curvature regions, which enable the use of large learning rates. By doing so, we were able to significantly reduce the performance gap between batch and weight normalized networks which had been previously reported in the literature. Therefore, we hope that our proposal replaces the current practice of choosing arbitrary initialization schemes for weight normalized networks.

We believe our proposal can also help in achieving better performance using weight normalization in settings which are not well-suited for batch normalization. One such scenario is training of recurrent networks in backpropagation through time settings, which often suffer from exploding/vanishing gradients, and batch statistics are timestep-dependent [116]. The current analysis was done for feedforward networks, and we plan to extend it to the recurrent setting. Another application where batch normalization often fails is RL, as good estimates of activation statistics are not available due to the online nature of some of these algorithms. We confirmed the benefits of our proposal in preliminary RL experiments.

Part II

Learning from Interaction

Introduction

Humans are able to discover solutions to new problems from interaction and experience, acquiring knowledge about the world by actively exploring it. This contrasts with the passive setting considered in Part I of this dissertation, where machines learned from expert-provided outputs for each instance in the training set. We will study the problem of agents learning from interaction with simulated environments through the lens of Reinforcement Learning (RL), a computational approach to goal-directed learning from interaction that does not rely on expert supervision. RL algorithms have recently achieved outstanding goals thanks to advances in simulation [169, 173], efficient and scalable learning algorithms [12, 13, 46, 170, 174, 175], function approximation [27, 176], and hardware accelerators [55, 177]. These landmarks include outperforming humans in board [172] and computer games [174, 178], and solving complex robotic control tasks [179, 180].

The process of training RL agents can be split into collecting experience and learning from it. We will consider a generic pipeline consisting of three types of processes: actors, learners and buffers [181]. Figure 6.6 shows how these processes interact with each other[5]. Each actor has its own copy of the agent and the environment, and collects data that is aggregated at the buffer. The learner samples experience from the buffer, improving the agent and communicating updated parameters to the actors. This paradigm is very flexible, and the nature of the algorithm being implemented will depend on the way in which the different processes are deployed and communicate with each other. On-policy methods will clear the buffer after every learner query and block actors until they receive a new set of parameters. On the other hand, off-policy algorithms will store past experience and support asynchronous acting and learning.

As is often the case in deep learning, devising methods that can leverage distributed computation to address complex problems is at the frontier of RL research. The neural networks used in RL are relatively small when compared to the ones considered in

[5]For simplicity, we will consider a setting with a single learner and buffer. Multiple buffer processes can be launched to allow storing a larger number of interactions. Similarly, multiple learners can be run in parallel, akin to the settings studied in Chapter 4.

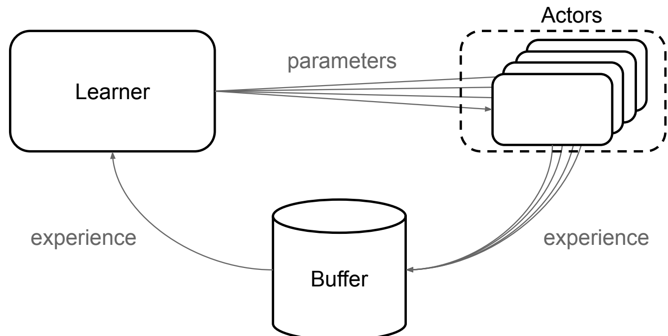


Figure 6.6: Pipeline for training RL agents. We consider three types of processes: actors, learners and buffers. Each actor has its own copy of the agent and the environment, and collects data that is aggregated at the buffer. The learner samples experience from the buffer, improving the agent and communicating updated parameters to the actors.

Part I, and the bottleneck for shortening training times is commonly found in the experience collection performed in the actors. Implementations on multi-core machines benefit from parallelizing experience collection across several actors [168], but the number of processes that can be run in parallel in a single machine soon becomes a limiting factor. Distributing these processes across different machines allows running hundreds of actors in parallel, but introduces several challenges. These include inefficient resource utilization, unstable learning due to the off-policiness of the experience, or data inefficiency due to the data being collected by different actors being redundant. Distributed RL solutions have achieved outstanding results using heterogeneous hardware configurations [12, 13]. Unfortunately, these are hard to put together for many organizations with access to general purpose clusters. Direct policy search methods have appeared as an alternative to traditional RL approaches, obtaining competitive results in research benchmarks while being scalable in homogeneous clusters. Chapter 7 presents a technique to improve the data efficiency of Evolution Strategies [49], a direct policy search method that offers almost linear speedups when distributed across hundreds of CPU cores.

The goal of RL agents is to maximize the rewards they can collect, but designing reward functions is a complex task that often leads to undesired behaviors [182]. As stronger and more efficient RL algorithms are developed, the bottleneck preventing more complex agents to emerge is pushed towards the design of reward functions. Rich simulated environments open the door to the discovery of a large plethora of behaviors, but defining reward functions for all of them becomes a herculean task. For this reason, an important step towards the next generation of RL agents might require devising methods that break the dependency on handcrafted reward functions. Chapter 8 studies one family of such methods, whose goal is letting agents discover useful skills in an unsupervised fashion. In this setting, the learner has an additional task: autonomously discovering reward functions from experience that will generate useful behaviors. Through theoretical and

empirical evidence, we show that the reward functions learned by existing approaches do not encourage agents to fully explore the environment. We then propose a novel method to overcome such limitation and demonstrate its benefits with respect to existing approaches.

7 Importance Weighted Evolution Strategies

Víctor Campos, Xavier Giró-i-Nieto, and Jordi Torres. Importance Weighted Evolution Strategies. In *NeurIPS Deep RL Workshop*, 2018

Evolution Strategies (ES) [49] was proposed as a scalable alternative to popular Reinforcement Learning (RL) techniques. Thanks to a reduced communication overhead, ES can be scaled to over a thousand CPU cores with almost linear speedups, providing massive improvements in wall-clock time when training agents in well-known RL benchmarks. This property makes ES very appealing for institutions with access to large CPU clusters. Thanks to the massive number of CPU cores in the MareNostrum IV supercomputer[1] at the Barcelona Supercomputing Center, we can reduce training times from several hours to only a few minutes by requesting more hardware resources.

The speedup of ES comes at the cost of a reduced data efficiency, i.e. more interactions with the environment are needed in order to achieve the same score as with competing methods. Even though this trade-off might not be problematic for simulated tasks, where one can turn compute into data, data efficiency is crucial for the deployment of RL agents in real world scenarios, e.g. robot manipulation tasks [183]. Research has been conducted to improve the data efficiency of other RL methods [184, 185], and we believe that ES would benefit from similar efforts as well.

[1] https://www.bsc.es/marenostrum/marenostrum

We aim at improving the data efficiency of ES while maintaining the scalability of the original method. Our contributions can be summarized as follows: (1) we propose Importance Weighted Evolution Strategies (IW-ES), an extension of ES that can perform more than one update per batch of experience, (2) analyze the scalability of IW-ES from the computational standpoint, and (3) report preliminary results for IW-ES under different configurations that provide insight on the potential of the method and possible improvements to overcome its current limitations.

7.1 Evolution Strategies

The term *Evolution Strategies* [186] makes reference to a class of black box optimization algorithms which implement heuristics inspired by natural evolution. However, we will use the term to refer to the particular algorithm proposed by Salimans et al. [49]. This method, which belongs to the class of Natural Evolution Strategies [187, 188], was shown to be competitive for solving RL problems while exhibiting some attractive features. These features include invariance to action frequency and reward distribution, the possibility to optimize non-differentiable policies, and being highly parallelizable thanks to an efficient communication strategy.

7.1.1 Formulation

Let F denote the objective function acting on parameters θ. In RL problems, it is defined as the stochastic score experienced by an agent after a complete trajectory following policy π_θ. ES seeks to maximize $\mathbb{E}_{\theta\sim p_\psi} F(\theta)$, the average objective over a population of solutions p_ψ, using the score function estimator for the gradient. Salimans et al. [49] instantiate the population as a multivariate Gaussian with diagonal covariance matrix centered at θ, thus obtaining the following estimator:

$$\nabla_\theta \, \mathbb{E}_{\epsilon\sim N(0,\sigma^2\mathbf{I})} \, F(\theta+\epsilon) = \frac{1}{\sigma^2} \, \mathbb{E}_{\epsilon\sim N(0,\sigma^2\mathbf{I})} \left[F(\theta+\epsilon)\epsilon\right] \tag{7.1}$$

which in practice is estimated with samples:

$$\nabla_\theta F(\theta) \approx \frac{1}{n\sigma^2} \sum_{i=1}^{n} F(\theta+\epsilon_i)\epsilon_i \tag{7.2}$$

Notice that this reduces to sampling Gaussian perturbation vectors $\epsilon_i \sim N(0,\mathbf{I})$, evaluating the performance of the perturbed policies, and aggregating the results over a batch of samples.

7.1.2 Scalability Analysis

The code released by Salimans et al. [49] uses a master-worker architecture. The master broadcasts the parameters at the beginning of each iteration, and the workers send back returns after running rollouts with perturbed versions of the policy. The communication overhead between workers is drastically reduced by sharing random seeds, resulting in a highly parallelizable method.

We adapt the code released by OpenAI[2], which uses TensorFlow [69] and Redis[3], to work with the distributed setting in the MareNostrum IV supercomputer. We evaluate the scalability of ES on the `Humanoid-v2` environment in OpenAI Gym [189], where the goal is solving a humanoid locomotion task using the Mujoco physics engine [173]. This is one of the most challenging continuous control tasks solvable by current RL methods, and was used by Salimans et al. [49] to showcase the scalability of ES. In these experiments, we use the default hyperparameters provided by Salimans et al. [49] and sweep over the number of CPU cores used for training agents. Figure 7.1 (top) shows how the number of environment interactions per second grows linearly with the number of CPU cores thanks to the reduced communication overhead. Since the number of perturbations to evaluate is generally much larger than the number of cores, ES manages to converge faster when given access to more resources. As shown in Figure 7.1 (bottom), the time needed to reach the score at which the task is considered solved decreases almost linearly when increasing the number of cores. ES is able to leverage 1,440 cores in MareNostrum IV in order to solve the challenging humanoid locomotion task in only 8 minutes.

7.2 Importance Weighted Evolution Strategies

ES samples large batches of data, in the order of thousands of trajectories, which are discarded after performing a single policy update. When coupled with SGD and small step sizes, this translates into a poor data efficiency. Such inefficiency is found in most on-policy RL methods, which are unable to leverage previous experience once the policy is updated.

Inspired by the multiple SGD updates per batch of experience in PPO [175], we propose to modify the ES algorithm to perform several updates to the policy before moving on to collecting a new batch of experience. Should each of these updates be small, it is likely that the population distributions before and after the update will have some overlap,

[2] https://github.com/openai/evolution-strategies-starter/
[3] https://redis.io/

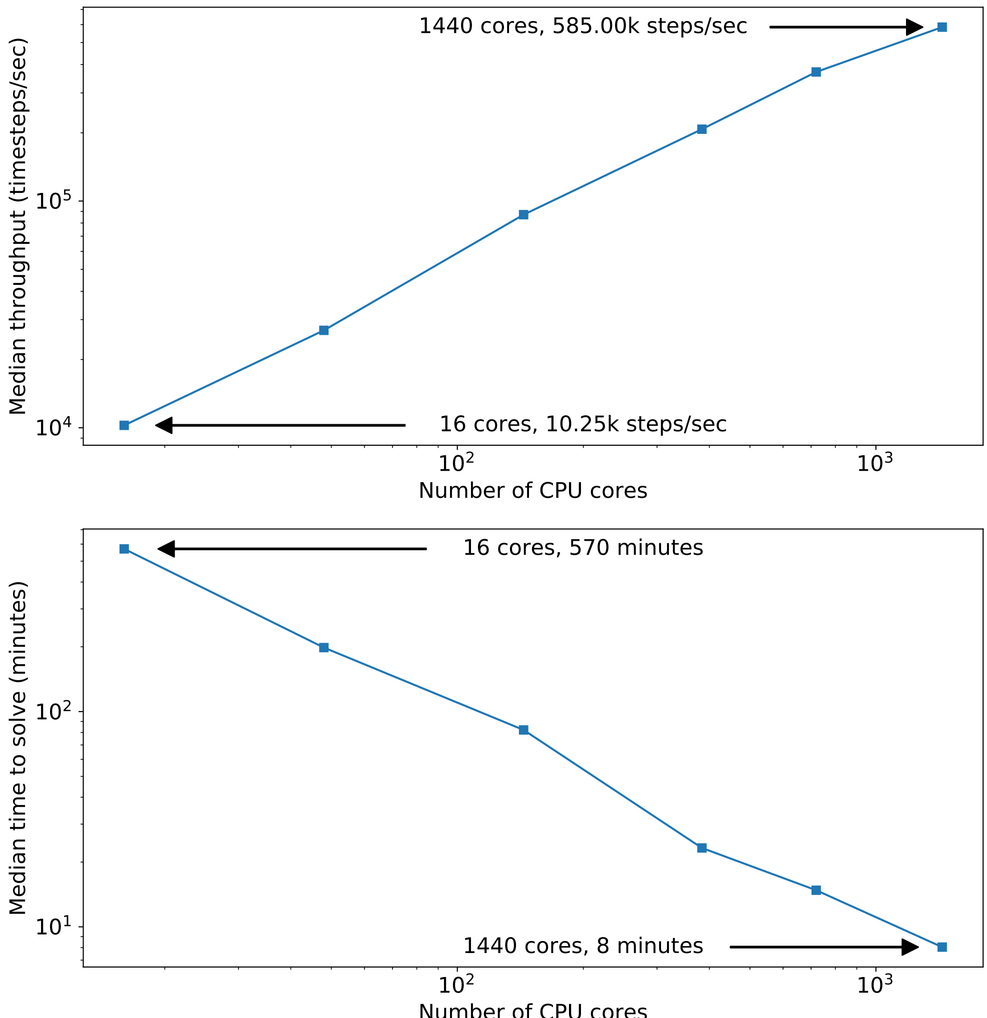


Figure 7.1: Scalability of Evolution Strategies in the humanoid locomotion task as the number of CPU cores grows. **Top:** throughput, measured as the number of environment interactions per second. **Bottom:** time to reach a target score. We report the median throughput and time over 7 individual runs for each setting.

thus making it possible to take more advantage of previous computations and reducing the number of interactions with the environment.

7.2.1 Formulation

Let $\theta^t \in \mathbb{R}^{|\theta|}$ denote the population mean after t updates, and $\epsilon_i^t \in \mathbb{R}^{|\theta|}$ denote the perturbations for which we computed fitness scores, $F(\theta^t + \epsilon_i^t)$. We can reuse those samples to update θ^{t+k} by relying on importance sampling to account for the discrepancy between the distribution of the current population and the distribution from which we

are actually sampling:

$$\nabla_\theta F(\theta) \approx \frac{1}{\sigma^2 \sum_i c_i} \sum_{i=1}^{n} F(\theta^t + \epsilon_i^t)(\theta^t + \epsilon_i^t - \theta^{t+k})c_i \quad (7.3)$$

where $c_i \in \mathbb{R}$ is the importance weight for the i-th perturbation vector. For perturbations drawn from a multivariate Gaussian distribution with diagonal covariance matrix, the computation of c_i can be decomposed as follows:

$$c_i = \frac{N(\theta^t + \epsilon_i^t - \theta^{t+k}; 0, \sigma^2\mathbf{I})}{N(\epsilon_i^t; 0, \sigma^2\mathbf{I})} = \frac{\prod_{j=1}^{|\theta|} N(\theta_j^t + \epsilon_{i,j}^t - \theta_j^{t+k}; 0, \sigma^2)}{\prod_{j=1}^{|\theta|} N(\epsilon_{i,j}^t; 0, \sigma^2)} \quad (7.4)$$

This process can be repeated iteratively for $k = 0, \ldots, K$, updating the policy up to $K + 1$ times before collecting a new batch of experience[4]. We consider K as a fixed hyperparameter, although future work will study strategies that optimally adapt K for each batch.

7.2.2 Scalability Analysis

One of the most appealing features of ES is its almost perfect scalability to hundreds of CPU cores, and any modification to the original method should retain such property. This section analyzes the scalability of IW-ES under the master-worker architecture used by Salimans et al. [49].

The proposed method requires the computation of importance weights, which has a complexity of $\mathcal{O}(\text{batch_size} \cdot |\theta|)$. If those computations are performed sequentially in the master, the time taken by sequential operations might eclipse the benefits of distributing the rollouts across hundreds of workers. This issue can be alleviated by parallelizing the computation of importance weights across all cores in the node hosting the master process. This was enough to provide a throughput close to the baseline method in most of our experiments, but setups with larger models or batch sizes might benefit from a higher level of parallelization. In that case, the computation of importance weights can be distributed across all workers just like the rollouts are: the master broadcasts the updated parameter vector, and the workers send back the scalar importance weights. Note that this incurs in a very little communication overhead, which is key to achieve an efficient distributed computation.

Another implementation trick that can accelerate the computation of importance weights consists in computing $N(\epsilon_{i,j}^t; 0, \sigma^2)$ for all possible perturbations at the start of training,

[4] The first update for each batch always reduces to the original gradient estimate in ES (Equation 7.2), as $\theta^{t+k} = \theta^t$ for $k = 0$. This is followed by K importance weighted updates.

trading off memory for computation. It takes advantage of the fact that each worker instantiates a large block of Gaussian noise at the start of training, and ϵ_i is obtained by sampling $|\theta|$ consecutive parameters at a random index in the noise block. This trick might provide important savings for large models, as the computation of the denominator in Equation 7.4 becomes $\mathcal{O}(1)$ instead of $\mathcal{O}(|\theta|)$.

7.3 Experiments

We implement our method on top of the TensorFlow-based code released by OpenAI, and run experiments on the MareNostrum IV supercomputer. Each experiment runs on 720 CPU cores, which are distributed across 15 machines with 48 cores each. The master process runs on a single core, but the computation of importance weights is parallelized across the 48 cores in the node hosting the master process to accelerate the execution.

We evaluate the method on the `Ant-v2` environment[5]. We use the default hyperparameters provided by Salimans et al. [49] unless otherwise stated. The policy is parameterized by a neural network with two hidden layers of 64 units each and a linear layer that emits continuous actions. Hidden layers are followed by *tanh* non-linearities. Importance weights are clipped at 1 for numerical stability [190, 191]. Following previous works [49, 192], we evaluate the median reward over approximately 30 stochastic rollouts at each iteration. All reported results are averaged over five different runs.

7.3.1 Effect of the Number of IW Updates

The proposed method relies on a high overlap between the population distributions before and after each update, otherwise the variance of the importance sampling estimate might become excessively large. For this reason, we first evaluate the effectiveness of additional updates using a low learning rate of 10^{-4} that prevents large updates to the policy parameters. As depicted in Figure 7.2a, we observe that additional importance weighted updates provide a faster convergence for a given budget of interactions with the environment. Increased data efficiency also translate in shorter wall-clock times thanks to a reduced computational overhead (Figure 7.2b). However, performance does not always improve when increasing K, e.g. setting $K = 5$ instead of $K = 4$ results in a performance degradation. This behavior is likely caused by an increased variance in the importance weighted updates for large values of K. These results suggest that IW-ES

[5] Although we provide an extensive analysis of IW-ES only on `Ant-v2`, we have observed similar behaviors on other complex environments, e.g. `Humanoid-v2`.

might benefit from strategies that adapt K for each iteration, omitting updates with excessive variance.

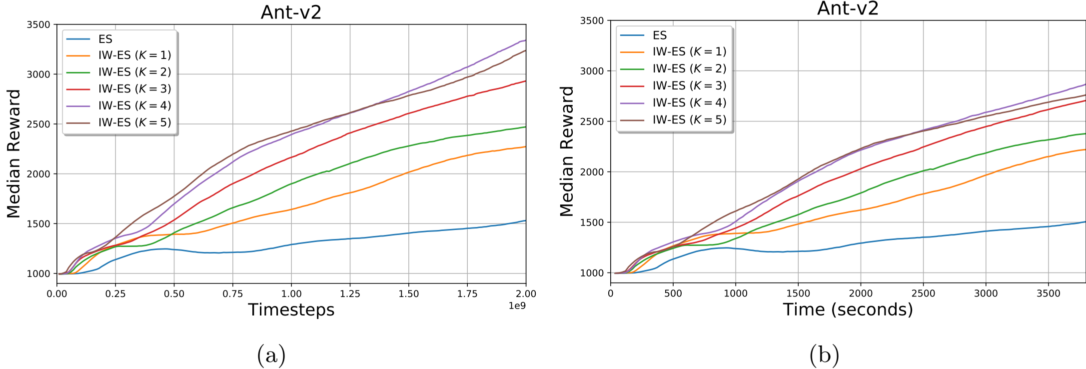


Figure 7.2: Performance of ES and IW-ES as a function of **(a)** the number of interactions with the environment, and **(b)** wall-clock time. K denotes the number of additional importance weighted updates after each standard update. We observe that additional updates increase the data efficiency of the method in the low learning rate regime, but performing too many importance weighted updates can be detrimental due to an increased variance, e.g. $K = 5$ underperforms $K = 4$. A similar trend is observed in terms of wall-clock time.

7.3.2 Effect of the Model Size

A potential source of instability for the proposed method is the computation of importance weights for large models, as they might approach zero or infinity much faster for large values of $|\theta|$ (see Equation 7.4). We experimentally evaluate whether this hinders the performance of IW-ES by training larger networks, with 256 and 512 units in each hidden layer. These larger models have 97k and 324k parameters, respectively, whereas previous experiments considered a much smaller network with 12k parameters. Results reported in Figure 7.3 suggest that IW-ES is robust to the number of parameters in the model, as the benefit of adding additional updates per batch are similar to those observed for smaller models.

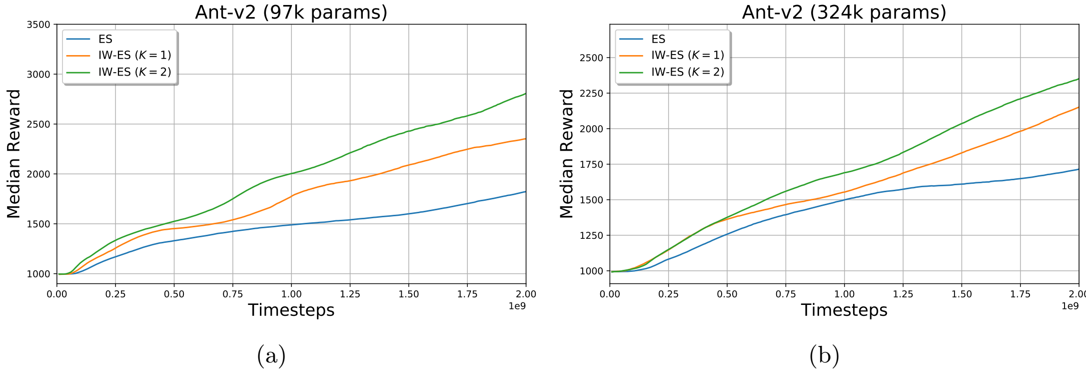


Figure 7.3: Performance of ES and IW-ES for larger networks with **(a)** 256 units per hidden layer, and **(b)** 512 units per hidden layer.

Figure 7.4 shows the throughput degradation introduced by IW-ES for each model size and number of importance weighted updates. Since our implementation only leverages 48 of the 720 available CPU cores for computing the importance weights, such computation becomes a bottleneck for larger models and hinders the scalability of the method. This observation motivates the distributed implementation described in Section 7.2.2, which should accelerate IW-ES considerably for large models thanks to the reduced communication overhead between machines.

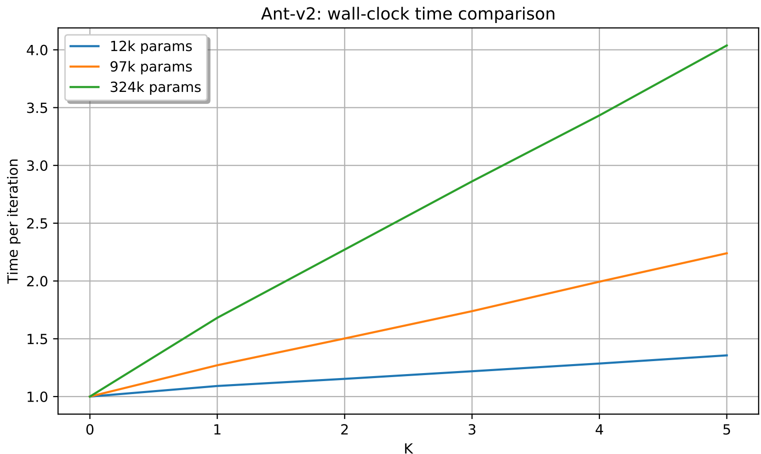
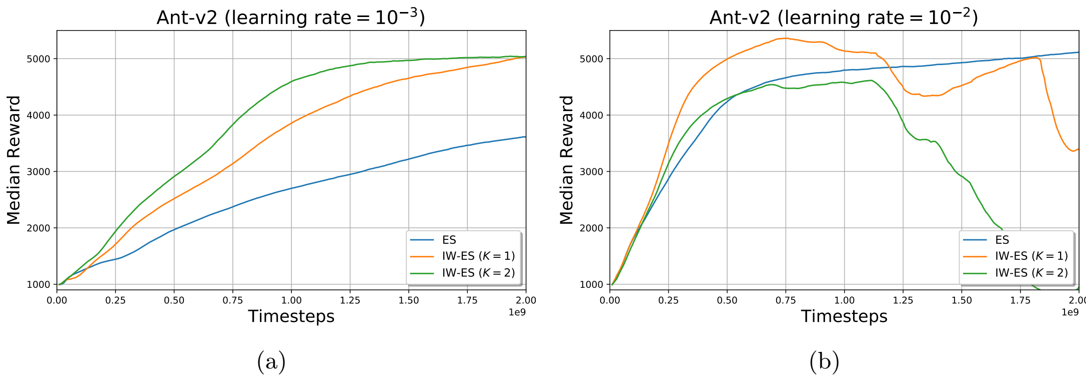


Figure 7.4: Time per iteration for different values of K, normalized by the time taken by ES (i.e. $K = 0$). Our implementation parallelizes the computation of importance weights only across the CPU cores in the node hosting the master process, which becomes a bottleneck for larger models.

7.3.3 Effect of the Learning Rate

ES benefits from larger learning rates than those employed in previous experiments, as they provide faster convergence and thus increased data efficiency, but larger step sizes might increase the variance of IW-ES updates as well due to a larger mismatch between distributions. We evaluate this hypothesis by training policies with larger learning rates of 10^{-3} and 10^{-2}. Results reported in Figure 7.5 confirm that importance weighted updates not only become less effective with larger learning rates, but can even become unstable and underperform the baseline ES. We hypothesize that this might be caused by an increased variance of the importance sampling estimates when using large learning rates.

These experiments consider the learning rate as a proxy for controlling the overlap between the distributions before and after each update, which is the actual measure determining the variance of importance weighted updates. Even though a finer grained search over learning rate values could be carried out in order to determine whether IW-ES can outperform ES under optimal hyperparameters, we argue that next steps should aim at



(a) (b)

Figure 7.5: Performance of ES and IW-ES with learning rates of **(a)** 10^{-3}, and **(b)** 10^{-2}. Larger learning rates reduce the benefits of IW-ES, likely due to an increased variance of the importance sampling estimate.

controlling the similarity between the distributions before and after each update. For instance, drawing a parallelism with trust region-based methods [170, 193], a constraint could be added on the KL divergence between distributions.

7.4 Related Work

Some works have proposed extensions or modifications to the original ES algorithm proposed by Salimans et al. [49]. These include an update rule inspired by genetic algorithms [50] and training a meta-population of agents that optimize both for reward and novelty [192]. The possibility of optimizing non-differentiable functions with ES has also allowed to learn loss functions for RL in a meta-learning setup [194].

The design of data-efficient methods for RL has garnered much research attention, mostly through off-policy methods that can leverage experience collected by policies other than the one being optimized. This advantage, often associated to value-based methods such as Q-learning [174, 195], usually results in an increased data efficiency. Policy-based methods may also leverage off-policy data by accounting for the discrepancy between the behavior and target policies [191, 196]. PPO [175] performs several SGD updates for every batch of collected experience, using importance sampling to leverage data collected by an outdated version of the policy, in a similar fashion to our IW-ES update rule.

There exist other RL agents that are able to leverage distributed training to obtain high throughputs. R2D2 [12] provides the DQN [174] family of agents with distributed acting, which is coordinated through a centralized replay buffer and learner. IMPALA [13] can scale training of actor-critic methods across many machines, enabling advances in multi-task RL [197]. This is achieved through algorithmic contributions that ensure stable

learning, as well as engineering advances that enable efficient communication across machines. The main drawback of these approaches comes from a fairly uncommon hardware setup, where each GPU learner is paired with hundreds of CPU cores that interact with the environment. Such heterogeneous combination of hardware resources may not be feasible to put together within many organizations. In comparison, ES requires from less engineering efforts to achieve high throughputs, thanks to the reduced communication overhead, and its hardware requirements are generally easier to meet.

7.5 Discussion

We introduced IW-ES, a variant of ES [49] that can perform several model updates with a single of batch of data. Under the desired conditions, i.e. when samples from the population distribution before the update are still likely under the updated distribution, IW-ES demonstrated a higher data efficiency than that of ES. For small models, these benefits can be introduced with a small increase in sequential computational load that maintains the scalability of ES. For larger models, we describe how to leverage distributed hardware to distribute further parallelize the added computation and achieve higher throughput rates.

Besides implementing the completely distributed version of IW-ES that can make the most of the available hardware, future work may focus on making IW-ES more resilient to large divergences between distributions that increase the variance of the importance sampling estimates. First, an adaptive strategy for K can be designed so that importance weighted updates are made only when their variance is sufficiently low. On the other hand, controlling the divergence after an update through a constraint in the training objective can make IW-ES more robust for large learning rates, and avoid the collapse observed in some experiments. Although applied in policy space instead of parameter space, similar motivations have led to more efficient and stable policy gradient methods [170, 175]. These lines of research may also lead to revisiting the role of σ, which controls the spread of the perturbation vectors in ES, but also plays an important role in determining the importance weights in IW-ES.

8

Unsupervised Discovery of State-Covering Skills

Víctor Campos, Alexander Trott, Caiming Xiong, Richard Socher, Xavier Giró-i-Nieto, and Jordi Torres. Explore, Discover and Learn: Unsupervised discovery of state-covering skills. In *ICML*, 2020

Training of Reinforcement Learning (RL) agents typically aims to solve a particular task, relying on task-specific reward functions to measure progress and drive learning. This contrasts with how intelligent creatures learn in the absence of external supervisory signals, acquiring abilities in a task-agnostic manner by exploring the environment. Methods for training models without expert supervision have already obtained promising results in fields like natural language processing [10, 198] and computer vision [199, 200]. In RL, analogous "unsupervised" methods are often aimed at learning generically useful behaviors for interacting within some environment, behaviors that may naturally accelerate learning once one or more downstream tasks become available.

The idea of unsupervised RL is often formulated through the lens of *empowerment* [201], which formalizes the notion of an agent discovering *what* can be done in an environment while learning *how* to do it. Central to this formulation is the concept of mutual information, a tool from information theory [202]. Mohamed and Rezende [203] derived a variational lower bound on the mutual information which can be used to learn options [204] in a task-agnostic fashion. Following classical empowerment [201], options are discovered by maximizing the mutual information between sequences of actions and final states. This results in *open loop* options, where the agent commits to a sequence

of actions *a priori* and follows them regardless of the observations received from the environment. Gregor et al. [205] developed an algorithm to learn *closed loop* options, whose actions are conditioned on the state, by maximizing the mutual information between states and some latent variables instead of action sequences. This approach has been extended by several works, which are surveyed in Section 8.1. Despite the interest in developing information-theoretic skill discovery methods, very little research has been conducted in understanding the limitations of these algorithms. We distinguish two categories of such limitations. The first type has to do with the nature of the objective itself, e.g. the difficulty of purely information-theoretic methods for capturing human priors [206]. We focus on the second group, i.e. on those issues introduced when adapting the objective to current optimization methods.

In order to maximize empowerment, an agent needs to learn to control the environment while discovering available options. It should not aim for states where it has the most control according to its current abilities, but for states where it expects it will achieve the most control *after* learning [205]. We empirically observe that this is not achieved by existing methods, which prematurely commit to already discovered options instead of exploring the environment to unveil novel ones. Figure 8.1 (top) showcases this failure mode when deploying existing algorithms on a 2D maze. We provide theoretical analysis showing that these methods tend to reinforce already discovered behaviors at the expense of exploring in order to discover new ones, resulting in behaviors that exhibit poor coverage of the available state space. Figure 8.1 (bottom right) depicts the skills discovered by our proposed Explore, Discover and Learn (EDL) paradigm, a three-stage methodology that is able to discover skills with much better coverage.

Our contributions can be summarized as follows. (1) We provide theoretical analysis and empirical evidence showing that existing skill discovery algorithms fail at learning state-covering skills. (2) We propose EDL, an alternative approach to information-theoretic option discovery that overcomes the limitations of existing methods. Crucially, EDL achieves this while optimizing the same information-theoretic objective as previous methods. (3) We validate the presented paradigm by implementing a solution that follows the three-stage methodology. Through extensive evaluation in controlled environments, we demonstrate the effectiveness of EDL, showcase its advantages over existing methods, and analyze its current limitations and directions for future research.

8.1 Information-Theoretic Skill Discovery

This section presents a generic mathematical framework that can be used to formulate, analyze and compare information-theoretic skill discovery methods in the literature. Let

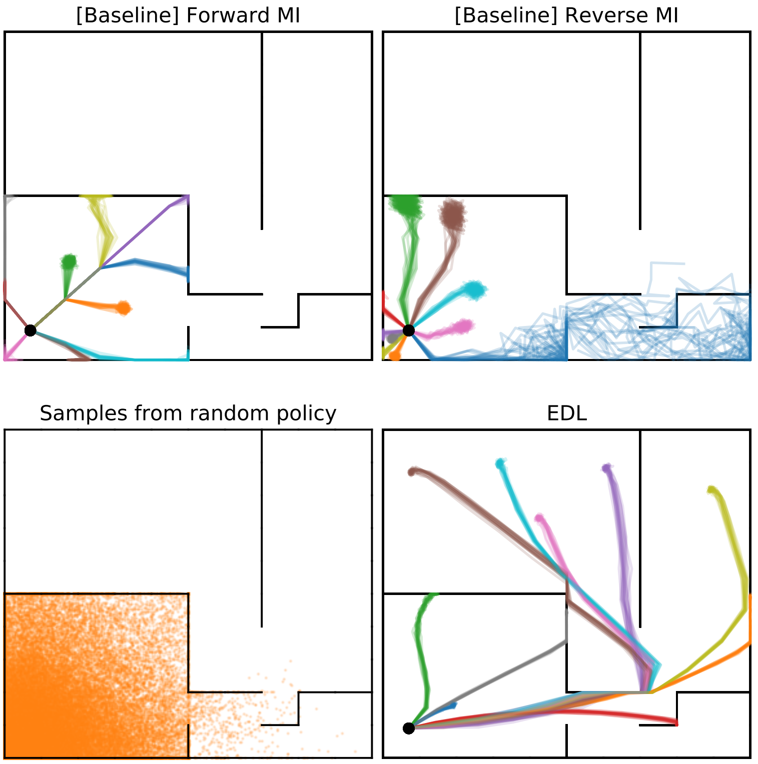


Figure 8.1: Skills learned on a maze with bottleneck states. Each colored line represents a trajectory initiated at the black dot by a different skill. Multiple rollouts per skill are reported in order to account for the stochasticity of the policy. The bottom left plot depicts states visited by a policy with random weights, showing which states are reachable by the agent at the beginning of training. The top row shows that existing methods fail at expanding this set of states, and end up committing to behaviors discovered by the random policy. This failure happens with both forms of the mutual information (MI). On the other hand, EDL discovers skills that provide a better coverage of the state space.

us consider a Markov Decision Process (MDP) $\mathcal{M} \equiv (\mathcal{S}, \mathcal{A}, p)$ with state space $\mathcal{S}$, action space $\mathcal{A}$ and transition dynamics p. We learn latent-conditioned policies $\pi(a|s,z)$, and define *skills* or *options* as the policies obtained when conditioning π on a fixed value of $z \in \mathcal{Z}$ [206, 207]. Let $S \sim p(s)$ be a random variable denoting states such that $S \in \mathcal{S}$, and $Z \sim p(z)$ be a random variable for latent variables. Using notation from information theory, we use $I(\cdot;\cdot)$ and $H(\cdot)$ to refer to the mutual information and Shannon entropy, respectively. Information-theoretic skill discovery methods seek to find a policy that maximizes the mutual information between S and Z. Due to symmetry, this measure can be expressed in the following two forms:

$$I(S;Z) = H(Z) - H(Z|S) \qquad // \textit{ reverse} \tag{8.1}$$

$$= H(S) - H(S|Z) \qquad // \textit{ forward} \tag{8.2}$$

For presentation clarity, we follow Gregor et al. [205] and refer to Equations 8.1 and 8.2 as the *reverse* and *forward* forms of the mutual information, respectively.

Our goal is to analyze which fundamental design choices are responsible for the properties and limitations of such algorithms. We classify existing skill discovery methods depending on the form of the mutual information they optimize, and implement a canonical algorithm for each form that allows for fair comparison. The following subsections describe existing skill discovery methods as well as the specific implementations considered in this work.

8.1.1 Reverse Form of the Mutual Information

The objective can be derived by expanding the definition of the mutual information in Equation 8.1, and then leveraging the non-negativity property of the KL divergence to compute a variational lower bound [208]:

$$I(S;Z) = \underbrace{\mathbb{E}_{s,z\sim p(s,z)}[\log p(z|s)]}_{-H(Z|S)} \underbrace{-\mathbb{E}_{z\sim p(z)}[\log p(z)]}_{H(Z)} \tag{8.3}$$

$$\geq \mathbb{E}_{s,z\sim p(s,z)}[\log q_\phi(z|s)] - \mathbb{E}_{z\sim p(z)}[\log p(z)] \tag{8.4}$$

where $q_\phi(z|s)$ is fitted by maximum likelihood on (s, z)-tuples collected by deploying the policy in the environment. This implicitly approximates the unknown posterior as $p(z|s) \approx \rho_\pi(z|s)$, where $\rho_\pi(z|s)$ is the empirical posterior induced by the policy.

Note that computing this measure will require sampling from two distributions, $p(z)$ and $p(s, z)$. The distribution over latent variables $p(z)$ can be learned as part of the optimization process or fixed beforehand, with the latter often yielding superior results [207]. However, sampling from the joint distribution over states and latents $p(s, z)$ is more problematic. A common workaround consists in assuming a generative model of the form $p(s, z) = p(z)p(s|z) \approx p(z)\rho_\pi(s|z)$, where $\rho_\pi(s|z)$ is the stationary state-distribution induced by $\pi(a|s, z)$ [205].

This category includes a variety of methods with slight differences. VIC [205] considers only the final state of each trajectory and a learnable prior $p(z)$. SNN4HRL [209] introduces a task-specific proxy reward, which encourages exploration and can be understood as a bonus to increase the entropy of the stationary state-distribution. DIAYN [207] additionally minimizes the mutual information between actions and skills given the state, resulting in a formulation that resembles maximum entropy RL [210]. VALOR [206] considers the posterior over sequences of states instead of individual states in order to encourage learning dynamical modes rather than goal-attaining modes. VISR [211] combines skill discovery with universal successor features approximators [212] to enable

fast task inference [213, 214]. DISCERN [215] and Skew-Fit [216] aim at learning a goal-conditioned policy in an unsupervised fashion, which can be understood as skill discovery methods where Z takes the form of states sampled from a buffer of previous experience.

We consider a variant of VIC [205] with a fixed prior $p(z)$ and where all states in a trajectory are considered in the objective[1]. This method can be seen as a version of DIAYN [207] where the scale of the entropy regularizer $H(A|S,Z)$ is set to 0. The variational lower bound in Equation 8.4 is optimized by training the policy $\pi(a|s,z)$ using the reward function

$$r(s,z') = \log q_\phi(z'|s) - \log p(z'),\ z' \sim p(z) \tag{8.5}$$

8.1.2 Forward Form of the Mutual Information

A similar lower bound to that in Equation 8.4 can be derived by expanding the forward form of the mutual information in Equation 8.2:

$$I(S;Z) = \underbrace{\mathbb{E}_{s,z\sim p(s,z)}[\log p(s|z)]}_{-H(S|Z)} \underbrace{-\mathbb{E}_{s\sim p(s)}[\log p(s)]}_{H(S)} \tag{8.6}$$

$$\geq \mathbb{E}_{s,z\sim p(s,z)}[\log q_\phi(s|z)] - \mathbb{E}_{s\sim p(s)}[\log p(s)] \tag{8.7}$$

where $q_\phi(s|z)$ is fitted by maximum likelihood on (s,z)-tuples collected by deploying the policy in the environment. This amounts to approximating $p(s|z)$ with the stationary state-distribution of the policy, $p(s|z) \approx \rho_\pi(s|z)$.

To the best of our knowledge, DADS [217] is the only method within this category. DADS follows a model-based setup where $I(S_{t+1};Z|S_t)$ is maximized. This is achieved by modelling changes in the state, $\Delta s = s_{t+1} - s_t$. When evaluated on locomotion environments that encode the position of the agent in the state vector, this setup favors the discovery of gaits that move in different directions. Similarly to methods in Section 8.1.1, $p(s,z)$ is approximated by relying on the stationary state-distribution induced by the policy and $p(s) \approx \rho_\pi(s) = \mathbb{E}_z[\rho_\pi(s|z)]$. We will consider a model-free variant of DADS where the variational lower bound in Equation 8.7 is optimized by training the policy $\pi(a|s,z)$ with a reward function

$$r(s,z') = \log q_\phi(s|z') - \log \frac{1}{L}\sum_{i=1}^{L} q_\phi(s|z_i),\ z', z_i \sim p(z) \tag{8.8}$$

[1] The original implementation by Gregor et al. [205] considered final states only, thus providing a sparser reward signal to the policy.

where $p(s)$ is approximated using q_ϕ and L random samples from the prior $p(z)$ as done by Sharma et al. [217]. When using a discrete prior, we marginalize over all skills.

8.2 Limitations of Existing Methods

Recall that maximizing empowerment implies fulfilling two tasks, namely discovering what is possible in the environment and learning how to achieve it. In preliminary experiments, we observed that existing methods discovered skills that provide a poor coverage of the state space. This suggests a limited capability for discovering what options are available.

This section provides insight for why existing methods do not encourage the discovery state-covering skills from a theoretical lens. This is achieved by analyzing the reward function of these methods, and studying its asymptotic behavior for known and novel states. Our main result shows that the agent receives larger rewards for visiting known states than discovering new ones. The following subsections introduce the considered assumptions and derive these results for both forms of the mutual information.

8.2.1 Assumptions

Maximizing the mutual information between states and latents requires knowledge of some distributions. Methods based on the forward form of the mutual information make use of $p(s|z)$ and $p(s)$, whereas those using the reverse form employ $p(z|s)$. Note that none of these are known *a priori*, so the common practice is to approximate them using the distributions induced by the policy. Distributions over states are approximated with the stationary state-distribution of the policy, $p(s|z) \approx \rho_\pi(s|z)$ and $p(s) \approx \rho_\pi(s) = \mathbb{E}_z\left[\rho_\pi(s|z)\right]$. The posterior $p(z|s)$ is approximated with the empirical distribution induced by running the policy, $p(z|s) \approx \rho_\pi(z|s)$. In practice, these distributions are estimated via maximum likelihood using rollouts from the policy.

8.2.2 Reverse Form of the Mutual Information

The objective for these methods is

$$
\begin{aligned}
I(S;Z) &= \mathbb{E}_{s,z\sim p(s,z)}[\log p(z|s)] - \mathbb{E}_{z\sim p(z)}[\log p(z)] && (8.9)\\
&\approx \mathbb{E}_{s,z\sim p(s,z)}[\log \rho_\pi(z|s)] - \mathbb{E}_{z\sim p(z)}[\log p(z)] && (8.10)
\end{aligned}
$$

where the unknown posterior $p(z|s)$ is approximated by the distribution induced by the policy, $\rho_\pi(z|s)$. This distribution is estimated with a model $q_\phi(z|s)$ trained via maximum likelihood on (s, z)-tuples collected by deploying the policy in the environment. For this analysis, however, we will assume access to a perfect estimate of $\rho_\pi(z|s)$. When considering the discovery of N discrete skills under a uniform prior, the reward in Equation 8.5 becomes

$$r(s, z') = \log \rho_\pi(z'|s) - \log p(z') \tag{8.11}$$

$$= \log \rho_\pi(z'|s) + \log N \tag{8.12}$$

where $z' \sim p(z)$. We will assume that $\sum_{i=1}^{N} \rho_\pi(z_i|s) = 1$ in our analysis.

Maximum reward for known states. The reward function encourages policies to discover skills that visit disjoint regions of the state space where $\rho_\pi(z'|s) \to 1$:

$$r_{\text{max}} = \log 1 + \log N = \log N \tag{8.13}$$

Reward for previously unseen states. Note that $\rho_\pi(z|s)$ is not defined for unseen states, and we will assume a uniform prior over skills in this undefined scenario, $\rho_\pi(z|s) = 1/N, \forall z$:

$$r_{\text{new}} = \log \frac{1}{N} + \log N = 0 \tag{8.14}$$

Alternatively, one could add a *background* class to the model in order to assign null probability to unseen states [218]. This differs from the setup in previous works, reason why it was not considered in the analysis. However, note that in this scenario the agent gets an even larger penalization for visiting new states:

$$r^{'}_{\text{new}} = \lim_{\rho_\pi(z'|s) \to 0} \log \rho_\pi(z'|s) + \log N = -\infty \tag{8.15}$$

These observations explain why the learned skills provide a poor coverage of the state space.

8.2.3 Forward Form of the Mutual Information

The objective for these methods is

$$I(S;Z) = \mathbb{E}_{s,z\sim p(s,z)}[\log p(s|z)] - \mathbb{E}_{s\sim p(s)}[\log p(s)] \tag{8.16}$$
$$= \mathbb{E}_{s,z\sim p(s,z)}[\log \rho_\pi(s|z)] - \mathbb{E}_{s\sim \rho_\pi(s)}[\log \rho_\pi(s)] \tag{8.17}$$

where the unknown distributions $p(s|z)$ and $p(s)$ are approximated using the stationary state-distribution, $p(s|z) \approx \rho_\pi(s|z)$ and $p(s) \approx \rho_\pi(s) = \mathbb{E}_z[\rho_\pi(s|z)]$. The stationary state-distribution is estimated with a model $q_\phi(s|z)$ trained via maximum likelihood on (s,z)-tuples collected by deploying the policy in the environment. For this analysis, however, we will assume access to a perfect estimate of $\rho_\pi(s|z)$. When considering the discovery of N discrete skills, the reward in Equation 8.8 can be expanded as follows:

$$r(s,z') = \log \rho_\pi(s|z') - \log \frac{1}{N}\sum_{\forall z_i} \rho_\pi(s|z_i) \tag{8.18}$$
$$= \log \frac{\rho_\pi(s|z')}{\sum_{\forall z_i}\rho_\pi(s|z_i)} + \log N \tag{8.19}$$
$$= \lim_{\epsilon\to 0} \log \frac{1}{1+\sum_{\forall z_i\neq z'} \frac{\rho_\pi(s|z_i)+\epsilon}{\rho_\pi(s|z')+\epsilon}} + \log N \tag{8.20}$$

where $z', z_i \sim p(z)$ and we added $\epsilon \to 0$ in the last step to prevent division by 0.

Maximum reward for known states. As observed by Sharma et al. [217], this reward function encourages skills to be predictable (i.e. $\rho_\pi(s|z') \to 1$) and diverse (i.e. $\rho_\pi(s|z_i) \to 0, \forall z_i \neq z'$):

$$r_{\max} = \log 1 + \log N = \log N \tag{8.21}$$

Reward for previously unseen states. In novel states, $\rho_\pi(s|z_i) \to 0, \forall z_i$:

$$r_{\max} = \lim_{\epsilon\to 0} \log \frac{1}{1+\sum_{\forall z_i\neq z'} \frac{\epsilon}{\epsilon}} + \log N \tag{8.22}$$
$$= \log \frac{1}{1+(N-1)} + \log N \tag{8.23}$$
$$= \log \frac{1}{N} + \log N \tag{8.24}$$
$$= 0 \tag{8.25}$$

This result shows that visiting known states instead of exploring unseen ones provides larger rewards to the agent, producing options that provide a poor coverage of the state space.

8.2.4 Summary of Findings

We analyzed the asymptotic behavior of the reward function for existing methods under the aforementioned approximations through a theoretical lens. The analysis considered an agent aiming to discover N discrete skills, and perfect estimations of all distributions. Our main result shows that the agent receives larger rewards for visiting known states than discovering new ones. Known states can receive a reward of up to $r_{\text{max}} = \log N$. On the other hand, previously unseen states will receive a smaller reward, $r_{\text{new}} = 0$. These observations hold for the forward and reverse forms of the mutual information, and provide theoretical insight for why existing methods do not discover state-covering skills. Figure 8.2 provides a numerical example on a gridworld environment, where we handcraft two skills and depict the reward landscape they generate.

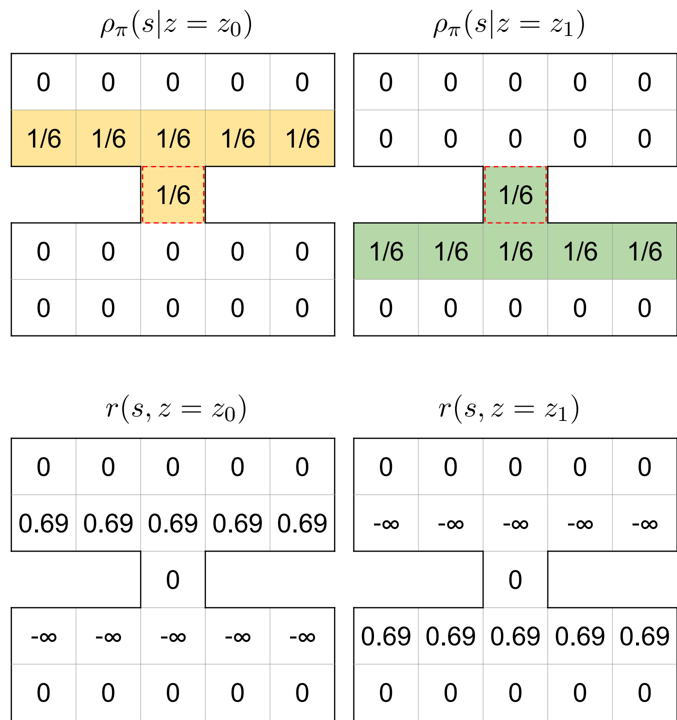


Figure 8.2: Analysis of the reward landscape on a toy gridworld with two handcrafted skills. Assuming perfect density estimation, both forms of the mutual information generate the same reward landscape. Each column depicts a different skill, and all rollouts always start from the central tile which is highlighted in red. Since skills rewarded for visiting known states where they are maximally distinguishable, but receive no reward for visiting novel states.

In order to provide preliminary experimental evidence for this result, we deploy the described algorithms on a 2D maze with bottleneck states (see Section 8.4 for details on the experimental setup). As shown in Figure 8.1 (top), existing methods fail at exploring the maze and most options just visit different regions of the initial room. Figure 8.1 (bottom left) depicts states visited by a policy with the same architecture, but random weights. Note that existing algorithms do not expand the set of states visited by this

random policy, but simply identify and reinforce different modes of behavior among them. This observation confirms that existing formulations fail at discovering available options, and motivates our study of alternative methods for option discovery.

8.3 Proposed Method

Maximizing the mutual information between states and latent variables requires access to unknown distributions, which existing methods approximate using the distributions induced by the policy. Instead of encouraging the agent to discover available options, this approximation reduces the problem to that of reinforcing already discovered behaviors. Since the policy is initialized randomly at the beginning of training, the discovered options seldom explore further than a random policy.

We propose an alternative approach, *Explore, Discover and Learn* (EDL), for modelling these unknown distributions and performing option discovery. Existing methods make use of the state distribution $p(s) \approx \rho_\pi(s) = \mathbb{E}_z\left[\rho_\pi(s|z)\right]$, which focuses $p(s)$ around states where the policy receives a high reward. This dependency contributes to the pathological learning dynamics described above. To break this dependency, EDL makes use of a *fixed* distribution over states $p(s)$ and is agnostic to the method by which this distribution is discovered or obtained. For a given distribution over states, EDL makes use of variational inference techniques to model $p(s|z)$ and $p(z|s)$. As its name suggests, EDL is composed of three stages: (*i*) exploration, (*ii*) skill discovery, and (*iii*) skill learning. These can be studied and improved upon independently, and the actual implementation of each stage will depend on the problem being addressed. The compartmentalization of these facets of the objective, together with the inclusion of a fixed distribution over states, are the key features of EDL. Table 8.1 positions this new approach with respect to existing ones.

Exploration. In the absence of any prior knowledge, a reasonable choice for the distribution over states $p(s)$ is a uniform distribution over all $\mathcal{S}$, which will encourage the discovery of state-covering skills. This stage comes with the challenge of being able to generate or sample from the distribution of states that the learned skills should ultimately cover. This is generally a difficult problem, for which we consider possible solutions. When an oracle is available, it can be queried for samples belonging to the set of valid states. If such an oracle is not available, one can train an exploration policy that induces a uniform distribution over states. Finding these policies is known as the problem of maximum entropy exploration, for which provably efficient algorithms exist under certain conditions [219]. When interested in some particular modes of behavior, one can leverage a more specific state distribution or adopt a non-parametric solution by

Assumptions	MI form	Methods
$p(z)$: fixed $p(z\|s) \approx \rho_\pi(z\|s)$ $p(s\|z) \approx \rho_\pi(s\|z)$ $p(s) \approx \rho_\pi(s) = \mathbb{E}_z\left[\rho_\pi(s\|z)\right]$	Forward	DADS [217]
	Reverse	VIC [205], SNN4HRL [209], DIAYN [207], VALOR [206], DISCERN [215], Skew-Fit [216], VISR [211]
$p(z), p(s)$: fixed $p(s\|z)$, $p(z\|s)$: modelled with VI	Forward	EDL (ours)

Table 8.1: Types of methods depending on the considered generative model and the version of the mutual information (MI) being maximized. Distributions denoted by ρ are induced by running the policy in the environment, whereas p is used for the true and potentially unknown ones. The dependency of existing methods on $\rho_\pi(s|z)$ causes pathological training dynamics by letting the agent influence over the states considered in the optimization process. EDL relies on a fixed distribution over states $p(s)$ to break this dependency and makes use of variational inference (VI) techniques to model $p(s|z)$ and $p(z|s)$.

sampling states from a dataset of extrinsically generated experience [220]. Note that unlike approaches in imitation learning [221], learning from demonstrations [222, 223], and learning from play [224], EDL does not require access to trajectories or actions emitted by an expert policy.

Skill discovery. Whereas existing methods sample latents $z \sim p(z)$ directly as an input to the latent-conditioned policy, EDL requires an indirect approach wherein latent codes are inferred from $p(s)$. More concretely, given a distribution over states, or samples from it, we treat skill discovery as learning to model $p(z|s)$ and $p(s|z)$. We turn to variational inference techniques for this purpose, and Variational Autoencoders (VAEs) [225] in particular. Fortunately, we can approximate both distributions by training a VAE on samples from $p(s)$ – the encoder q_ψ models $p(z|s)$, whereas the decoder q_ϕ models $p(s|z)$. Intuitively, this process determines which latent codes are assigned to each region of the state space, and which states should be visited by each skill. The fact that exploration and skill discovery are disentangled enables learning variational posteriors for different $p(z)$ priors without needing to re-learn a new latent-conditioned policy every time. This is an interesting property, as the task of defining the prior over skills is not straightforward. In contrast, previous methods perform exploration and skill discovery at the same time, so that modifying $p(z)$ inevitably involves exploring the environment from scratch.

Skill learning. The final stage consists in training a policy $\pi_\theta(a|s, z)$ that maximizes the mutual information between states and latent variables. EDL adopts the forward form of the mutual information, and the reader is referred to Appendix B for a detailed explanation of this choice. Since $p(s)$ is fixed, Equation 8.7 can be maximized in a

reinforcement learning-styled setup with the reward function

$$r(s, z') = \log q_\phi(s|z'),\ z' \sim p(z) \tag{8.26}$$

where $q_\phi(s|z)$ is given by the decoder of the VAE trained on the skill discovery stage. This final stage can be seen as training a policy that mimics the decoder *within* the MDP, i.e. a policy that will visit the state that the decoder would generate for each latent code z. Note that the reward function is fixed, unlike that in previous methods which continuously changes depending on the behavior of the policy.

8.4 Experiments

Some previous works have evaluated skill discovery methods on complex environments, such as robotic locomotion [173] or 3D navigation [226], whose complexity renders policy learning difficult. This burden falls on the underlying RL algorithm, which needs to learn a more complicated policy in order to achieve the desired behavior. Note that this does not necessarily make the task of discovering options more difficult. As an example, consider the process of discovering useful locomotion skills. These options will likely require the agent to move in different directions, no matter if it is controlling a simple point mass or a complex humanoid.

We take a different approach and consider controlled synthetic environments. These are fully-continuous 2D mazes where the agent observes its current position and outputs actions that control its location, which is affected by collisions with walls. Varying the maze topology allows for an analysis of skill discovery methods in the face of specific challenges, providing insight on the properties and limitations of these algorithms. The topology is not given to the agent, which needs to infer the position of walls from interaction.

All experiments consider discrete priors over skills. This choice allows for a fair comparison between methods, as those based on the reverse form of the mutual information are not straightforward to combine with continuous priors. We consider the two methods described in Section 8.1 as baselines. The skill discovery stage in EDL is performed with a VQ-VAE [227] to handle the discrete prior, and the real-valued codes it discovers are used to condition the policy. We adopt the common distributional assumption for continuous data where $p(s|z)$ is Gaussian [225], which does not consider the actual connectivity within the MDP and results in reward functions that can become fraught with local optima. For this reason, we use Sibling Rivalry [228] to escape local optima during the skill learning stage in some environments. Note that the described implementation

is just a possible solution that follows the proposed paradigm, which is not limited to the specific choices made in our experimental setup.

Figures 8.1–8.6 report 20 rollouts per skill to account for the stochasticity of the policy. The initial state is denoted by a black dot and the color of the rollout denotes the skill upon which it was conditioned, thus figures are best viewed in color. All experiments consider agents that learn 10 skills, value that was selected to provide a good balance between learning a variety of behaviors and ease of visualization. When visualizing states visited by a random policy, we collect 100 rollouts with each (untrained) skill. Trajectories from these skills highly overlap with each other, so we use a single color for all of them to reduce clutter. We refer the reader to the appendix for a detailed description of the experimental setup and the hyperparameters.

Exploration with SMM. In the absence of any prior knowledge, we would like to discover skills across the whole state space by defining a uniform distribution over states, $p(s)$. In the controlled environments considered in this work, this can be achieved by sampling states from an oracle. In order to understand the impact of not having access to an oracle, we employ State Marginal Matching (SMM) [229] with a uniform target distribution to perform the exploration stage in EDL. Evaluation is performed on a simple maze where the forward and reverse baselines already fail to learn state-covering skills, as depicted in Figure 8.3 (top). In contrast, Figure 8.3 (bottom) shows how EDL can learn state-covering skills even in the realistic scenario where an oracle is not available[2].

Impact of the initial state. Baseline methods rely on $\rho_\pi(s|z)$ to perform skill discovery, which is initially induced by a random policy. This introduces a strong dependence on the distribution over initial states, $p(s_0)$. Changes to $p(s_0)$ might make some behaviors harder to learn, e.g. reaching a certain position becomes more difficult the further an agent spawns from it. A change in $p(s_0)$ should have little impact on what options are deemed important as long as all options are still achievable. We evaluate this phenomenon on two corridor-shaped mazes, which have the same topology but differ in the position of the initial state. We will refer to these environments as E_{center} and E_{left}, in which the agent spawns in the center and the left section of the corridor, respectively. Figure 8.4 (top) shows how the baselines discover completely different skills depending on $p(s_0)$. When replicating this experiment using EDL with SMM exploration, we get two different setups. Figure 8.4 (bottom left) shows the result of performing exploration and skill discovery in E_{center} and then learning skills in both E_{center} and E_{left}. Figure 8.4 (bottom right) depicts the impact of performing exploration and skill discovery in E_{left} instead. Skills

[2] We succeeded at training skills discovered by EDL without Sibling Rivalry. However, it greatly reduced the number of runs in the grid search that got trapped in local optima. The presented results used Sibling Rivalry to take advantage of this fact and reduce variance in the results.

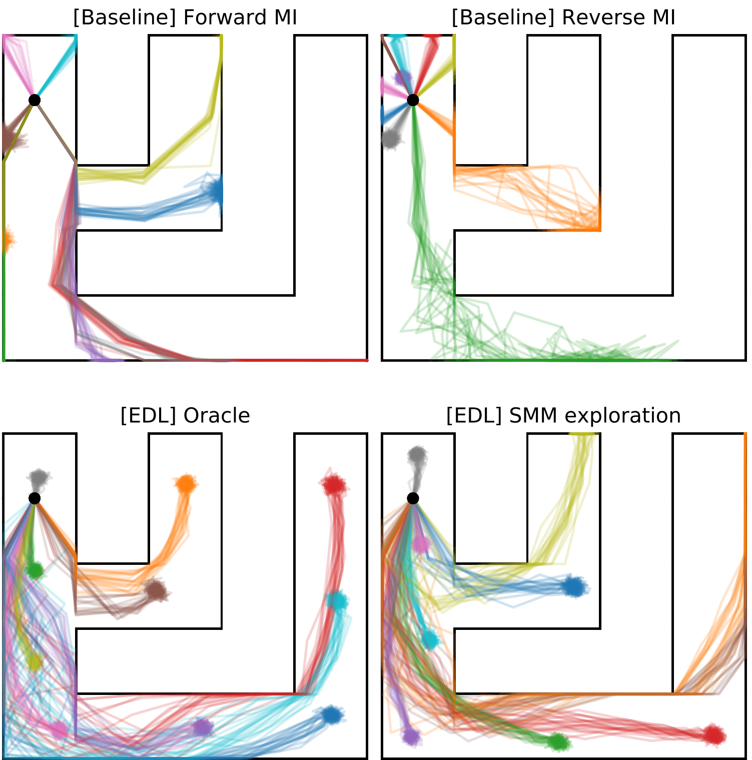


Figure 8.3: Impact of replacing the oracle with SMM in the exploration stage. **Top:** baselines fail at discovering skills that reach the right side of the maze. **Bottom:** EDL discovers skills that are spread across the whole maze, even when replacing the oracle with SMM. We observed that SMM tended to collect more samples near the walls, which explains the slight difference in the discovered options.

learned in both setups are very similar, with differences coming from the slightly different distribution over states collected by SMM.

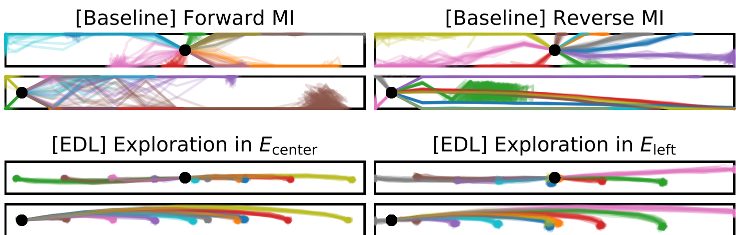


Figure 8.4: Impact of the distribution over initial states, $p(s_0)$. **Top:** baselines are very sensitive to $p(s_0)$ and discover very different skills depending on this distribution. **Bottom:** we report two different experiments with EDL. The setup on the left performs exploration and skill discovery in E_{center} and then learns skills in both E_{center} and E_{left}. The one on the right performs exploration and skill discovery in E_{left} instead. Options discovered by EDL are very similar in both setups.

Encouraging specific behaviors. In many settings, the user has some knowledge about which areas of state space will be most relevant for downstream tasks. Inducing proper priors might help to overcome the curse of dimensionality in complex environments, where most skills discovered under a uniform prior might not be useful for the tasks we are interested in. Existing methods can leverage prior knowledge by maximizing

$I(f(S); Z)$ instead of $I(S; Z)$, where $f(S)$ is a function of the states. For instance, this function can compute the center of mass of a robot in order to encourage the discovery of locomotion skills [207]. However, this method fails at incorporating more complex priors, such as encouraging the agent to only learn locomotion skills that move in specific directions or where the center of mass needs to be above a certain height to ensure that the robot does not fall down. EDL offers more flexibility for leveraging priors through the definition of $p(s)$, e.g. by drawing samples from a dataset of human play [220]. We simulate this scenario by performing skill discovery with an oracle that samples states uniformly from a subset of the state space. Figure 8.5 reports results in a tree-shaped maze, where we introduce the prior that skills should visit the right side of the maze only. EDL effectively incorporates this prior, and learns state-covering skills in its absence.

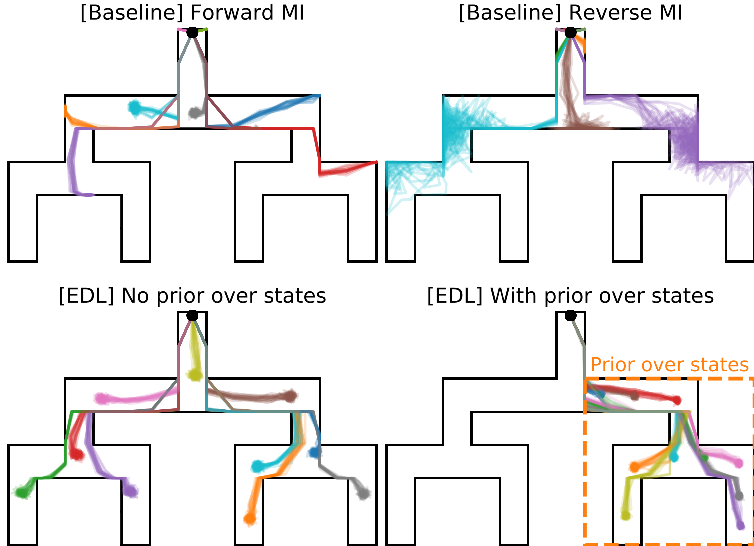


Figure 8.5: Incorporating priors over skills, where we are interested in learning skills on the right side of the maze. **Top:** this type of prior cannot be incorporated into baseline methods, whose discovered options are agnostic to it. **Bottom:** in the absence of a prior, EDL learns options across the whole state space. When incorporating the prior, the agent devotes all its capacity to learning skills within the region of interest.

Impact of bottleneck states. The maze with bottleneck states from Figure 8.1, where baseline approaches fail to explore a large extent of the state space, is a challenging environment where the limitations of EDL can be evaluated. We were unable to explore this type of maze effectively with SMM. Given that SMM relies on a curiosity-like bonus [229], we attribute this failure to well-known issues of these methods such as derailment and detachment [230]. Note that these problems are related to the sub-optimality of the density estimation method and RL solver, as shown by the bounds derived by Hazan et al. [219]. In light of this, we rely on an oracle to simulate perfect exploration and evaluate the skill discovery and learning stages of EDL. On this maze, the reward functions that EDL introduces create deceptive local optima in which policies tend to get stuck

(c.f. Figure 8.10). Sibling Rivalry proved crucial to avoid these failures, and allowed the policy to learn the skills depicted in Figure 8.1 (bottom right). These observations suggest that the main bottlenecks for the proposed approach to skill discovery are maximum entropy exploration and avoiding local optima when learning to maximize the EDL reward (Equation 8.26). Given that EDL decouples the process in three stages, advances in these fields are straightforward to incorporate and will boost the performance of this type of option discovery method.

Interpolating between skills. The skill discovery stage in EDL with a categorical prior $p(z)$ can be seen as the process of learning a discrete number of goals, together with an embedded representation for each of them. In the experimental setup presented in this work, each embedded representation corresponds to one of the continuous vectors in the VQ-VAE's codebook, z_i, whereas each goal state is given by $g_i = \text{argmax}_s q_\phi(s|z_i)$. The idea of goal embeddings was introduced as part the Universal Value Function Approximator (UVFA) framework [231]. UVFAs can generalize to unseen goals, and here we explore how the policies learned by EDL generalize to unseen latent codes z – where we construct new codes by interpolating the ones discovered by EDL. The results in Figure 8.6 suggest that the policy learns to generalize, with interpolated skills reaching states that come from the interpolation in Euclidean space of the goals of the original skills.

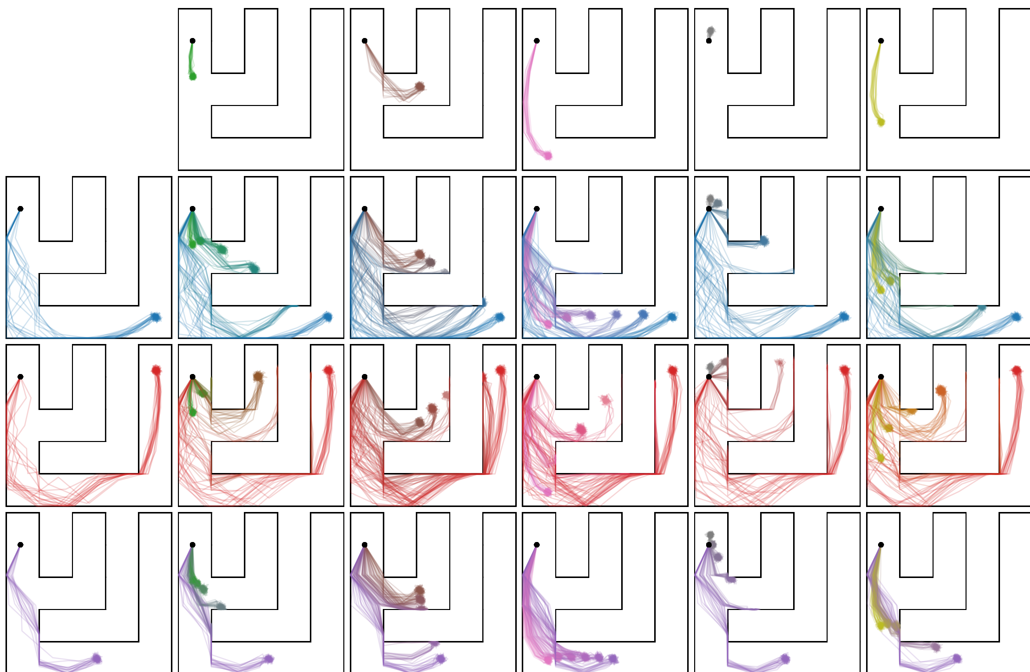


Figure 8.6: Interpolating skills learned by EDL. Interpolation is performed at the latent variable level by blending the z vector of two skills. The first row and column show the original skills being interpolated, which were selected randomly from the set of learned options. When plotting interpolated skills, we blend the colors used for the original skills.

Additional visualizations. We include visualizations that provide further insight about the results in Figure 8.1. These include the goal states discovered by methods using the forward form of the mutual information (Figure 8.7), and a visualization of the reward landscape of each method (Figures 8.8, 8.9 and 8.10).

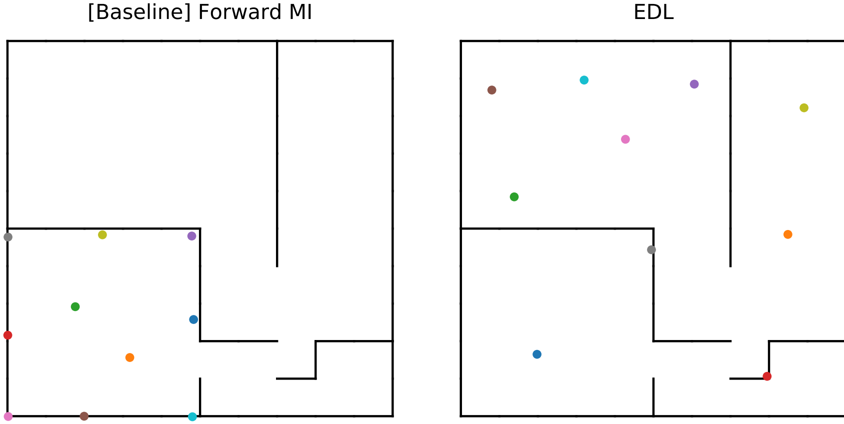


Figure 8.7: Goal states discovered by methods using the forward form of the mutual information in Figure 8.1. We define a goal state as the most likely state under $q_\phi(s|z)$ for each skill, i.e. $g_i = \text{argmax}_s q_\phi(s|z_i)$. The baseline method relies on the stationary state-distribution induced by the policy to discover goals. This policy seldom leaves the initial room, limiting the goals that can be discovered. In contrast, the uniform distribution over states in EDL enables the discovery of goals across the whole maze.

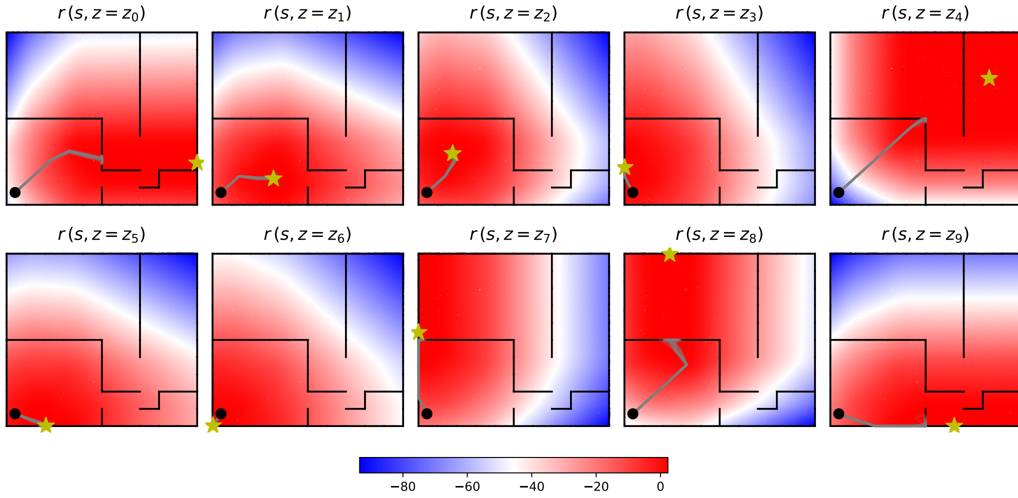


Figure 8.8: Reward landscape per skill at convergence for the forward MI agent in Figure 8.1. Trajectories from each skill starting from the black dot are plotted in gray. The yellow star indicates the point of maximum reward for each skill. For some skills, this point belongs to an unexplored region of the state space, contrary to the intuition in Section 8.2. Note that this is due to the Gaussian assumption over $p(s|z)$ in the density model.

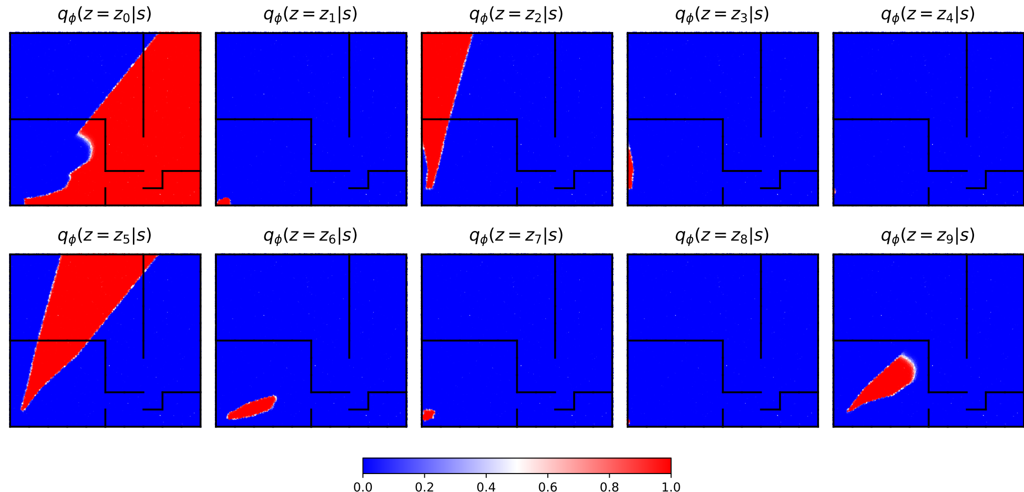


Figure 8.9: Approximate posterior $q_\phi(z|s)$ at convergence for the reverse MI agent in Figure 8.1. Recall that the reward function for this agent is $r(s,z) = \log q_\phi(z|s) - \log p(z)$, and $\log p(z)$ is constant in our experiments due to the choice of prior over latent variables. The state space is partitioned in disjoint regions, so that skills only need to enter their corresponding region in order to maximize reward. Note how $q_\phi(z|s)$ extrapolates this partition to states that have never been visited by the policy. When combined with an entropy bonus, this reward landscape results in skills that produce highly entropic trajectories within each region.

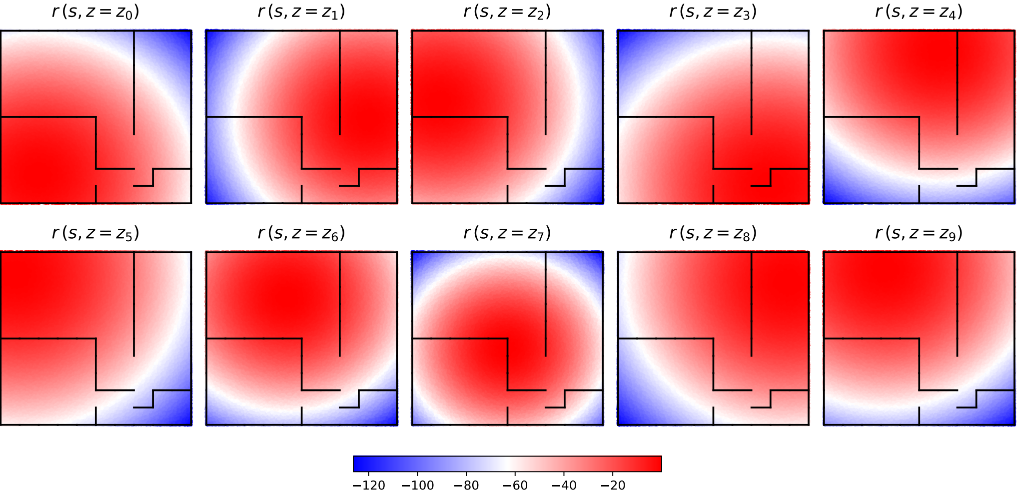


Figure 8.10: Reward landscape per skill at convergence for the EDL agent in Figure 8.1. The reward functions follow a bell shape centered at each of the centroids in Figure 8.7 (right). These are dense signals that ease optimization, but training is prone to falling in local optima due to their deceptive nature.

8.5 Related Work

Option discovery. Temporally-extended high-level primitives, also known as options, are an important resource in the RL toolbox [204, 232, 233]. The process of defining options involves task-specific knowledge, which might be difficult to acquire and has motivated research towards methods that automatically discover such options. These include

learning options while solving the desired task [234], leveraging demonstrations [235], training goal-oriented low-level policies [236], and meta-learning primitives from a distribution of related tasks [237]. Skills discovered by information-theoretic methods have also been used as primitives for Hierarchical RL [207, 209, 217].

Intrinsic rewards. Agents need to encounter a reward before they can start learning, but this process might become highly inefficient in sparse reward setups when relying on standard exploration techniques [238]. This issue can be alleviated by introducing intrinsic rewards, i.e. denser reward signals that can be automatically computed. These rewards are generally task-agnostic and might come from state visitation pseudo-counts [239, 240], unsupervised control tasks [241], learning to predict environment dynamics [242–244], self-imitation [245], and self-play [246, 247].

Novelty Search. Discovering a set of diverse and task-agnostic behaviors in the absence of a fitness function has been explored in the evolutionary computation community [248, 249]. Quality Diversity algorithms aim at combining the best of both worlds by optimizing task-specific fitness functions while encouraging diverse behaviors in a population of agents [250–252]. These methods rely on a behavior characterization function, which is tasked with summarizing the behavior of an agent into a vector representation. There have been efforts towards learning such functions [253], but it is still a common practice for practitioners to design a different function for each task [192].

Goal-oriented RL. The standard RL framework can be extended to consider policies and reward functions that are conditioned on some goal $g \in \mathcal{G}$ [231]. Given a known distribution over goals $p(g)$ that the agent should achieve, this setup allows for efficient training techniques involving experience relabeling [179] and reward shaping [228]. Defining such distribution requires expert domain knowledge, an assumption that is not always fulfilled. As a result, methods that can learn to reach *any* given state have garnered research interest [215, 216]. These approaches can be seen as skill discovery algorithms where $\mathcal{Z} = \mathcal{S}$, i.e. where each goal state defines a different skill. This raises the question of whether methods that can reach any state are superior to those learning a handful of skills. We argue that the latter offer important benefits in terms of exploration when used by a meta-controller to solve downstream tasks. When $p(z)$ is a simple distribution, the meta-controller benefits from a reduced search space, which is one of the motivations behind building hierarchies and options [233]. On the other hand, exploring with state-reaching policies involves a search space of size $|\mathcal{S}|$. This figure will quickly increase as the complexity of the environment grows, making exploration inefficient. Moreover, this setup assumes that the meta-controller is able to sample from $\mathcal{S}$ in order to emit goals for the goal-conditioned policy.

8.6 Discussion

We provided theoretical and empirical evidence that poor state space coverage is a predominant failure mode of existing skill discovery methods. The information-theoretic objective requires access to unknown distributions, which these methods approximate with those induced by the policy. These approximations lead to pathological training dynamics where the agent obtains larger rewards by visiting already discovered states rather than exploring the environment. We proposed EDL, a novel option discovery approach that leverages a fixed distribution over states and variational inference techniques to break the dependency on the distributions induced by the policy. Importantly, this alternative approach optimizes the same objective derived from information theory used in previous methods. EDL succeeds at discovering state-covering skills in environments where previous methods failed. It offers additional advantages, such as being more robust to changes in the distribution of the initial state and enabling the user to incorporate priors over which behaviors are considered useful. Our experiments suggest that EDL discovers a meaningful latent space for skills even when tasked with learning a discrete set of options, whose latent codes can be combined in order to produce a richer set of behaviors.

The proposed EDL paradigm is not limited to the implementation considered in this work. Each of the three stages of the method poses its own challenges, but can benefit from advances in their respective research directions as well. This modular design allows us to incorporate to our implementation recent advances such as exploration with SMM [229], vector quantization techniques to impose discrete priors in VAEs [227], and relabeling techniques to optimize deceptive reward functions [228]. Future breakthroughs in these directions could contribute towards scaling up skill discovery methods to richer environments, potentially leading to the emergence of complex behaviors [254].

There are several research directions to be explored in future work. Improvements in pure exploration methods would make EDL applicable to a broader range of environments. Despite the existence of strong theoretical results [219], these approaches involve the optimization of reward functions that are challenging for current algorithms [230]. In our experiments, we adopted the common distributional assumption for continuous data where $p(s|z)$ is Gaussian [225]. This assumption was responsible for the deceptive reward functions discovered in our experiments, and might be detrimental in some other environments. This motivates research towards discovering embedding spaces for states where distances are related to the closeness of states within the MDP [255], and learning reward functions that reflect similarity in controllable aspects of the environment [215].

Finally, leveraging information-theoretic methods to perform unsupervised task discovery in the meta-RL framework [256] is another interesting direction for future research.

9 Conclusion

The last decade has seen extraordinary progress in machine learning applications, driven by the fast pace of advances in deep learning research. Approaches based on representations derived from expert knowledge have been replaced by systems that can learn tasks end to end, mapping raw signals to the desired output space. The quest towards solving problems in an end to end fashion produced a first wave of groundbreaking advances in domains where enough training data was available, including computer vision [6], speech recognition [37], machine translation [257], and control [174]. While neural networks had been used way before the recent advent of deep learning [53], the key factor behind their current success is *scale*. In particular, this has been possible thanks to the development of algorithms that can benefit from increased compute and data, and this dissertation focused on pushing the limits of scale in these two axes.

The first research direction considered in this thesis was concerned with developing algorithms that can make the most of the available hardware. Given that the current trend in high performance computing consists in accelerating workloads by distributing them across devices, this required studying algorithms that are inherently parallel. When learning from examples, Chapter 4 explored strategies for distributing training of Convolutional Neural Networks (CNNs) in a homogeneous GPU cluster in order to shorten training times. This enables faster iteration, which accelerates progress in both research and industrial applications. When learning from interaction, Chapter 7 introduced a technique to improve the data efficiency of Evolution Strategies (ES), a method that scales gracefully to thousands of CPU cores. Importantly, these gains are achieved without compromising the scalability of the original method.

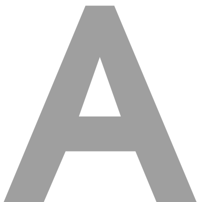
The second research direction in this dissertation explored the design of methods that can scale up with the amount of available data. This is a broad research question, and we considered three relevant problems within this area of research. Chapter 5 introduced a novel Recurrent Neural Network architecture that is able to solve tasks involving sequences while ignoring some of the input elements. Skipping patterns are decided based on the data seen so far, which allows training models for different computational budgets that learn how to make the most efficient usage of the available resources. In Chapter 6, we derived a novel initialization for a particular class of neural networks that enables robust training of very deep models. This is a very important direction, as increasing the depth of neural networks allows training complex models on large collections of data, but depth often leads to unstable training. Finally, Chapter 8 analyzed the limitations of existing unsupervised skill discovery methods within the Reinforcement Learning (RL) framework and proposed a novel method to overcome them. The proposed method leads to the discovery of skills that provide a better coverage of what is possible in the environment, and thus are more likely to be useful for solving downstream tasks. Providing agents with the capability of autonomously acquiring useful and reusable knowledge, as opposed to maximizing handcrafted reward functions, is key towards scaling up RL to complex domains.

Unprecedented results have been obtained when pushing the limits of scale. AlphaGo [258], a system combining model-based planning with neural networks, defeated the world champion the game of Go – one of the grand challenges in artificial intelligence research. Arguably, its most important component was *self-play*, which let the agent play against itself in order to generate training data. This strategy enabled trading off large amounts of compute for data, quickly accumulating the equivalent to years of human experience. In a similar fashion, the very large-scale language models by OpenAI have shown outstanding zero-shot [198] and few-shot [11] learning capabilities after ingesting a virtually unlimited amount of data downloaded from the Internet. Despite the observation that model quality grows sub-linearly with increased size, the limits of scale have yet to be found. Continuous improvements have been obtained so far by training larger models whenever enough training data and compute have become available. However, it is reasonable to ask whether intelligence can be solved through scale only. Will we ever build a dataset which contains everything needed for intelligence to emerge?

In spite of recent advances, all modern artificial intelligence solutions still belong to the category of *narrow artificial intelligence* – methods that aim at solving a single task. This type of systems already enabled numerous applications with which we interact in our daily lives, such as voice assistants, predictive typing, translators or image retrieval systems. In general, having mastered one task will not help these systems in solving a new problem – which contrasts with how humans can leverage prior knowledge and

abilities when solving previously unseen problems. For instance, this means that an agent controlling a robot will need to learn that objects fall, or that it cannot walk through walls, every single time it needs to solve a new task. Solving the problem of *transfer*, i.e. designing machines with the ability to accumulate and leverage prior knowledge efficiently, is one of the grand challenges in artificial intelligence. Overcoming this limitation will allow our agents to quickly become competent in new tasks for which little data is available, likely creating a new wave of artificial intelligence advances.

Learning-based methods have revolutionized the field of artificial intelligence, with capabilities far beyond those of expert systems. They expand the types of problems that can be addressed with artificial intelligence, relaxing the requirements from being able to formalize a solution to simply providing feedback on the results. This setting pushes the bottleneck towards our own ability for designing objectives and feedback signals. Creating supervisory signals is an arduous process, which explains why only a handful of them are available for current datasets or simulated environments. Moreover, they are limited by our own creativity and needs. What if our agents could define their own goals out of curiosity? This would definitely help in scaling up the number of tasks our agents can solve, and likely encouraging them to build a better model of the world in the process. From a pragmatic view, this would lead to more efficient learning systems that can extract more bits of information from the available data. However, this might have even more profound consequences – what if artificial intelligence systems could find answers to questions we have not even thought of yet?



Qualitative Results for Skip RNN

This appendix contains additional qualitative results for the Skip RNN models.

A.1 Adding Task

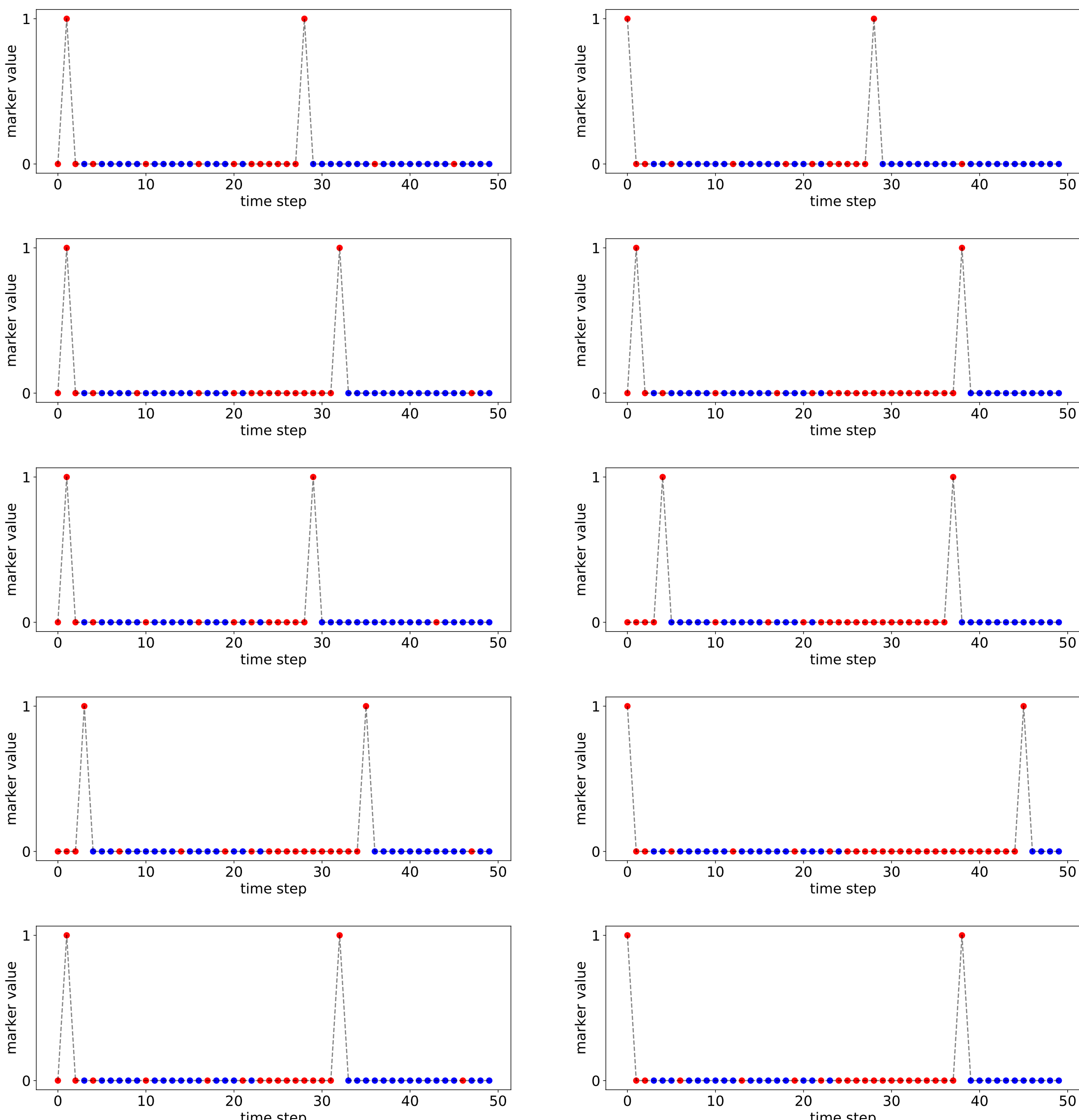


Figure A.1: Sample usage examples for the Skip GRU with $\lambda = 10^{-5}$ on the adding task. Red dots indicate used samples, whereas blue ones are skipped.

A.2 Frequency Discrimination Task

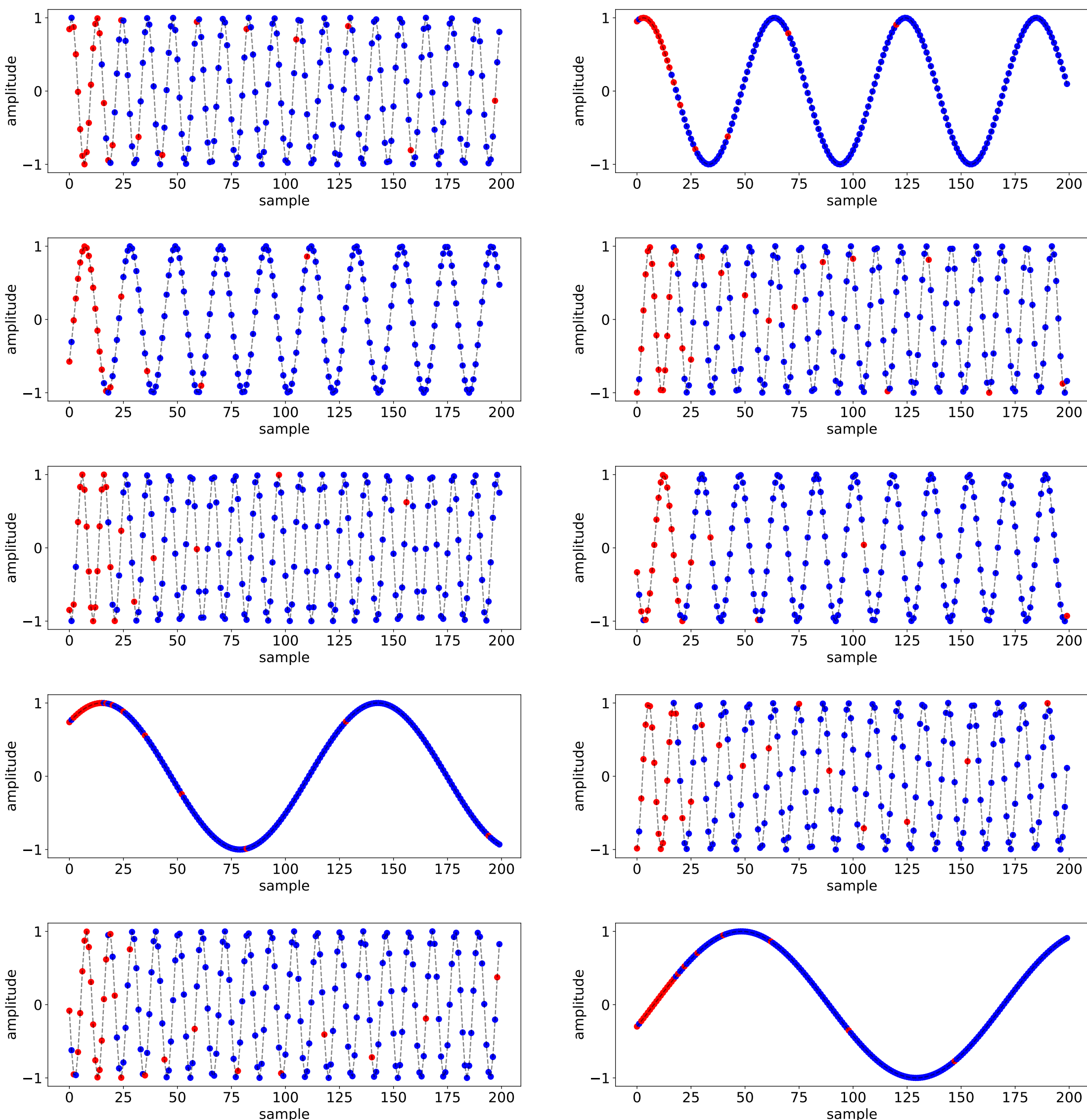


Figure A.2: Sample usage examples for the Skip LSTM with $\lambda = 10^{-4}$ on the frequency discrimination task with $T_s = 0.5$ms. Red dots indicate used samples, whereas blue ones are skipped. The network learns that using the first samples is enough to classify the frequency of the sine waves, in contrast to a uniform downsampling that may result in aliasing.

B

Choice of Mutual Information's Form for EDL

The main novelty of EDL is an alternative for modelling the unknown distributions, which in principle could work with either form of the mutual information. For the sake of comparison with previous works, all experiments consider discrete skills. This was achieved through a categorical posterior $p(z|s)$ that was approximated with a VQ-VAE [227]. The encoder of the VQ-VAE takes an input x, produces output $z_e(x)$, and maps it to the closest element in the codebook, $e \in \mathbb{R}^{K \times D}$. The posterior categorical distribution $q(z|x)$ probabilities are defined as one-hot as follows:

$$q(z=k|x) = \begin{cases} 1 & \text{for } k = \text{argmin}_j ||z_e(x) - e_j||_2 \\ 0 & \text{otherwise} \end{cases} \tag{B.1}$$

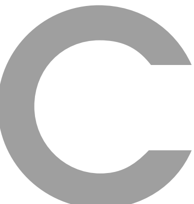
One could consider the reverse form of the mutual information and train the policy with a reward function as follows:

$$r(s,z) = q(z|s) \tag{B.2}$$

where we assumed a uniform prior over z and removed the constant $\log p(z)$ term from the reward.

We can foresee two issues with this reward function. It is sparse, i.e. many states provide no reward at all, which might hinder training unless proper exploration strategies are used [228, 230]. A similar behavior was observed in existing methods using the reverse form of the mutual information (c.f. Figure 8.9). Moreover, the fact that many states

produce a maximum reward of 1 might lead to unpredictable skills when paired with an entropy bonus. Such unpredictability might not be desirable when training a meta-controller to solve a downstream task by combining the learned skills [217].



Implementation Details

C.1 Robust Initialization for WeightNorm & ResNets

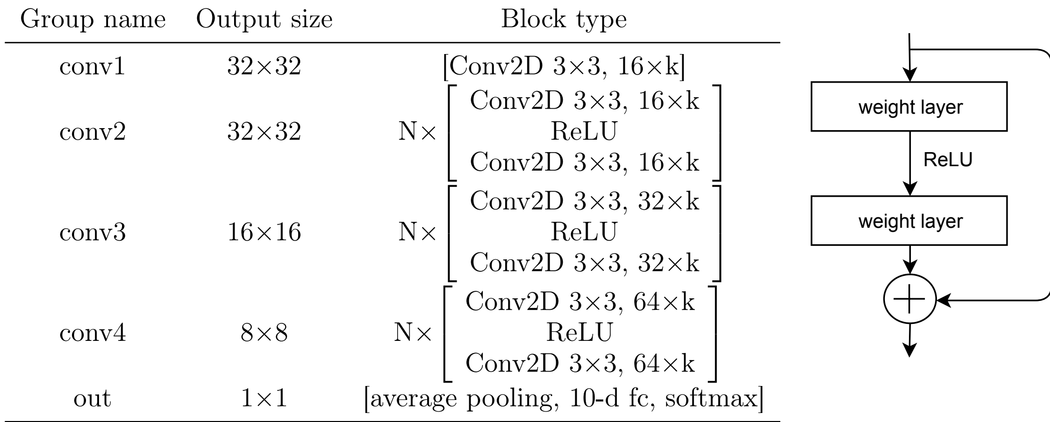
C.1.1 Synthetic Data

Feedforward networks. We use a weight normalized 20 layer Multilayer Perceptron with 1000 randomly generated input samples in $\mathbb{R}^{500}$. We test three initialization strategies. (1) He initialization [63] for the weight matrices and the gain parameter $\mathbf{g}$ for all layers are initialized to 1. (2) Proposed initialization, where weights are initialized to be orthogonal and gains are set as $\sqrt{2n_{l-1}/n_l}$. (3) Proposed initialization, where weights are initialized using He initialization and gains are set as $\sqrt{2n_{l-1}/n_l}$. In all cases biases are set to 0. At initialization itself, we forward propagate the 1000 randomly generated input samples, measure the norm of hidden activations, and compute the mean and standard deviation of the ratio of norm of hidden activation to the norm of the input. This is shown in Figure 6.1 (top left). In Figure 6.1 (top right), we similarly record the norm of hidden activation gradient by backpropagating 1000 random error vectors, and measure the ratio of the norm of hidden activation gradient to the norm of the error vector. We find that the proposed initialization preserves norm in both directions while vanilla He initialization fails. This shows the importance of proper initialization of the γ parameter of weight normalization.

Residual networks. We use a weight normalized residual network with 40 residual blocks with 1000 randomly generated input samples in $\mathbb{R}^{500}$. The network architecture is exactly as described in Equation 6.6, with a residual block composed of two Fully

Connected (FC) layers, i.e. FC1 → ReLU → FC2. The weight normalization layers are inserted after FC layers. We test three initialization strategies. (1) He initialization [63] for all the weight matrices, and gain parameter $\mathbf{g} = \mathbf{1}$. (2) Proposed initialization where weights are initialized to be orthogonal and gains are set as $\sqrt{2 \cdot \text{fan-in}/\text{fan-out}}$ for FC1 and $\sqrt{\text{fan-in}/(40 \cdot \text{fan-out})}$ for FC2. (3) Proposed initialization where weights are initialized using He initialization and gains are set same as in the previous case. In all cases biases are set to 0. At initialization itself, we forward propagate the 1000 randomly generated input samples, measure the norm of hidden activations $\mathbf{h}^b$ and compute the mean and standard deviation of the ratio of norm of hidden activation to the norm of the input $\mathbf{x}$. This is shown in Figure 6.1 (bottom left). In Figure 6.1 (bottom right), we similarly record the norm of hidden activation gradient by backpropagating 1000 random error vectors and measure the ratio of the norm of hidden activation gradient $\frac{\partial \ell}{\partial \mathbf{h}^b}$ to the norm of the error vector $\frac{\partial \ell}{\partial \mathbf{h}^B}$. We find that the proposed initialization preserves norm in both directions while vanilla He initialization fails. This shows the importance of proper initialization of the $\mathbf{g}$ parameter of weight normalization.

C.1.2 Residual Network Architecture



Group name	Output size	Block type
conv1	32×32	[Conv2D 3×3, 16×k]
conv2	32×32	N× [Conv2D 3×3, 16×k; ReLU; Conv2D 3×3, 16×k]
conv3	16×16	N× [Conv2D 3×3, 32×k; ReLU; Conv2D 3×3, 32×k]
conv4	8×8	N× [Conv2D 3×3, 64×k; ReLU; Conv2D 3×3, 64×k]
out	1×1	[average pooling, 10-d fc, softmax]

Figure C.1: **Left:** Architecture of Wide Residual Networks (WRNs) considered in this work. Downsampling is performed through strided convolutions by the first layers in groups `conv3` and `conv4`. **Right:** Structure of a residual block. Note that there is no non-linearity after residual connections, unlike He et al. [63].

C.1.3 MNIST

Parameter	**Value**
Data split	10% of the original train is set aside for validation purposes
Number of hidden layers	$\{2, 5, 10, 20, 100, 200\}$
Size of hidden layers	$\{512, 1024\}$
Number of epochs	150
Initial learning rate*	$\{0.1, 0.01, 0.001, 0.0001, 0.00001\}$
Learning rate schedule	Decreased by 10× at epochs 50 and 100
Batch size	128
Weight decay	0.0001
Optimizer	SGD with momentum = 0.9

Table C.1: Hyperparameters for MNIST experiments. Values between brackets were used in the grid search. Learning rate of 0.00001 was considered for depths 100 and 200 only.

C.1.4 CIFAR

Parameter	**Value**
Data split	10% of the original train is set aside for validation purposes
Architecture	$2 \times$ [Conv2D $3 \times 3/2, 512$] $(\mathrm{N} - 2) \times$ [Conv2D $3 \times 3/1, 512$] Global Average Pooling 10-d Linear, softmax
Number of hidden layers (N)	$\{5, 25, 100\}$
Number of epochs	500
Initial learning rate	$\{0.01, 0.001^*\}$
Learning rate schedule	Decreased by 10× at epoch 166
Batch size	100
Weight decay	$\{0.001, 0.0001\}$
Optimizer	SGD without momentum

Table C.2: Hyperparameters for CNN experiments on CIFAR-10. Values between brackets were used in the grid search. Learning rate of 0.001 was considered for depth 100 only.

Parameter	**Value**
Data split	10% of the original train is set aside for validation purposes
WRN's N (residual blocks per stage)	$\{1, 16, 166, 1666\}$
WRN's k (width factor)	1
Number of epochs	1
Initial learning rate*	$\{7 \cdot 10^{-1}, 3 \cdot 10^{-1}, 1 \cdot 10^{-1}, \ldots, 10^{-5}, \ldots, 10^{-7}\}$
Batch size	128
Weight decay	0.0005
Optimizer	SGD with momentum = 0.9

Table C.3: Hyperparameters for WRN experiments on CIFAR-10. Values between brackets were used in the grid search. Learning rates smaller than 10^{-5} were considered for $N = 1666$ (10,000 layers) only.

C.2 Unsupervised Discovery of State-Covering Skills

C.2.1 Environments

The maze environments are adapted from the open-source implementation[1] by Trott et al. [228]. The agent does not observe the walls, whose location needs to be inferred from experience and makes exploration difficult. The initial state for each episode is sampled from a 1×1 tile. See Table C.4 for details about the environments and the topology of each maze.

Parameter	**Value**
State space	$\mathcal{S} \in \mathbb{R}^2$
Action space	$\mathcal{A} \in [-0.95, 0.95]^2$
Episode length	50
Size: Bottleneck maze (Figure 8.1)	10×10
Size: Square maze (Figure 8.3)	5×5
Size: Corridor maze (Figure 8.4)	1×12
Size: Tree maze (Figure 8.5)	7×7

Table C.4: Environment details.

C.2.2 RL Agents

Policy networks emit the parameters of a Beta distribution [259], which are then shifted and scaled to match the task action range. Entropy regularization is employed to prevent convergence to deterministic behaviors early in training. We use a categorical distribution with uniform probabilities for the skill prior $p(z)$. Agents are trained with PPO [175] and the Adam optimizer [43]. Hyperparameters are tuned for each method independently using a grid search. See Table C.5 for details.

[1] https://github.com/salesforce/sibling-rivalry

Hyperparameter	**Value**
Discount factor	0.99
λ_{GAE}	0.98
λ_{entropy}	$\{0.001, 0.005, 0.01, 0.025\}$
$\epsilon_{\text{SiblingRivalry}}$	$\{2.5, 5.0, 7.5\}$
Optimizer	Adam
Learning rate	$\{0.0003, 0.001\}$
Learning rate schedule	Constant
Advantage normalization	Yes
Input normalization	{Yes, No}
Hidden layers	2
Units per layer	128
Non-linearity	ReLU
Horizon	2500
Batch size	250
Number of epochs	4

Table C.5: Hyperparameters used in the experiments. Values between brackets were used in the grid search, and tuned independently for each method.

C.2.3 Exploration

When relying on State Marginal Matching (SMM) [229] for exploration, we implement the version that considers a mixture of policies with a uniform target distribution $p^*(s)$. The density model $q(s)$ is approximated with a Variational Autoencoder (VAE). We use states in the replay buffer as a non-parametric approach to sampling from the desired $p(s)$ [215]. Sampling states from the replay buffer is similar to a uniform Historical Averaging strategy. This worked well in our experiments, but exponential sampling strategies might be needed in other environments to avoid oversampling states collected by the initially random policies [219]. Our implementation follows the open-source code released by the authors[2], which relies on SAC [260] for policy optimization. Hyperparameters are tuned for each environment independently using a grid search. See Table C.6 for details.

[2] https://github.com/RLAgent/state-marginal-matching

Hyperparameter	Value
Discount factor	0.99
Target smoothing coefficient	0.005
Target update interval	1
α_{entropy}	$\{0.1, 1, 10\}$
β_{VAE}	$\{0.01, 0.1, 1\}$
Optimizer	Adam
Policy: Learning rate	0.001
SMM discriminator: Learning rate	0.001
VAE: Learning rate	0.01
Learning rate schedule	Constant
Policies in the mixture	4
Input normalization	No
Policy: Hidden layers	2
SMM discriminator: Hidden layers	2
VAE encoder: Hidden layers	2
VAE decoder: Hidden layers	2
Units per layer	128
Non-linearity	ReLU
Gradient steps	1
Batch size	128
Replay buffer size	50k

Table C.6: Hyperparameters used for exploration using SMM. Values between brackets were used in the grid search, and tuned independently for each environment. Training ends once the buffer is full.

C.2.4 Skill Discovery

The skill discovery stage in the proposed method is done with a VQ-VAE [227], which allows learning discrete latents. We implement the version that relies on a commitment loss to learn the dictionary. The size of the codebook is set to the number of desired skills. Hyperparameters are tuned for each environment and exploration method independently using a grid search. See Table C.7 for details.

Hyperparameter	**Value**
Code size	16
$\beta_{\text{commitment}}$	$\{0.25, 0.5, 0.75, 1.0, 1.25\}$
Optimizer	Adam
Learning rate	0.0002
Learning rate schedule	Constant
Batch size	256
Number of samples	4096
Input normalization	Yes
Encoder: Hidden layers	2
Decoder: Hidden layers	2
Units per layer	128
Non-linearity	ReLU

Table C.7: Hyperparameters used for training the VQ-VAE in the skill discovery stage. Values between brackets were used in the grid search, and tuned independently for each environment and exploration method.

D

Proofs

Theorem 1. *Let* $\mathbf{v} = \mathrm{ReLU}\left(\sqrt{\frac{2n}{m}} \cdot \hat{\mathbf{R}}\mathbf{u}\right)$, *where* $\mathbf{u} \in \mathbb{R}^n$ *and* $\hat{\mathbf{R}} \in \mathbb{R}^{m \times n}$. *If* $\mathbf{R}_i \overset{i.i.d.}{\sim} P$ *where* P *is any isotropic distribution in* $\mathbb{R}^n$, *or alternatively* $\hat{\mathbf{R}}$ *is a randomly generated matrix with orthogonal rows, then for any fixed vector* $\mathbf{u}$, $\mathbb{E}[\|\mathbf{v}\|^2] = K_n \cdot \|\mathbf{u}\|^2$ *where,*

$$K_n = \begin{cases} \frac{2S_{n-1}}{S_n} \cdot \left(\frac{2}{3} \cdot \frac{4}{5} \cdots \frac{n-2}{n-1}\right) & \text{if } n \text{ is odd} \\ \frac{2S_{n-1}}{S_n} \cdot \left(\frac{1}{2} \cdot \frac{3}{4} \cdots \frac{n-2}{n-1}\right) \cdot \frac{\pi}{2} & \text{otherwise} \end{cases} \tag{D.1}$$

and S_n *is the surface are of a unit* n*-dimensional sphere.*

Proof: *During the proof, take note of the distinction between the notations* $\hat{\mathbf{R}}_i$ *and* $\mathbf{R}_i$. *Our goal is to compute,*

$$\mathbb{E}[\|\mathbf{v}\|^2] = \mathbb{E}[\sum_{i=1}^{m} v_i^2] \tag{D.2}$$

$$= \sum_{i=1}^{m} \mathbb{E}[v_i^2] \tag{D.3}$$

Suppose the weights are randomly generated to be orthogonal with uniform probability over all rotations. Due to the linearity of expectation, when taking the expectation of any unit v_i *over the randomly generated orthogonal weight matrix, the expectation marginalizes over all the rows of the weight matrix except the* i^{th} *row. As a consequence, for each unit* i, *the expectation is over an isotropic random variable since the orthogonal matrix is generated randomly with uniform probability over all rotations. Therefore, we can*

equivalently write,

$$\mathbb{E}[\|\mathbf{v}\|^2] = m\mathbb{E}[v_i^2] \tag{D.4}$$

Note that the above equality would trivially hold if all rows of the weight matrix were sampled i.i.d. from an isotropic distribution. In other words, the above equality holds irrespective of the two choice of distributions used for sampling the weight matrix.

We have,

$$\mathbb{E}[v_i^2] = \mathbb{E}[\max(0, \sqrt{\frac{2n}{m}} \cdot \hat{\mathbf{R}}_i^T \mathbf{u})^2] \tag{D.5}$$

$$= \int_{\mathbf{R}_i} p(\mathbf{R}_i) \max(0, \sqrt{\frac{2n}{m}} \cdot \|\mathbf{u}\| \cos\theta)^2 \tag{D.6}$$

where $p(\mathbf{R}_i)$ denotes the probability distribution of the random variable $\mathbf{R}_i$, and θ is the angle between vectors $\hat{\mathbf{R}}_i$ and $\mathbf{u}$. Hence θ is a function of $\hat{\mathbf{R}}_i$. Since $\mathbf{R}_i$ is sampled from an isotropic distribution, the direction and scale of $\mathbf{R}_i$ are independent. Thus,

$$\int p(\mathbf{R}_i) \max(0, \sqrt{\frac{2n}{m}} \cdot \|\mathbf{u}\| \cos\theta)^2 = \int_{\mathbf{R}_i} p(\|\mathbf{R}_i\|) \int_{\hat{\mathbf{R}}_i} p(\hat{\mathbf{R}}_i) \max(0, \sqrt{\frac{2n}{m}} \cdot \|\mathbf{u}\| \cos\theta)^2 \tag{D.7}$$

$$= \int_{\hat{\mathbf{R}}_i} p(\hat{\mathbf{R}}_i) \max(0, \sqrt{\frac{2n}{m}} \cdot \|\mathbf{u}\| \cos\theta)^2 \tag{D.8}$$

$$= \frac{2n}{m} \cdot \|\mathbf{u}\|^2 \int_{\hat{\mathbf{R}}_i} p(\hat{\mathbf{R}}_i) \max(0, \cos\theta)^2 \tag{D.9}$$

Since P is an isotropic distribution in $\mathbb{R}^n$, the likelihood of all directions is uniform. It essentially means that $p(\hat{\mathbf{R}}_i)$ can be seen as a uniform distribution over the surface area of a unit n-dimensional sphere. We can therefore re-parameterize $p(\hat{\mathbf{R}}_i)$ in terms of θ by aggregating the density $p(\hat{\mathbf{R}}_i)$ over all points on this n-dimensional sphere at a fixed angle θ from the vector $\mathbf{u}$. This is similar to the idea of Lebesgue integral. To achieve this, we note that all the points on the n-dimensional sphere at a constant angle θ from $\mathbf{u}$ lie on an $(n-1)$-dimensional sphere of radius $\sin\theta$. Thus the aggregate density at an angle θ from $\mathbf{u}$ is the ratio of the surface area of the $(n-1)$-dimensional sphere of radius

$\sin\theta$ *and the surface area of the unit* (n)*-dimensional sphere. Therefore,*

$$\int_{\hat{\mathbf{R}}_i} p(\hat{\mathbf{R}}_i)\max(0,\cos\theta)^2 = \int_0^{\pi} \frac{S_{n-1}}{S_n}\cdot|\sin^{n-1}\theta|\cdot\max(0,\cos\theta)^2 \tag{D.10}$$

$$= \frac{S_{n-1}}{S_n}\int_0^{\pi/2}\sin^{n-1}\theta\cos^2\theta \tag{D.11}$$

$$= \frac{S_{n-1}}{S_n}\int_0^{\pi/2}\sin^{n-1}\theta(1-\sin^2\theta) \tag{D.12}$$

$$= \frac{S_{n-1}}{S_n}\int_0^{\pi/2}\sin^{n-1}\theta-\sin^{n+1}\theta \tag{D.13}$$

Now we use a known result in existing literature that uses integration by parts to evaluate the integral of exponentiated sine function, which states,

$$\int\sin^n\theta = -\frac{1}{n}\sin^{n-1}\theta\cos\theta+\frac{n-1}{n}\int\sin^{n-2}\theta \tag{D.14}$$

Since our integration is between the limits 0 *and* $\pi/2$*, we find that the first term on the R.H.S. in the above expression is 0. Recursively expanding the* $n-2^{th}$ *power sine term, we can similarly eliminate all such terms until we are left with the integral of* $\sin\theta$ *or* $\sin^0\theta$ *depending on whether n is odd or even. For the case when n is odd, we get,*

$$\int_0^{\pi/2}\sin^n\theta = \left(\frac{2}{3}\cdot\frac{4}{5}\ldots\frac{n-1}{n}\right)\int_0^{\pi/2}\sin\theta \tag{D.15}$$

$$= -\left(\frac{2}{3}\cdot\frac{4}{5}\ldots\frac{n-1}{n}\right)\cos\theta|_0^{\pi/2} \tag{D.16}$$

$$= \left(\frac{2}{3}\cdot\frac{4}{5}\ldots\frac{n-1}{n}\right) \tag{D.17}$$

For the case when n is even, we similarly get,

$$\int_0^{\pi/2}\sin^n\theta = \left(\frac{1}{2}\cdot\frac{3}{4}\cdot\frac{5}{6}\ldots\frac{n-1}{n}\right)\int_0^{\pi/2}\sin^0\theta \tag{D.18}$$

$$= \left(\frac{1}{2}\cdot\frac{3}{4}\cdot\frac{5}{6}\ldots\frac{n-1}{n}\right)\int_0^{\pi/2}1 \tag{D.19}$$

$$= \left(\frac{1}{2}\cdot\frac{3}{4}\cdot\frac{5}{6}\ldots\frac{n-1}{n}\right)\cdot\frac{\pi}{2} \tag{D.20}$$

Thus,

$$\int_0^{\pi/2}\sin^{n-1}\theta-\sin^{n+1}\theta = \begin{cases}\frac{1}{n}\cdot\left(\frac{2}{3}\cdot\frac{4}{5}\ldots\frac{n-2}{n-1}\right) & \text{if } n \text{ is odd}\\ \frac{1}{n}\cdot\left(\frac{1}{2}\cdot\frac{3}{4}\ldots\frac{n-2}{n-1}\right)\cdot\frac{\pi}{2} & \text{otherwise}\end{cases} \tag{D.21}$$

Define,

$$K_n = \begin{cases} \frac{2S_{n-1}}{S_n} \cdot \left(\frac{2}{3} \cdot \frac{4}{5} \cdots \frac{n-2}{n-1}\right) & \text{if } n \text{ is odd} \\ \frac{2S_{n-1}}{S_n} \cdot \left(\frac{1}{2} \cdot \frac{3}{4} \cdots \frac{n-2}{n-1}\right) \cdot \frac{\pi}{2} & \text{otherwise} \end{cases} \tag{D.22}$$

Then,

$$\int_{\hat{\mathbf{R}}_i} p(\hat{\mathbf{R}}_i) \max(0, \cos\theta)^2 = \frac{0.5K_n}{n} \tag{D.23}$$

Thus,

$$\mathbb{E}[\|\mathbf{v}\|^2] = m\mathbb{E}[v_i^2] \tag{D.24}$$

$$= m \cdot \frac{2n}{m} \cdot \|\mathbf{u}\|^2 \cdot \frac{0.5K_n}{n} \tag{D.25}$$

$$= K_n \cdot \|\mathbf{u}\|^2 \tag{D.26}$$

which proves the claim. □

Lemma 1. *If network weights are sampled i.i.d. from a Gaussian distribution with mean 0 and biases are 0 at initialization, then conditioned on $\mathbf{h}^{l-1}$, each dimension of $\mathbb{1}(\mathbf{a}^l)$ follows an i.i.d. Bernoulli distribution with probability 0.5 at initialization.*

Proof: *Note that $\mathbf{a}^l := \mathbf{W}^l\mathbf{h}^{l-1}$ at initialization (biases are 0) and $\mathbf{W}^l$ are sampled i.i.d. from a random distribution with mean 0. Therefore, each dimension $\mathbf{a}_i^l$ is simply a weighted sum of i.i.d. zero mean Gaussian, which is also a 0 mean Gaussian random variable.*

To prove the claim, note that the indicator operator applied on a random variable with 0 mean and symmetric distribution will have equal probability mass on both sides of 0, which is the same as a Bernoulli distributed random variable with probability 0.5. Finally, each dimension of $\mathbf{a}^l$ is i.i.d. simply because all the elements of $\mathbf{W}^l$ are sampled i.i.d., and hence each dimension of $\mathbf{a}^l$ is a weighted sum of a different set of i.i.d. random variables. □

Theorem 2. *Let $\mathbf{v} = \sqrt{2} \cdot \left(\hat{\mathbf{R}}^T\mathbf{u}\right) \odot \mathbf{z}$, where $\mathbf{u} \in \mathbb{R}^m$, $\mathbf{R} \in \mathbb{R}^{m \times n}$ and $\mathbf{z} \in \mathbb{R}^n$. If each $\mathbf{R}_i \overset{i.i.d.}{\sim} P$ where P is any isotropic distribution in $\mathbb{R}^n$ or alternatively $\hat{\mathbf{R}}$ is a randomly generated matrix with orthogonal rows and $\mathbf{z}_i \overset{i.i.d.}{\sim}$ Bernoulli(0.5), then for any fixed vector $\mathbf{u}$, $\mathbb{E}[\|\mathbf{v}\|^2] = \|\mathbf{u}\|^2$.*

Proof: *Our goal is to compute,*

$$\mathbb{E}[\|\mathbf{v}\|^2] = 2 \cdot \mathbb{E}[\|(\sum_{i=1}^{n} \hat{\mathbf{R}}_i u_i) \odot \mathbf{z}\|^2] \tag{D.27}$$

$$= 2 \cdot \mathbb{E}[\sum_{j=1}^{m} (\sum_{i=1}^{n} \hat{\mathbf{R}}_{ij} u_i)^2 \cdot z_j^2] \tag{D.28}$$

$$= 2 \cdot \mathbb{E}[z_j^2] \cdot \mathbb{E}[\sum_{j=1}^{m} (\sum_{i=1}^{n} \hat{\mathbf{R}}_{ij} u_i)^2] \tag{D.29}$$

$$= \mathbb{E}[\|(\sum_{i=1}^{n} \hat{\mathbf{R}}_i u_i)\|^2] \tag{D.30}$$

$$= \mathbb{E}[\sum_{i=1}^{n} u_i^2 \|\hat{\mathbf{R}}_i\|^2 + \sum_{i \neq j} u_i u_j \cdot \hat{\mathbf{R}}_i^T \hat{\mathbf{R}}_j] \tag{D.31}$$

$$= \|\mathbf{u}\|^2 + \sum_{i \neq j} u_i u_j \cdot \mathbb{E}[\hat{\mathbf{R}}_i^T \hat{\mathbf{R}}_j] \tag{D.32}$$

$$= \|\mathbf{u}\|^2 + \sum_{i \neq j} u_i u_j \cdot \mathbb{E}[\cos\phi] \tag{D.33}$$

where ϕ is the angle between $\hat{\mathbf{R}}_i$ and $\hat{\mathbf{R}}_j$. For orthogonal matrix $\hat{\mathbf{R}}$ $\cos\phi$ is always 0, while for $\hat{\mathbf{R}}$ such that each $\mathbf{R}_i \overset{i.i.d.}{\sim} P$ where P is any isotropic distribution, $\mathbb{E}[\cos\phi] = 0$. Thus for both cases[1] *we have that,*

$$\mathbb{E}[\|\mathbf{v}\|^2] = \|\mathbf{u}\|^2 \tag{D.34}$$

which proves the claim. □

Theorem 3. *Let $\mathcal{R}(\{F_b(\cdot)\}_{b=0}^{B-1}, \theta, \alpha)$ be a residual network with output $f_\theta(\cdot)$. Assume that each residual block $F_b(\cdot)$ $(\forall b)$ is designed such that at initialization, $\|F_b(\mathbf{h})\| = \|\mathbf{h}\|$ for any input $\mathbf{h}$ to the residual block, and $< \mathbf{h}, F_b(\mathbf{h}) > \approx 0$. If we set $\alpha = 1/\sqrt{B}$, then,*

$$\|f_\theta(\mathbf{x})\|^2 \approx c \cdot \|\mathbf{x}\|^2 \tag{D.35}$$

where $c \in [\sqrt{2}, \sqrt{e}]$.

Proof: *Let $\mathbf{x}$ denote the input of the residual network. Consider the first hidden state $\mathbf{h}^1$ given by,*

$$\mathbf{h}^1 := \mathbf{x} + \alpha F_1(\mathbf{x}) \tag{D.36}$$

[1] This also suggests that orthogonal initialization is strictly better than Gaussian initialization since the result holds without the dependence on expectation in contrast to the Gaussian case.

Then the squared norm of $\mathbf{h}^1$ *is given by,*

$$\|\mathbf{h}^1\|^2 = \|\mathbf{x} + \alpha F_1(\mathbf{x})\|^2 \tag{D.37}$$

$$= \|\mathbf{x}\|^2 + \alpha^2 \|F_1(\mathbf{x})\|^2 + 2\alpha < \mathbf{x}, F_1(\mathbf{x}) > \tag{D.38}$$

Since $\|\mathbf{F}_1(\mathbf{x})\|^2 = \|\mathbf{x}\|^2$ *and* $< \mathbf{x}, F_1(\mathbf{x}) > \approx 0$ *due to our assumptions, we have,*

$$\|\mathbf{h}^1\|^2 \approx \|\mathbf{x}\|^2 \cdot (1 + \alpha^2) \tag{D.39}$$

Similarly,

$$\mathbf{h}^2 := \mathbf{h}^1 + \alpha F_2(\mathbf{h}^1) \tag{D.40}$$

Thus,

$$\|\mathbf{h}^2\|^2 = \|\mathbf{h}^1\|^2 + \alpha^2 \|F_2(\mathbf{h}^1)\|^2 + 2\alpha < \mathbf{h}^1, F_2(\mathbf{h}^1) > \tag{D.41}$$

Then due to our assumptions we get,

$$\|\mathbf{h}^2\|^2 \approx \|\mathbf{h}^1\|^2 \cdot (1 + \alpha^2) \tag{D.42}$$

Thus we get,

$$\|\mathbf{h}^2\|^2 \approx \|\mathbf{x}\|^2 \cdot (1 + \alpha^2)^2 \tag{D.43}$$

Extending such inequalities to the Bth *residual block, we get,*

$$\|\mathbf{h}^B\|^2 \approx \|\mathbf{x}\|^2 \cdot (1 + \alpha^2)^B \tag{D.44}$$

Setting $\alpha = 1/\sqrt{B}$*, we get,*

$$\|\mathbf{h}^B\|^2 \approx \|\mathbf{x}\|^2 \cdot \left(1 + \frac{1}{B}\right)^B \tag{D.45}$$

Note that the factor $\left(1 + \frac{1}{B}\right)^B \to e$ *as* $B \to \infty$ *due to the following well known result,*

$$\lim_{B \to \infty} \left(1 + \frac{1}{B}\right)^B = e \tag{D.46}$$

Since $B \in \mathbb{Z}$, $\left(1 + \frac{1}{B}\right)^{B/2}$ *lies in* $[\sqrt{2}, \sqrt{e}]$.

Thus we have proved the claim. $\square$

Theorem 4. *Let $\mathcal{R}(\{F_b(\cdot)\}_{b=0}^{B-1}, \theta, \alpha)$ be a residual network with output $f_\theta(\cdot)$. Assume that each residual block $F_b(\cdot)$ $(\forall b)$ is designed such that at initialization, $\|\frac{\partial F_b(\mathbf{h}^b)}{\partial \mathbf{h}^b}\mathbf{u}\| = \|\mathbf{u}\|$ for any fixed input $\mathbf{u}$ of appropriate dimensions, and $< \frac{\partial \mathcal{L}}{\partial \mathbf{h}^b}, \frac{\partial \mathbf{F}_{b-1}}{\partial \mathbf{h}^{b-1}} \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{h}_b} > \approx 0$. If $\alpha = \frac{1}{\sqrt{B}}$, then,*

$$\|\frac{\partial \mathcal{L}}{\partial \mathbf{h}^1}\| \approx c \cdot \|\frac{\partial \mathcal{L}}{\partial \mathbf{h}^B}\| \tag{D.47}$$

where $c \in [\sqrt{2}, \sqrt{e}]$.

Proof: *Recall,*

$$\mathbf{h}^b := \mathbf{x} + \alpha F_b(\mathbf{h}^{b-1}) \tag{D.48}$$

Therefore, taking derivative on both sides,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{b-1}} = (\mathbf{I} + \alpha \cdot \frac{\partial F_b}{\partial \mathbf{h}^{b-1}}) \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{h}^b} \tag{D.49}$$

$$= \frac{\partial \mathcal{L}}{\partial \mathbf{h}^b} + \alpha \cdot \frac{\partial F_b}{\partial \mathbf{h}^{b-1}} \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{h}^b} \tag{D.50}$$

Taking norm on both sides,

$$\|\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{b-1}}\|^2 = \|\frac{\partial \mathcal{L}}{\partial \mathbf{h}^b}\|^2 + \alpha^2 \cdot \|\frac{\partial F_b}{\partial \mathbf{h}^{b-1}} \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{b-1}}\|^2 + 2\alpha \cdot < \frac{\partial \mathcal{L}}{\partial \mathbf{h}^b}, \frac{\partial F_b}{\partial \mathbf{h}^{b-1}} \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{b-1}} > \tag{D.51}$$

Due to our assumptions, we have,

$$\|\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{b-1}}\|^2 \approx \|\frac{\partial \mathcal{L}}{\partial \mathbf{h}^b}\|^2 + \alpha^2 \cdot \|\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{b-1}}\|^2 \tag{D.52}$$

$$= (1 + \alpha^2) \cdot \|\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{b-1}}\|^2 \tag{D.53}$$

Applying this result to all B residual blocks we have that,

$$\|\frac{\partial \mathcal{L}}{\partial \mathbf{h}^1}\|^2 \approx (1 + \alpha^2)^B \cdot \|\frac{\partial \mathcal{L}}{\partial \mathbf{h}^B}\|^2 \tag{D.54}$$

Setting $\alpha = 1/\sqrt{B}$, we get,

$$\|\frac{\partial \mathcal{L}}{\partial \mathbf{h}^1}\|^2 \approx (1 + 1/B)^B \cdot \|\frac{\partial \mathcal{L}}{\partial \mathbf{h}^B}\|^2 \tag{D.55}$$

Note that the factor $\left(1 + \frac{1}{B}\right)^B \to e$ as $B \to \infty$ due to the following well known result,

$$\lim_{B \to \infty} \left(1 + \frac{1}{B}\right)^B = e \tag{D.56}$$

Since $B \in \mathbb{Z}$, $\left(1 + \frac{1}{B}\right)^{B/2}$ lies in $[\sqrt{2}, \sqrt{e}]$.

Thus we have proved the claim. □

Bibliography

[1] Alan Turing. Computing machinery and intelligence-am turing. *Mind*, 1950.

[2] John McCarthy, Marvin L Minsky, Nathaniel Rochester, and Claude E Shannon. A proposal for the Dartmouth summer research project on artificial intelligence, August 31, 1955. *AI magazine*, 2006.

[3] Ada Lovelace. Sketch of the analytical engine invented by Charles Babbage, 1842.

[4] Richard Sutton. The bitter lesson. *http://http://www.incompleteideas.net/*, 2019. URL http://incompleteideas.net/IncIdeas/BitterLesson.html.

[5] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *CVPR*, 2009.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *NIPS*, 2012.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, 2015.

[8] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. Imagenet training in minutes. In *ICPP*, 2018.

[9] Dhruv Mahajan, Ross Girshick, Vignesh Ramanathan, Kaiming He, Manohar Paluri, Yixuan Li, Ashwin Bharambe, and Laurens van der Maaten. Exploring the limits of weakly supervised pretraining. In *ECCV*, 2018.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*, 2019.

[11] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell,

et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[12] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. In *ICLR*, 2018.

[13] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures. In *ICML*, 2018.

[14] Víctor Campos, Francesc Sastre, Maurici Yagües, Míriam Bellver, Xavier Giró-i-Nieto, and Jordi Torres. Distributed training strategies for a computer vision deep learning algorithm on a distributed GPU cluster. *Procedia Computer Science*, 2017.

[15] Víctor Campos, Brendan Jou, Xavier Giró-i-Nieto, Jordi Torres, and Shih-Fu Chang. Skip RNN: Learning to skip state updates in recurrent neural networks. In *ICLR*, 2018.

[16] Devansh Arpit*, Víctor Campos*, and Yoshua Bengio. How to initialize your network? Robust initialization for WeightNorm & ResNets. In *NeurIPS*, 2019.

[17] Víctor Campos, Xavier Giró-i-Nieto, and Jordi Torres. Importance Weighted Evolution Strategies. In *NeurIPS Deep RL Workshop*, 2018.

[18] Víctor Campos, Alexander Trott, Caiming Xiong, Richard Socher, Xavier Giró-i-Nieto, and Jordi Torres. Explore, Discover and Learn: Unsupervised discovery of state-covering skills. In *ICML*, 2020.

[19] Víctor Campos, Brendan Jou, and Xavier Giró-i-Nieto. From pixels to sentiment: Fine-tuning CNNs for visual sentiment prediction. *Image and Vision Computing*, 2017.

[20] Xunyu Lin, Víctor Campos, Xavier Giró-i-Nieto, Jordi Torres, and Cristian Canton Ferrer. Disentangling motion, foreground and background features in videos. In *CVPR Brave New Motion Representations Workshop*, 2017.

[21] Víctor Campos, Francesc Sastre, Maurici Yagües, Jordi Torres, and Xavier Giró-i-Nieto. Scaling a convolutional neural network for classification of adjective noun pairs with tensorflow on gpu clusters. In *CCGRID*, 2017.

[22] Dèlia Fernández, Alejandro Woodward, Víctor Campos, Xavier Giró-i-Nieto, Brendan Jou, and Shih-Fu Chang. More cat than cute?: Interpretable prediction of adjective-noun pairs. In *ACM MM MUSA Workshop*, 2017.

[23] Daniel Fojo, Víctor Campos, and Xavier Giró-i-Nieto. Comparing fixed and adaptive computation time for recurrent neural networks. In *ICLR Workshop Track*, 2018.

[24] Amaia Salvador, Míriam Bellver, Víctor Campos, Manel Baradad, Ferran Marqués, Jordi Torres, and Xavier Giró-i-Nieto. Recurrent neural networks for semantic instance segmentation. In *CVPR DeepVision Workshop*, 2018.

[25] Víctor Campos, Xavier Giró-i-Nieto, Brendan Jou, Jordi Torres, and Shih-Fu Chang. Sentiment concept embedding for visual affect recognition. In *Multimodal Behavior Analysis in the Wild*. Elsevier, 2019.

[26] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2020.

[27] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[28] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 1989.

[29] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. In *NIPS*, 2014.

[30] Paul Bloom. *How children learn the meanings of words*. MIT press, 2002.

[31] Lechao Xiao, Yasaman Bahri, Jascha Sohl-Dickstein, Samuel S Schoenholz, and Jeffrey Pennington. Dynamical isometry and a mean field theory of cnns: How to train 10,000-layer vanilla convolutional neural networks. In *ICML*, 2018.

[32] Yoon Kim. Convolutional neural networks for sentence classification. In *EMNLP*, 2014.

[33] Ossama Abdel-Hamid, Abdel-rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on audio, speech, and language processing*, 2014.

[34] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *CVPR*, 2015.

[35] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

[36] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. In *ICLR*, 2015.

[37] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *ICASSP*, 2013.

[38] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 1997.

[39] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *EMNLP*, 2014.

[40] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Neurocomputing: foundations of research. chapter: Learning internal representations by error propagation, 1988.

[41] Yu Nesterov. A method of solving a convex programming problem with convergence rate o(1/sqrt(k)). In *Soviet Mathematics Doklady*, 1983.

[42] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 2012.

[43] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[44] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR*, 2014.

[45] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[46] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *ICLR*, 2016.

[47] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *ICLR*, 2016.

[48] David Ha. A visual guide to evolution strategies. *blog.otoro.net*, 2017. URL https://blog.otoro.net/2017/10/29/visual-evolution-strategies/.

[49] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[50] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[51] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[52] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[53] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

[54] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *CVPR*, 2014.

[55] NVIDIA. NVIDIA Tesla V100 GPU architecture. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2018.

[56] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *NIPS*, 2012.

[57] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *ICLR*, 2017.

[58] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous SGD. *arXiv preprint arXiv:1604.00981*, 2016.

[59] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[60] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010.

[61] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

[62] Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *NIPS*, 2016.

[63] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[64] CUDA Nvidia. Compute unified device architecture programming guide. 2007.

[65] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[66] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *ACM MM*, 2014.

[67] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I Jordan. Sparknet: Training deep networks in spark. *arXiv preprint arXiv:1511.06051*, 2015.

[68] He Ma, Fei Mao, and Graham W Taylor. Theano-mpi: a theano-based distributed training framework. *arXiv preprint arXiv:1605.08325*, 2016.

[69] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI symposium at USENIX*, 2016.

[70] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[71] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

[72] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[73] Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep learning with elastic averaging SGD. In *NIPS*, 2015.

[74] S Sundhar Ram, Angelia Nedic, and Venugopal V Veeravalli. Asynchronous gossip algorithms for stochastic optimization. In *GameNets*, 2009.

[75] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns. In *Interspeech*, 2014.

[76] Rosalind W. Picard. *Affective Computing*, volume 252. MIT Press Cambridge, 1997.

[77] Daniel McDuff, Rana El Kaliouby, Jeffrey F Cohn, and Rosalind W Picard. Predicting ad liking and purchase intent: Large-scale analysis of facial responses to ads. *IEEE Transactions on Affective Computing*, 2015.

[78] Robert Plutchik. *Emotion: A Psychoevolutionary Synthesis*. Harper & Row, 1980.

[79] Michel Cabanac. What is emotion? *Behavioural processes*, 2002.

[80] Bo Pang and Lillian Lee. Opinion mining and sentiment analysis. *Information Retrieval*, 2008.

[81] Damian Borth, Rongrong Ji, Tao Chen, Thomas Breuel, and Shih-Fu Chang. Large-scale visual sentiment ontology and detectors using adjective noun pairs. In *ACM MM*, 2013.

[82] Brendan Jou, Tao Chen, Nikolaos Pappas, Miriam Redi, Mercan Topkara, and Shih-Fu Chang. Visual affect around the world: A large-scale multilingual visual sentiment ontology. In *ACM MM*, 2015.

[83] Tao Chen, Damian Borth, Trevor Darrell, and Shih-Fu Chang. DeepSentiBank: Visual sentiment concept classification with deep convolutional neural networks. *arXiv:1410.8586*, 2014.

[84] Brendan Jou and Shih-Fu Chang. Going deeper for multilingual visual sentiment detection. *arXiv preprint arXiv:1605.09211*, 2016.

[85] Damian Borth, Tao Chen, Rongrong Ji, and Shih-Fu Chang. Sentibank: large-scale ontology and classifiers for detecting sentiment and emotions in visual content. In *ACM MM*, 2013.

[86] BSC Support Team. Greasy user guide. https://github.com/jonarbo/GREASY/blob/master/doc/greasy_userguide.pdf, 2012.

[87] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *NIPS*, 2014.

[88] Peter H Jin, Qiaochu Yuan, Forrest Iandola, and Kurt Keutzer. How to scale distributed deep learning? *arXiv preprint arXiv:1611.04581*, 2016.

[89] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[90] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 1994.

[91] Daniel Neil, Michael Pfeiffer, and Shih-Chii Liu. Phased LSTM: accelerating recurrent network training for long or event-based sequences. In *NIPS*, 2016.

[92] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[93] Yoshua Bengio. Deep learning of representations: Looking forward. In *SLSP*, 2013.

[94] Andrew Davis and Itamar Arel. Low-rank approximations for conditional feedforward computation in deep neural networks. *arXiv preprint arXiv:1312.4461*, 2013.

[95] Lanlan Liu and Jia Deng. Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. *arXiv preprint arXiv:1701.00299*, 2017.

[96] Amjad Almahairi, Nicolas Ballas, Tim Cooijmans, Yin Zheng, Hugo Larochelle, and Aaron Courville. Dynamic capacity networks. In *ICML*, 2016.

[97] Mason McGill and Pietro Perona. Deciding how to decide: Dynamic routing in artificial neural networks. In *ICML*, 2017.

[98] Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

[99] Junyoung Chung, Sungjin Ahn, and Yoshua Bengio. Hierarchical multiscale recurrent neural networks. In *ICLR*, 2017.

[100] Yacine Jernite, Edouard Grave, Armand Joulin, and Tomas Mikolov. Variable computation in recurrent neural networks. In *ICLR*, 2017.

[101] Jan Koutnik, Klaus Greff, Faustino Gomez, and Juergen Schmidhuber. A clockwork rnn. In *ICML*, 2014.

[102] David Krueger, Tegan Maharaj, János Kramár, Mohammad Pezeshki, Nicolas Ballas, Nan Rosemary Ke, Anirudh Goyal, Yoshua Bengio, Hugo Larochelle, Aaron Courville, et al. Zoneout: Regularizing rnns by randomly preserving hidden activations. In *ICLR*, 2017.

[103] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *ECCV*, 2016.

[104] Volodymyr Mnih, Nicolas Heess, Alex Graves, et al. Recurrent models of visual attention. In *NIPS*, 2014.

[105] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *ICML*, 2015.

[106] Jimmy Ba, Volodymyr Mnih, and Koray Kavukcuoglu. Multiple object recognition with visual attention. *arXiv preprint arXiv:1412.7755*, 2014.

[107] Gunnar A Sigurdsson, Xinlei Chen, and Abhinav Gupta. Learning visual storylines with skipping recurrent neural networks. In *ECCV*, 2016.

[108] Subhabrata Bhattacharya, Felix X Yu, and Shih-Fu Chang. Minimally needed evidence for complex event recognition in unconstrained videos. In *ICMR*, 2014.

[109] Serena Yeung, Olga Russakovsky, Greg Mori, and Li Fei-Fei. End-to-end learning of action detection from frame glimpses in videos. In *CVPR*, 2016.

[110] Yu-Chuan Su and Kristen Grauman. Leaving some stones unturned: dynamic feature prioritization for activity detection in streaming video. In *ECCV*, 2016.

[111] Colin Raffel and Dieterich Lawson. Training a subsampling mechanism in expectation. In *ICLR Workshop Track*, 2017.

[112] James Bradbury, Stephen Merity, Caiming Xiong, and Richard Socher. Quasi-recurrent neural networks. In *ICLR*, 2017.

[113] Tao Lei and Yu Zhang. Training rnns as fast as cnns. *arXiv preprint arXiv:1709.02755*, 2017.

[114] Adams Wei Yu, Hongrae Lee, and Quoc V Le. Learning to skim text. In *ACL*, 2017.

[115] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 1992.

[116] Tim Cooijmans, Nicolas Ballas, César Laurent, Çağlar Gülçehre, and Aaron Courville. Recurrent batch normalization. In *ICLR*, 2017.

[117] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[118] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[119] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

[120] Geoffrey Hinton. Neural networks for machine learning. Coursera video lectures, 2012.

[121] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[122] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *ICML*, 2013.

[123] Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In *NIPS*, 2015.

[124] Quoc V Le, Navdeep Jaitly, and Geoffrey E Hinton. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*, 2015.

[125] Saizheng Zhang, Yuhuai Wu, Tong Che, Zhouhan Lin, Roland Memisevic, Ruslan R Salakhutdinov, and Yoshua Bengio. Architectural complexity measures of recurrent neural networks. In *NIPS*, 2016.

[126] Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks. In *ICML*, 2016.

[127] Andrew L Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *ACL*, 2011.

[128] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, 2013.

[129] Takeru Miyato, Andrew M Dai, and Ian Goodfellow. Adversarial training methods for semi-supervised text classification. In *ICLR*, 2017.

[130] Shayne Longpre, Sabeek Pradhan, Caiming Xiong, and Richard Socher. A way out of the odyssey: Analyzing and combining recent insights for lstms. *arXiv preprint arXiv:1611.05104*, 2016.

[131] Jeffrey Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. In *CVPR*, 2015.

[132] Joe Yue-Hei Ng, Matthew Hausknecht, Sudheendra Vijayanarasimhan, Oriol Vinyals, Rajat Monga, and George Toderici. Beyond short snippets: Deep networks for video classification. In *CVPR*, 2015.

[133] Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. Ucf101: A dataset of 101 human actions classes from videos in the wild. *arXiv preprint arXiv:1212.0402*, 2012.

[134] Evan Shelhamer, Kate Rakelly, Judy Hoffman, and Trevor Darrell. Clockwork convnets for video semantic segmentation. *arXiv preprint arXiv:1608.03609*, 2016.

[135] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. Learning spatiotemporal features with 3d convolutional networks. In *ICCV*, 2015.

[136] Karen Simonyan and Andrew Zisserman. Two-stream convolutional networks for action recognition in videos. In *NIPS*, 2014.

[137] Joao Carreira and Andrew Zisserman. Quo vadis, action recognition? a new model and the kinetics dataset. In *CVPR*, 2017.

[138] Gunnar Sigurdsson, Gül Varol, Xiaolong Wang, Ali Farhadi, Ivan Laptev, and Abhinav Gupta. Hollywood in homes: Crowdsourcing data collection for activity understanding. In *ECCV*, 2016.

[139] Gunnar A Sigurdsson, Santosh Divvala, Ali Farhadi, and Abhinav Gupta. Asynchronous temporal fields for action recognition. *arXiv preprint arXiv:1612.06371*, 2016.

[140] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[141] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 2014.

[142] Mohammad Sadegh Aliakbarian, Fatemehsadat Saleh, Mathieu Salzmann, Basura Fernando, Lars Petersson, and Lars Andersson. Encouraging LSTMs to anticipate actions very early. *arXiv preprint arXiv:1703.07023*, 2017.

[143] Boris Hanin and David Rolnick. How to start training: The effect of initialization and architecture. In *NeurIPS*, 2018.

[144] Boris Hanin. Which neural net architectures give rise to exploding and vanishing gradients? In *NeurIPS*, 2018.

[145] Stanislaw Jastrzebski, Zachary Kenton, Devansh Arpit, Nicolas Ballas, Asja Fischer, Yoshua Bengio, and Amos Storkey. Three factors influencing minima in sgd. *arXiv preprint arXiv:1711.04623*, 2017.

[146] Samuel L Smith and Quoc V Le. A bayesian perspective on generalization and stochastic gradient descent. In *ICLR*, 2018.

[147] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. In *ICLR*, 2018.

[148] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, 2010.

[149] Igor Gitman and Boris Ginsburg. Comparison of batch normalization and weight normalization algorithms for the large-scale image classification. *arXiv preprint arXiv:1709.08145*, 2017.

[150] Wenling Shang, Justin Chiu, and Kihyuk Sohn. Exploring normalization in deep residual networks with concatenated rectified linear units. In *AAAI*, 2017.

[151] Devansh Arpit, Yingbo Zhou, Bhargava U Kota, and Venu Govindaraju. Normalization propagation: A parametric technique for removing internal covariate shift in deep networks. In *ICML*, 2016.

[152] Dmytro Mishkin and Jiri Matas. All you need is a good init. *arXiv preprint arXiv:1511.06422*, 2015.

[153] Philipp Krähenbühl, Carl Doersch, Jeff Donahue, and Trevor Darrell. Data-dependent initializations of convolutional neural networks. *arXiv preprint arXiv:1511.06856*, 2015.

[154] Andrew M Saxe, James L McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. In *ICLR*, 2014.

[155] Ben Poole, Subhaneil Lahiri, Maithra Raghu, Jascha Sohl-Dickstein, and Surya Ganguli. Exponential expressivity in deep neural networks through transient chaos. In *NIPS*, 2016.

[156] Jeffrey Pennington, Samuel Schoenholz, and Surya Ganguli. Resurrecting the sigmoid in deep learning through dynamical isometry: theory and practice. In *NIPS*, 2017.

[157] Jeffrey Pennington, Samuel S Schoenholz, and Surya Ganguli. The emergence of spectral universality in deep networks. In *AISTATS*, 2018.

[158] Masato Taki. Deep residual networks and weight initialization. *arXiv preprint arXiv:1709.02956*, 2017.

[159] Wojciech Tarnowski, Piotr Warchoł, Stanisław Jastrzębski, Jacek Tabor, and Maciej A Nowak. Dynamical isometry is achieved in residual networks in a universal way for any activation function. *arXiv preprint arXiv:1809.08848*, 2018.

[160] Stanisław Jastrzębski, Devansh Arpit, Nicolas Ballas, Vikas Verma, Tong Che, and Yoshua Bengio. Residual connections encourage iterative inference. In *ICLR*, 2018.

[161] Hongyi Zhang, Yann N Dauphin, and Tengyu Ma. Fixup initialization: Residual learning without normalization. In *ICLR*, 2019.

[162] Terrance DeVries and Graham W Taylor. Improved regularization of convolutional neural networks with cutout. *arXiv preprint arXiv:1708.04552*, 2017.

[163] Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. In *ICLR*, 2018.

[164] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *BMVC*, 2016.

[165] Yann LeCun. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/, 1998.

[166] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

[167] Ping Luo, Xinjiang Wang, Wenqi Shao, and Zhanglin Peng. Towards understanding regularization in batch normalization. In *ICLR*, 2019.

[168] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016.

[169] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 2013.

[170] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *ICML*, 2015.

[171] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft II: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

[172] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 2017.

[173] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *IROS*, 2012.

[174] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 2015.

[175] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[176] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 2015.

[177] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017.

[178] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in Starcraft II using multi-agent reinforcement learning. *Nature*, 2019.

[179] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *NIPS*, 2017.

[180] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019.

[181] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

[182] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In *NIPS*, 2017.

[183] Martin Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom van de Wiele, Vlad Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by playing solving sparse reward tasks from scratch. In *ICML*, 2018.

[184] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *ICLR*, 2016.

[185] Shixiang Gu, Timothy Lillicrap, Zoubin Ghahramani, Richard E Turner, and Sergey Levine. Q-prop: Sample-efficient policy gradient with an off-policy critic. In *ICLR*, 2017.

[186] Ingo Rechenberg. Evolutionsstrategie–optimierung technisher systeme nach prinzipien der biologischen evolution. 1973.

[187] Daan Wierstra, Tom Schaul, Jan Peters, and Juergen Schmidhuber. Natural evolution strategies. In *WCCI*, 2008.

[188] Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural evolution strategies. *JMLR*, 2014.

[189] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[190] Paweł Wawrzyński. Real-time reinforcement learning by sequential actor–critics and experience replay. *Neural Networks*, 2009.

[191] Rémi Munos, Tom Stepleton, Anna Harutyunyan, and Marc Bellemare. Safe and efficient off-policy reinforcement learning. In *NIPS*, 2016.

[192] Edoardo Conti, Vashisht Madhavan, Felipe Petroski Such, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents. In *NIPS*, 2018.

[193] Nicolas Heess, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, Ali Eslami, Martin Riedmiller, et al. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*, 2017.

[194] Rein Houthooft, Richard Y Chen, Phillip Isola, Bradly C Stadie, Filip Wolski, Jonathan Ho, and Pieter Abbeel. Evolved policy gradients. *arXiv preprint arXiv:1802.04821*, 2018.

[195] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards.* PhD thesis, King's College, Cambridge, 1989.

[196] Doina Precup, Richard S Sutton, and Satinder Singh. Eligibility traces for off-policy policy evaluation. In *ICML*, 2001.

[197] Matteo Hessel, Hubert Soyer, Lasse Espeholt, Wojciech Czarnecki, Simon Schmitt, and Hado van Hasselt. Multi-task deep reinforcement learning with popart. *arXiv preprint arXiv:1809.04474*, 2018.

[198] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 2019.

[199] Olivier J Hénaff, Ali Razavi, Carl Doersch, SM Eslami, and Aaron van den Oord. Data-efficient image recognition with contrastive predictive coding. *arXiv preprint arXiv:1905.09272*, 2019.

[200] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. *arXiv preprint arXiv:1911.05722*, 2019.

[201] Christoph Salge, Cornelius Glackin, and Daniel Polani. Empowerment – an introduction. In *Guided Self-Organization: Inception*. Springer, 2014.

[202] Claude E Shannon. A mathematical theory of communication. *The Bell system technical journal*, 1948.

[203] Shakir Mohamed and Danilo Jimenez Rezende. Variational information maximisation for intrinsically motivated reinforcement learning. In *NIPS*, 2015.

[204] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 1999.

[205] Karol Gregor, Danilo Jimenez Rezende, and Daan Wierstra. Variational intrinsic control. *arXiv preprint arXiv:1611.07507*, 2016.

[206] Joshua Achiam, Harrison Edwards, Dario Amodei, and Pieter Abbeel. Variational option discovery algorithms. *arXiv preprint arXiv:1807.10299*, 2018.

[207] Benjamin Eysenbach, Abhishek Gupta, Julian Ibarz, and Sergey Levine. Diversity is all you need: Learning skills without a reward function. In *ICLR*, 2019.

[208] David Barber and Felix V Agakov. The IM algorithm: a variational approach to information maximization. In *NIPS*, 2003.

[209] Carlos Florensa, Yan Duan, and Pieter Abbeel. Stochastic neural networks for hierarchical reinforcement learning. In *ICLR*, 2017.

[210] Tuomas Haarnoja, Haoran Tang, Pieter Abbeel, and Sergey Levine. Reinforcement learning with deep energy-based policies. In *ICML*, 2017.

[211] Steven Hansen, Will Dabney, Andre Barreto, Tom Van de Wiele, David Warde-Farley, and Volodymyr Mnih. Fast task inference with variational intrinsic successor features. In *ICLR*, 2020.

[212] Diana Borsa, André Barreto, John Quan, Daniel Mankowitz, Rémi Munos, Hado van Hasselt, David Silver, and Tom Schaul. Universal successor features approximators. In *ICLR*, 2019.

[213] André Barreto, Will Dabney, Rémi Munos, Jonathan J Hunt, Tom Schaul, Hado P van Hasselt, and David Silver. Successor features for transfer in reinforcement learning. In *NIPS*, 2017.

[214] Andre Barreto, Diana Borsa, John Quan, Tom Schaul, David Silver, Matteo Hessel, Daniel Mankowitz, Augustin Zidek, and Remi Munos. Transfer in deep reinforcement learning using successor features and generalised policy improvement. In *ICML*, 2018.

[215] David Warde-Farley, Tom Van de Wiele, Tejas Kulkarni, Catalin Ionescu, Steven Hansen, and Volodymyr Mnih. Unsupervised control through non-parametric discriminative rewards. In *ICLR*, 2019.

[216] Vitchyr H Pong, Murtaza Dalal, Steven Lin, Ashvin Nair, Shikhar Bahl, and Sergey Levine. Skew-fit: State-covering self-supervised reinforcement learning. *arXiv preprint arXiv:1903.03698*, 2019.

[217] Archit Sharma, Shixiang Gu, Sergey Levine, Vikash Kumar, and Karol Hausman. Dynamics-aware unsupervised discovery of skills. *arXiv preprint arXiv:1907.01657*, 2019.

[218] Joan Capdevila, Jesús Cerquides, and Jordi Torres. Mining urban events from the tweet stream through a probabilistic mixture model. *Data mining and knowledge discovery*, 2018.

[219] Elad Hazan, Sham M Kakade, Karan Singh, and Abby Van Soest. Provably efficient maximum entropy exploration. In *ICML*, 2019.

[220] William H Guss, Cayden Codel, Katja Hofmann, Brandon Houghton, Noboru Kuno, Stephanie Milani, Sharada Mohanty, Diego Perez Liebana, Ruslan Salakhutdinov, Nicholay Topin, et al. The minerl competition on sample efficient reinforcement learning using human priors. *arXiv preprint arXiv:1904.10079*, 2019.

[221] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In *NIPS*, 2016.

[222] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Gabriel Dulac-Arnold, et al. Deep q-learning from demonstrations. *arXiv preprint arXiv:1704.03732*, 2017.

[223] Matej Večerík, Todd Hester, Jonathan Scholz, Fumin Wang, Olivier Pietquin, Bilal Piot, Nicolas Heess, Thomas Rothörl, Thomas Lampe, and Martin Riedmiller. Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards. *arXiv preprint arXiv:1707.08817*, 2017.

[224] Corey Lynch, Mohi Khansari, Ted Xiao, Vikash Kumar, Jonathan Tompson, Sergey Levine, and Pierre Sermanet. Learning latent plans from play. *arXiv preprint arXiv:1903.01973*, 2019.

[225] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. In *ICLR*, 2014.

[226] Charles Beattie, Joel Z Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, et al. Deepmind lab. *arXiv preprint arXiv:1612.03801*, 2016.

[227] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning. In *NeurIPS*, 2017.

[228] Alexander Trott, Stephan Zheng, Caiming Xiong, and Richard Socher. Keeping your distance: Solving sparse reward tasks using self-balancing shaped rewards. In *NeurIPS*, 2019.

[229] Lisa Lee, Benjamin Eysenbach, Emilio Parisotto, Eric Xing, Sergey Levine, and Ruslan Salakhutdinov. Efficient exploration via state marginal matching. *arXiv preprint arXiv:1906.05274*, 2019.

[230] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Go-explore: a new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995*, 2019.

[231] Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *ICML*, 2015.

[232] Ronald Parr and Stuart J Russell. Reinforcement learning with hierarchies of machines. In *NIPS*, 1998.

[233] Doina Precup. Temporal abstraction in reinforcement learning. 2001.

[234] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *AAAI*, 2017.

[235] Roy Fox, Sanjay Krishnan, Ion Stoica, and Ken Goldberg. Multi-level discovery of deep options. *arXiv preprint arXiv:1703.08294*, 2017.

[236] Ofir Nachum, Shane Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. *arXiv preprint arXiv:1805.08296*, 2018.

[237] Kevin Frans, Jonathan Ho, Xi Chen, Pieter Abbeel, and John Schulman. Meta learning shared hierarchies. In *ICLR*, 2018.

[238] Ian Osband, Benjamin Van Roy, and Zheng Wen. Generalization and exploration via randomized value functions. In *ICML*, 2016.

[239] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. In *NIPS*, 2016.

[240] Haoran Tang, Rein Houthooft, Davis Foote, Adam Stooke, OpenAI Xi Chen, Yan Duan, John Schulman, Filip DeTurck, and Pieter Abbeel. # exploration: A study of count-based exploration for deep reinforcement learning. In *NIPS*, 2017.

[241] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. In *ICLR*, 2017.

[242] Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. Vime: Variational information maximizing exploration. In *NIPS*, 2016.

[243] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *ICML*, 2017.

[244] Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. *arXiv preprint arXiv:1810.12894*, 2018.

[245] Junhyuk Oh, Yijie Guo, Satinder Singh, and Honglak Lee. Self-imitation learning. In *ICML*, 2018.

[246] Sainbayar Sukhbaatar, Zeming Lin, Ilya Kostrikov, Gabriel Synnaeve, Arthur Szlam, and Rob Fergus. Intrinsic motivation and automatic curricula via asymmetric self-play. *arXiv preprint arXiv:1703.05407*, 2017.

[247] Hao Liu, Alexander Trott, Richard Socher, and Caiming Xiong. Competitive experience replay. In *ICLR*, 2019.

[248] Joel Lehman and Kenneth O Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 2011.

[249] Joel Lehman and Kenneth O Stanley. Novelty search and the problem with objectives. In *Genetic programming theory and practice IX*. Springer, 2011.

[250] Antoine Cully, Jeff Clune, Danesh Tarapore, and Jean-Baptiste Mouret. Robots that can adapt like animals. *Nature*, 2015.

[251] Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

[252] Justin K Pugh, Lisa B Soros, and Kenneth O Stanley. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 2016.

[253] Elliot Meyerson, Joel Lehman, and Risto Miikkulainen. Learning behavior characterizations for novelty search. In *GECCO*, 2016.

[254] Max Jaderberg, Wojciech M Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castaneda, Charles Beattie, Neil C Rabinowitz, Ari S Morcos, Avraham Ruderman, et al. Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science*, 2019.

[255] Carlos Florensa, Jonas Degrave, Nicolas Heess, Jost Tobias Springenberg, and Martin Riedmiller. Self-supervised learning of image embedding for continuous control. *arXiv preprint arXiv:1901.00943*, 2019.

[256] Allan Jabri, Kyle Hsu, Abhishek Gupta, Ben Eysenbach, Sergey Levine, and Chelsea Finn. Unsupervised curricula for visual meta-reinforcement learning. In *NeurIPS*, 2019.

[257] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014.

[258] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

[259] Po-Wei Chou, Daniel Maturana, and Sebastian Scherer. Improving stochastic policy gradients in continuous control with deep reinforcement learning using the beta distribution. In *ICML*, 2017.

[260] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *ICML*, 2018.