Université de Montréal

A deep learning theory for neural networks grounded in physics

par

Benjamin Scellier

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée en vue de l'obtention du grade de
Philosophiæ Doctor (Ph.D.)
en Informatique

December 31, 2020

Université de Montréal

Faculté des arts et des sciences

Cette thèse intitulée

A deep learning theory for neural networks grounded in physics

présentée par

Benjamin Scellier

a été évaluée par un jury composé des personnes suivantes :

Irina Rish
(président-rapporteur)

Yoshua Bengio
(directeur de recherche)

Pierre-Luc Bacon
(membre du jury)

Yann Ollivier
(examinateur externe)

(représentant du doyen de la FESP)

Résumé

Au cours de la dernière décennie, l'apprentissage profond est devenu une composante majeure de l'intelligence artificielle, ayant mené à une série d'avancées capitales dans une variété de domaines. L'un des piliers de l'apprentissage profond est l'optimisation de fonction de coût par l'algorithme du gradient stochastique (SGD). Traditionnellement en apprentissage profond, les réseaux de neurones sont des fonctions mathématiques différentiables, et les gradients requis pour l'algorithme SGD sont calculés par rétropropagation. Cependant, les architectures informatiques sur lesquelles ces réseaux de neurones sont implémentés et entraînés souffrent d'inefficacités en vitesse et en énergie, dues à la séparation de la mémoire et des calculs dans ces architectures. Pour résoudre ces problèmes, le neuromorphique vise à implementer les réseaux de neurones dans des architectures qui fusionnent mémoire et calculs, imitant plus fidèlement le cerveau. Dans cette thèse, nous soutenons que pour construire efficacement des réseaux de neurones dans des architectures neuromorphiques, il est nécessaire de repenser les algorithmes pour les implémenter et les entraîner. Nous présentons un cadre mathématique alternative, compatible lui aussi avec l'algorithme SGD, qui permet de concevoir des réseaux de neurones dans des substrats qui exploitent mieux les lois de la physique. Notre cadre mathématique s'applique à une très large classe de modèles, à savoir les systèmes dont l'état ou la dynamique sont décrits par des équations variationnelles. La procédure pour calculer les gradients de la fonction de coût dans de tels systèmes (qui dans de nombreux cas pratiques ne nécessite que de l'information locale pour chaque paramètre) est appelée "equilibrium propagation" (EqProp). Comme beaucoup de systèmes en physique et en ingénierie peuvent être décrits par des principes variationnels, notre cadre mathématique peut potentiellement s'appliquer à une grande variété de systèmes physiques, dont les applications vont au delà du neuromorphique et touchent divers champs d'ingénierie.

Mots clés : apprentissage profond, apprentissage machine, système physique, equilibrium propagation, modèle à énergie, principe variationnel, principe de moindre action, règle d'apprentissage locale, algorithme du gradient stochastique, réseau de Hopfield, réseau resistif, théorie des circuits électriques, calcul neuromorphique

Abstract

In the last decade, deep learning has become a major component of artificial intelligence, leading to a series of breakthroughs across a wide variety of domains. The workhorse of deep learning is the optimization of loss functions by stochastic gradient descent (SGD). Traditionally in deep learning, neural networks are differentiable mathematical functions, and the loss gradients required for SGD are computed with the backpropagation algorithm. However, the computer architectures on which these neural networks are implemented and trained suffer from speed and energy inefficiency issues, due to the separation of memory and processing in these architectures. To solve these problems, the field of neuromorphic computing aims at implementing neural networks on hardware architectures that merge memory and processing, just like brains do. In this thesis, we argue that building large, fast and efficient neural networks on neuromorphic architectures also requires rethinking the algorithms to implement and train them. We present an alternative mathematical framework, also compatible with SGD, which offers the possibility to design neural networks in substrates that directly exploit the laws of physics. Our framework applies to a very broad class of models, namely those whose state or dynamics are described by variational equations. This includes physical systems whose equilibrium state minimizes an energy function, and physical systems whose trajectory minimizes an action functional (principle of least action). We present a simple procedure to compute the loss gradients in such systems, called equilibrium propagation (EqProp), which requires solely locally available information for each trainable parameter. Since many models in physics and engineering can be described by variational principles, our framework has the potential to be applied to a broad variety of physical systems, whose applications extend to various fields of engineering, beyond neuromorphic computing.

Keywords: deep learning, machine learning, physical system, equilibrium propagation, energy-based model, variational principle, principle of least action, local learning rule, stochastic gradient descent, Hopfield networks, resistive networks, circuit theory, principle of minimum dissipated power, co-content, neuromorphic computing

Contents

List of Tables

List of Figures

Abbreviations List

BPTT	Backpropagation Through Time
CHL	Contrastive Hebbian Learning
CIFAR-10	A dataset of images of animals and objects (a standard benchmark in machine learning)
ConvNet	Convolutional Network
DHN	Deep Hopfield Network
EBM	Energy-Based Model
EqProp	Equilibrium Propagation
GDD	Gradient Descending Dynamics, a property that relates the dynamics of EqProp to the loss gradients, in the discrete-time setting of Chapter 5
GPU	Graphics Processing Unit

LBM	Lagrangian-Based Model
MNIST	A dataset of images of handwritten digits (a standard benchmark in machine learning)
RBP	Recurrent Back-Propagation
SGD	Stochastic Gradient Descent

To the memory of my mother

Acknowledgements

I had the privilege to do my doctoral studies at Mila to conduct research in the blooming field of neural networks, on a topic that I am extremely passionate about. The end of my program marks the end of a chapter, and it is an opportunity for me to reflect on what I have learned and how I have grown thanks to the people who have gone through this journey with me.

First of all, I want to thank my advisor Yoshua Bengio for being a great mentor and for communicating me his passion for neural network research and the 'science of intelligence' in general (both 'artificial' and 'natural'). I am thankful for the freedom he granted me in my work, and for his guidance, his contributions and great insights, as well as his support throughout my PhD program. More than just a scientific advisor, Yoshua is a friend and a truly inspiring person who constantly takes actions for the common interests of society.

I would like to thank all my collaborators for their enthusiasm and contributions. Research is often exciting, but can be frustrating or discouraging sometimes ; nevertheless, collaborating with them was always truly enjoyable. I would like to thank my friend Maxence Ernoult who I had the privilege to meet at a conference, after which we started an extremely fruitful collaboration, together with Axel Laborieux, Julie Grollier and Damien Querlioz. I would also like to thank Jack Kendall for collaborating together with his colleagues Ross Pantone and Kalpana Manickavasagam, and for all the great and inspiring scientific discussions. I also thank my friends and colleagues João Sacramento, Olexa Bilaniuk, Walter Senn, Anirudh Goyal, Jonathan Binas and Thomas Mesnard for our collaborations.

I thank Wulfram Gerstner, Walter Senn and Alexandre Thierry for inviting me to their research groups, as well as the organizers of the Barbados workshop on 'learning in the neocortex' – Blake Richards, Timothy Lillicrap, Konrad Körding and Denis Therien. I also thank the organizers of the Brains, Minds and Machines summer school, and the organizers of the Cellular, Computational and Cognitive Neuroscience summer school. These visits, workshops and summer schools were precious opportunities for me to further expand my knowledge in other scientific disciplines, and I am grateful for the friends and colleagues that I met during these events, for the inspiring conversations that we had and the moments that we shared together. I also thank the juries of my thesis, Yann Ollivier, Pierre-Luc Bacon

and Irina Rish, for their time and their insightful comments on my manuscript, as well as Bertrand Maubert, João Sacramento and Nicolas Zucchet for valuable feedback.

I would also like to thank all my friends in Montreal, Singapore, Switzerland and elsewhere, for all the wonderful moments that we have spent together, which brightened the journey of my doctoral studies. A special mention to the 'friends of Normanton' for all the memories and moments of craziness. I also would like to thank all of those who were present and gave me their invaluable supports during the most difficult moments of my PhD program.

Finally, I thank my family members for their unconditional supports. I am thankful to my parents and brother who always supported me and gave me the freedom to pursue things that I felt like doing. Last but not least, I would like to thank my girlfriend Mannie for her support, her patience, and her love.

I would like to dedicate my thesis to my mother who passed away during the course of my PhD program. She has been a wonderful mother, always giving me support in all possible ways she could. She always encouraged me in pursuing my dreams, and she followed me around the world, even in the most difficult moments of her illness. She was so brave, always smiling and she attracted people's sympathy so naturally by her generosity and care. She inspired me so much and I have learned so much through her. I am extremely grateful that I had a mother like her.

Chapter 1

Introduction

What is intelligence? Are there general principles from which every aspect of intelligence derives? If yes, can we discover these principles, formulate them in mathematical language, and use them to build machines that possess human-like intelligence? And if we managed to do so, would this teach us something about ourselves and the human nature? Here are some of the fascinating questions at the interface of several disciplines of science, engineering and philosophy that motivated me to pursue a PhD in artificial intelligence.

In this introductory chapter, we start by motivating the brain-inspired approach to artificial intelligence (AI). We then review the most important principles of current AI systems. Then, we point out one of their weaknesses: the energy inefficiency of their current implementation in hardware. Finally, we present a novel mathematical framework which allows us to preserve the core principles of current AI systems, while suggesting a path to implement them in substrates that directly exploit physics to do the computations for us. In the long run, this mathematical framework may help us develop AI systems that are much larger, much faster and much more efficient than those that we use today.

1.1. On Artificial Intelligence

Computers are since long able to surpass humans at tasks that we like to think of as intellectually advanced. For example, in 1997, Deep Blue, a computer program created by IBM, beat the world champion Garry Kasparov at the game of chess [Campbell et al., 2002]. While this was an impressive achievement, the form of intelligence that we may want to assign to Deep Blue is very rudimentary, though. The strategy of Deep Blue consists in analyzing essentially every possible scenario to pick the move leading to the best possible outcome. The state of technology at the time made it possible, with enough computing power, to automate this brute-force strategy. Humans on the other hand are not able to analyze every possible scenario at the game of chess in a reasonable amount of time. Our brains haven't evolved to do that. Instead, we develop intuitions about what are the most

promising moves. Developing the right intuitions is what makes the game of chess challenging for us.

Conversely, there are plenty of tasks that humans (and other animals) do so naturally and effortlessly that it can be hard to appreciate the difficulty to build machines to automate them. Consider for example the task of classifying images of cats and dogs. Although we now have computer programs that can classify images fairly reliably, it is only in the past ten years that we have seen impressive improvements in this area. No one is said to be 'intelligent' for being able to tell apart cats from dogs, given that a two-year old can already do this. So why was it so difficult to design programs to classify images? Solving this task is indeed deceptively more complex than it seems. In fact, when we see something, myriads of calculations are performed continuously and automatically in our brains. These calculations happen 'behind the scenes', unconsciously, until the concept of 'cat' or 'dog' pops up to our consciousness. We take for granted the fact that our brains do all these calculations for us, every second of every day. Perhaps one way to appreciate the difficulty that it represents for a programmer to write a program to classify images, is to have in mind that when our eyes see a cat, the computer program sees a bunch of numbers corresponding to pixel intensities (Fig. 1). The difficulty for the programmer thus resides in making sense and handling these numbers in such a way that they produce the answer 'cat'. Similarly, the human brain is able to learn to hear and recognize sounds, to learn to process the sense of touch, etc. We can perceive the world around us, make sense of it and interact with it like no computer or machine can do today. Undoubtedly, humans (and other animals) are in many ways incredibly smarter than machines today.

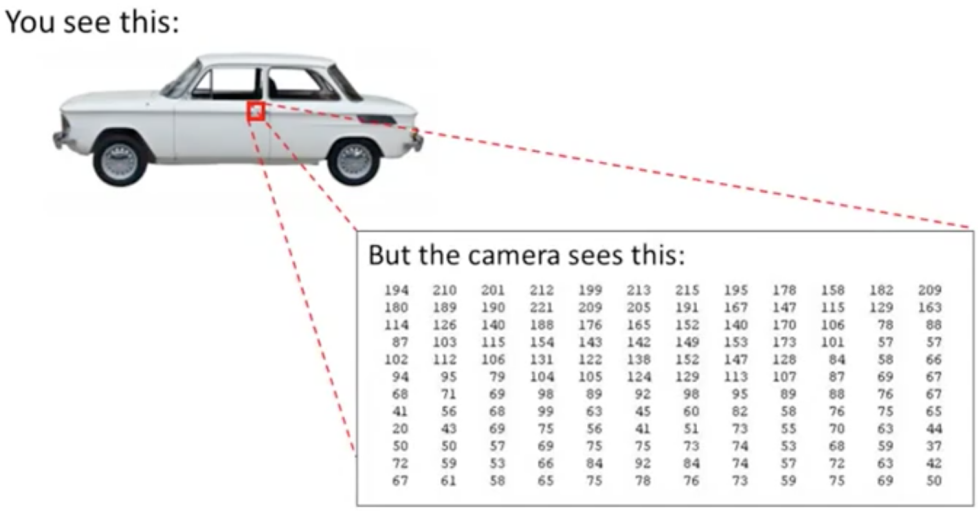

Fig. 1. Picture of a car, and the corresponding representation in computer language. Each number represents the intensity of a pixel. The task of image classification consists in making sense of these numbers to produce the answer 'car'. Figure credits: Ng [2014].

1.1.1. Human Intelligence as a Benchmark

Artificial intelligence (AI) emerged as a research discipline in the 1950s from the idea that every aspect of human intelligence can in principle be discovered, understood, and built into machines. The idea of AI started after Alan Turing formalized the notion of computation and began to study how computers can solve problems on their own [Turing, 1950], but the term *artificial intelligence* was first coined by John McCarthy in 1956. Today, AI is usually used in a broader sense, to refer more broadly to the field of study concerned with designing computational systems to solve practical tasks that we want to automate, whether or not such tasks require some form of human-like intelligence, and whether or not such computational systems are inspired by the brain. However, human intelligence is a natural benchmark for us to build 'intelligent' machines. This is an arbitrary choice, and implies by no means that we humans should be regarded as the 'perfect' or 'ultimate' form of intelligence. But, until we are able to build machines that can do all the incredible things that we humans can do, as easily and as effortlessly as we do them, it seems rather natural to take human intelligence as a benchmark. In the quest of building intelligent machines, Turing proposed that the goal would be reached when our machines can exhibit intelligent behaviours indistinguishable from that of humans.

1.1.2. Machine Learning Basics

Consider the task of classifying images of cats and dogs mentioned earlier. Say that each image is made of 1000 by 1000 pixels, each pixel being described by three numbers (in the RGB color representation). Thus, each image can be represented by a vector of three million numbers. The goal is to come up with a program which, given such a vector x as input, produces 'dog' or 'cat' as output, accordingly. Because there exist many very different vectors x associated to the concept of 'dog', there is no obvious, simple and reliable rule to recognize a dog. To solve this task, the program must combine a very large number of 'weak rules'.

Early forms of AI consisted of explicit, manually-crafted rules, e.g. depending on formal logic. However, using this methodology to figure out all the weak rules that are necessary to correctly classify images is a really daunting task, given the complexity of real-world images. One of the key features of the brain, which these traditional programs did not have, is its ability to learn from experience and to adapt to the environment. Arthur Samuel introduced a new approach to AI, called *machine learning* (ML) [Samuel, 1959], that takes inspiration from how we learn. In the ML approach, instead of operating with predetermined (i.e. immutable) instructions, the program is made of flexible rules that depend on adjustable

parameters. As we modify the parameters, the program changes. The goal is then to tune these parameters so that the resulting program solves the task we want.

To solve the task of image classification, the ML approach requires to collect lots of examples that specify the correct output (the label) for a given input (the image). Such a collection of examples is called a *dataset*. Then we use these examples to adjust the parameters of the ML program so that, for each input image, the program produces as output the label associated to that image. Such a procedure to adjust the parameters is called a *learning algorithm*: the ML program *learns* from examples to solve the problem. Once trained, the program obtained can then be used to predict outputs for new unseen inputs. The performance of the program is assessed on a separate set of examples called the *test dataset*.

In the setting of image classification, the data is such that the labels are provided together with the corresponding images. This setting, where the expected result is known in advance for the available data, is called *supervised learning*. This type of learning is currently the most widely used and successful approach to ML. Depending on the task that we want to solve, and the type and the amount of data that is available, there are two other main machine learning paradigms for training an ML program: unsupervised and reinforcement learning. Unsupervised learning refers to data for which no explicitly identified labels exist. Reinforcement learning refers to the case where no exact labels exist, but a scalar value is available (usually called 'reward') that provides some knowledge on whether a proposed output is good or bad.

1.2. Neural Networks

Artificial neural networks (ANNs) are a family of ML models inspired by the basic working mechanisms of the brain. In recent years ANNs have had resounding success in AI, in areas as diverse as image recognition, speech recognition, image generation, speech synthesis, text generation and machine language translation. We start this section by presenting the basic concepts of neuroscience that have inspired the design of ANNs. Then we present the key principles at the heart of these ANNs. Finally we point out some weaknesses in the current implementation of these principles in hardware, which makes these neural networks orders of magnitude less energy efficient than brains.

Subsection 1.2.1 is inspired by Dehaene [2020, Chapter 5].

1.2.1. Neuroscience Basics

The foundations of modern neuroscience were laid by Santiago Ramon y Cajal, several decades before AI research started. Cajal was the first to observe the brain's micro-organisation with a microscope. He observed that the brain consists of disjoint nerve cells

(the *neurons*), not of a continuous network as the proponents of the *reticular theory* thought before him. Neurons have a very particular shape. Each neuron is composed of three main parts (Figure 2): a large 'tree' composed of thousands of branches (the *dendrites*[1]), a cell body (also called the *soma*), and a long fiber which extends out of the cell body towards other neurons (the *axon*). A neuron collects information from other neurons through its dendritic tree. The messages collected in the dendrites converge to the cell body, where they are compiled. After compilation, the neuron sends a unique message, called *action potential* (or *spike*), which is carried along its *axon* away from the cell body. In turn, this message is delivered to other neurons.

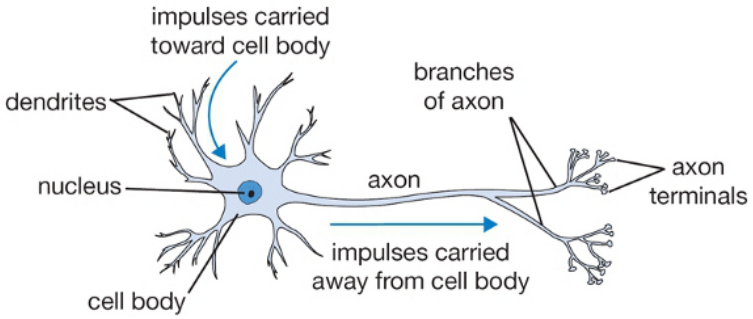

Fig. 2. Schema of a neuron[2].

While neurons are distinct cells, they come into contact at certain points called *synapses* (Figure 3). Synapses are junction zones through which neurons communicate. Specifically, each synapse is the point of contact of the axon of a neuron (called *pre-synaptic* neuron) and the dendrite of another neuron (called *post-synaptic* neuron). The message traveling through the axon of the pre-synaptic neuron is electrical, but the synapse turns it into a chemical message. The axon terminal of the pre-synaptic neuron contains some sorts of pockets (the *vesicles*) filled with molecules (the *neurotransmitters*). When the electrical signal reaches the axon terminal, the vesicles open and the neurotransmitters flow in the small synaptic gap between the two neurons. The neurotransmitters then bind with the membrane of the post-synaptic neuron at specific points (the *receptors*). A neurotransmitter acts on a receptor as a key in a lock: they open 'gates' (called *channels*) in the post-synaptic membrane. As a result, ions flow from the extra-cellular fluid through these channels and generate a current in the post-synaptic neuron. To sum up, the message coming from the pre-synaptic neuron went from electrical to chemical, back to electrical, and in the process, the message was transmitted to the post-synaptic neuron.

Each synapse is a chemical factory in which numerous elements can be modified: the number of vesicles and their size, the number of receptors and their efficacy, as well as the

[1]In Greek, the word *dendron* means tree
[2]https://towardsdatascience.com/a-gentle-introduction-to-neural-networks-series-part-1-2b90b87795bc

size and the shape of the synapse itself. All these elements affect the strength with which the pre-synaptic electrical message is transmitted to the post-synaptic neuron. Synapses are constantly modified and these modifications reflect what we *learn*.

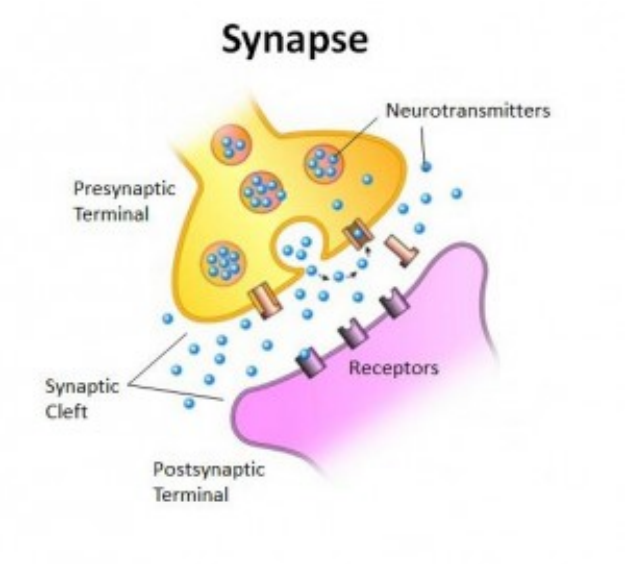

Fig. 3. Schema of a synapse[3].

The human brain is composed of around 100 billion (10^{11}) neurons, interconnected by a total of around a quadrillion (10^{15}) synapses. The brain is a huge parallel computer: in this incredibly complex machine, all the synapses work in parallel – like independent nanoprocessors – to process the messages sent between the neurons. Besides, synapses are modified in response to experience, and in turn these modifications alter our behaviours. Thus, synapses are both the computing units and the memory units of the brain. For every task we do, all our thoughts, memories and all our behaviours emerge from the neural activity generated by this machinery.

One of the fundamental questions of neuroscience is that of figuring out the *learning algorithms* of the brain: what is the set of rules which translate the experiences we have into synaptic changes, and how do these synaptic changes modify our behaviour? Understanding the brain's learning algorithms is not only key to understanding the biological basis of intelligence, but would also unlock the development of truly intelligent machines.

1.2.2. Artificial Neural Networks

Artificial neural networks are ML models that draw inspiration from real brains. Artificial neurons imitate the functionality of biological neurons. These models are highly simplified: they keep some essential ideas from real neurons and synapses but they discard many details

[3]https://thesalience.wordpress.com/neuroscience/the-chemical-synapse/chemical-synapses/

of their working mechanisms. The first neuron model was introduced by McCulloch and Pitts [1943], but the idea to use such artificial neurons in machine learning was proposed by Rosenblatt [1958] and Widrow and Hoff [1960]. Artificial neurons used today in deep learning are essentially unchanged and rely on the same basic math algebra.

Each neuron i is described by a single number y_i. This number can be thought of as the *firing rate* of neuron i, that is the rate of spikes sent along its axon. Each synapse is also described by a single number, representing its strength. The strength of the synapse connecting pre-synaptic neuron j to post-synaptic neuron i is denoted W_{ij}. These artificial synapses can transmit signals with different efficacies depending on their strength. The neuron calculates a nonlinear function of the weighted sum of its inputs:

$$y_i = \sigma\left(\sum_j W_{ij} y_j\right). \tag{1.1}$$

The *pre-activation* $x_i = \sum_j W_{ij} y_j$ is a weighted sum of the messages received from other neurons, weighted by the corresponding synaptic strengths. x_i can be though of as the membrane voltage of neuron i. σ is a function called an *activation function*, which maps x_i onto the firing-rate y_i.

Such artificial neurons can be combined to form an artificial neural network (ANN). Each neuron in the network receives messages from other neurons (the y_j's), compiles them (x_i), and sends in turn a message to other neurons (y_i). Thus, a network of interconnected neurons exploits the composition of many elementary operations to form more complex computations. The synaptic strengths (the W_{ij}'s), also called *weights*, play the role of adjustable parameters that parameterize this computation.

Deep learning refers to ANNs composed of multiple layers of neurons [LeCun et al., 2015, Goodfellow et al., 2016]. These *deep neural networks* were inspired by the structure of the visual cortex in the brain, each layer corresponding to a different brain region. One of the core ideas of neural networks is that of *distributed representations*, the idea that the vector of neuron's states can represent abstract concepts, by opposition to other approaches to AI that use discrete symbols to represent concepts. In a deep network, each layer of neurons applies specialized operations and transformations on its inputs, with the intuition that each layer builds up more abstract concepts than the previous [Bengio, 2009].

1.2.3. Energy-Based Models vs Differentiable Neural Networks

Several families of neural networks emerged in the 1980s. One of these families is that of *energy-based models*, which includes the Hopfield network [Hopfield, 1982] and the Boltzmann machine [Ackley et al., 1985]. In these models, under the assumption that the synaptic weights are symmetric (i.e. $W_{ij} = W_{ji}$ for every pair of neurons i and j), the dynamics

of the network converges to an equilibrium state, after iterating Eq. 1.1 a large number of times for every neuron i. Because of the large number of iterations required, these models tend to be slow. This is one of the reasons why these neural networks have been mostly discontinued today. However, by reinterpreting the equilibrium equation of energy-based models as a variational principle of physics, I believe that these models could be the basis of a new generation of fast, efficient and scalable neural networks grounded in physics. We will come back to this point later in the discussion (Section 1.3).

The family of neural networks that is at the heart of the on-going deep learning revolution is that of *differentiable neural networks*, which became popular thanks to the discovery of the *backpropagation algorithm* to train them [Rumelhart et al., 1988]. In such neural networks, each operation in the process of computation is differentiable. The earliest models of this kind were feedforward neural networks (e.g. the multi-layer perceptron), wherein the connections between the neurons do not form loops. Recurrent neural networks can also be cast to this category of differentiable neural networks, by unfolding the graph of their computations in time. Since their inception, differentiable neural networks have come a long way. Many novel architectures have been introduced, in particular: convolutional neural networks [Fukushima, 1980, LeCun et al., 1989], Long Short-term Memory [Hochreiter and Schmidhuber, 1997, Graves, 2013], and attention mechanisms [Bahdanau et al., 2014, Vaswani et al., 2017].

1.2.4. Stochastic Gradient Descent

The computations performed by a neural network are parameterized by its synaptic weights. The goal is to find weight values for which the computations of the neural network solve the task of interest. One essential idea of machine learning is to introduce a *loss function*, which provides a numerical measure of how good or bad the computations of the model are, with respect to the task that we want to solve. The goal is then to minimize the loss function with respect to the model weights. For example, in image classification, the computations of the model produce an output which represents a 'guess' for the class of that image, and the loss provides a graded measure of 'wrongness' between that guess and the actual image label. A smaller value of the loss function means that the model produces an output closer to the desired target. The loss function is minimal when the output is equal to the desired target.

One of the most important ideas of deep learning today is *stochastic gradient descent* (SGD). Provided that the loss function is differentiable with respect to the network weights, we can use the gradient of the loss function to indicate the direction of the minimum of this function. SGD consists in taking examples from the training set one at a time, and adjusting the network weights iteratively in proportion to the negative of the gradient of the

loss function. At each iteration, the network performance (as measured per the loss value) slightly improves.

A key discovery that has greatly eased and accelerated deep learning research is the following. Given a computer program that computes a *differentiable* scalar function $f : \mathbb{R}^n \to \mathbb{R}$, it is possible to automatically transform the program into another program that computes the gradient operator $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$. The gradient $\nabla f(\theta)$ can then be evaluated at any given point $\theta \in \mathbb{R}^n$, with a computational overhead that scales linearly with the complexity of the original program. This technique, known as *reverse-mode automatic differentiation* [Speelpenning, 1980], provides a general framework for 'backpropagating loss gradients' [Rumelhart et al., 1988] in any *differentiable computational graph*. In the last decade, dozens of deep learning frameworks and libraries have been developed, which exploit reverse-mode automatic differentiation to compute gradients in arbitrary differentiable computational graphs. This includes Theano [Bergstra et al., 2010] – a framework that was developed at Université de Montréal – and the more recent Tensorflow [Abadi et al., 2016] and PyTorch [Paszke et al., 2017] frameworks. The emergence of these deep learning frameworks has considerably accelerated deep learning research, by enabling researchers to quickly design neural network architectures and train them by SGD. Thanks to these frameworks, deep learning researchers can explore the space of differentiable computational graphs much more rapidly, as they seek novel and more effective neural architectures.

1.2.5. Landscape of Loss Functions

It was not trivial to discover that deep neural networks can be trained at all. Until the seminal work of Hinton et al. [2006], common belief was that neural networks with more than two layers were essentially impossible to train. In particular, because the landscape of the loss function associated to a deep network is typically highly non-convex, a common misconception was that gradient-descent methods would likely get stuck at bad local minima. In terms of generalization performance, the large over-parameterization of neural networks was also against general prescriptions from classical statistics and learning theory. One of the surprising discoveries of the deep learning revolution was that, in such highly non-convex and over-parameterized statistical models, provided that the loss landscape has appropriate shape, SGD can solve complex tasks by finding excellent parameter values that generalize well to unseen examples.

Several elements contributed to unlock the training of deep neural networks ; among others: the discovery [Glorot et al., 2011] that the ReLU ('Rectified Linear Unbounded') activation function usually outperforms the sigmoid activation function, the discovery of better weight initialization schemes [Glorot and Bengio, 2010, Saxe et al., 2013, He et al., 2015], and the batch-normalization technique to systematically normalize signal amplitudes

at each layer of a network [Ioffe and Szegedy, 2015]. Besides, fundamental advances have come from the introduction of new network architectures such as the ones mentioned earlier, and from novel machine learning paradigms such as that of generative adversarial networks [Goodfellow et al., 2014]. All these techniques have as an effect to modify the landscape of the loss function, as well as the starting parameter (before training) in the parameter space. Understanding how the landscape of the loss function can be appropriately shaped to ease optimization by SGD is an active area of research [Poggio et al., 2017, Arora, 2018].

1.2.6. Deep Learning Revolution

In recent years, deep neural networks have proved capable of solving very complex problems across a wide variety of domains. Today, they achieve state-of-the-art performance in image recognition [He et al., 2016], speech recognition [Hinton et al., 2012, Chan et al., 2016], machine translation [Vaswani et al., 2017], image-to-text [Sharma et al., 2018], text-to-speech [Oord et al., 2016], text generation [Brown et al., 2020], and synthesis of realistic-looking portraits [Karras et al., 2019], among many other applications. Neural networks have become better than humans at playing Atari games [Mnih et al., 2013], playing the game of Go [Silver et al., 2018] and playing Starcraft [Vinyals et al., 2019]. Perhaps what is most exciting is that, although these neural networks are designed to solve very different tasks and deal with different types of data, they are all trained using the same handful of basic principles. As we scale these neural networks, more advanced aspects of intelligence seem to emerge from this handful of principles.

While the pace of progress in neural network research is breathtaking, we should also emphasize that, without any question, neural networks are nowhere close to 'surpass' humans. In their current form, they miss key elements of human intelligence. Whenever neural networks beat humans, they beat us at a very specific task and/or under very specific conditions. We may need new learning paradigms to move away from 'task-specific' neural networks towards 'multi-functional' and continually learning neural networks. Besides, as of today, neural networks cannot handle and combine abstract concepts nearly as flexibly as humans do. Making progress along these lines may require new breakthroughs, to give these neural networks the ability to develop a 'thought language' and a sense of causal reasoning, among others.

1.2.7. Graphics Processing Units

The on-going deep learning revolution owes to other technological developments too. In the past decades, the amount of data available has greatly increased, and powerful multi-core parallel processors for general purpose computing, such as Graphics Processing Units

(GPUs), have emerged [Owens et al., 2007]. Training large models on large datasets – here is part of the recipe to make a deep neural network solve a challenging task. In the 1980s, when deep neural networks were first conceived, the lack of data and computational power to train them made it practically unfeasible to demonstrate their effectiveness.

The last decade of neural network research seems to hint at a simple and straightforward strategy to further improve performance of our AI systems: scaling. Using more memory and more compute to train larger models on larger datasets – here is the current trend to build state-of-the-art deep learning systems. The largest model ever built thus far, GPT-3 [Brown et al., 2020], has a capacity of 175 billion parameters.

Training these neural networks requires very large amounts of computations. Standard practice today is to distribute the computations across more and more GPUs, to train larger and larger models. Still, even using thousands of GPUs working in parallel, training these neural networks can take months. For example, AlphaZero learnt to play the game of Go by playing 140 million games, which took 5000 processors and two weeks. Moreover, training such models can cost millions of dollars, just for electricity consumption, not to mention their ecological impact. Yet, even these large neural networks are only a tiny fraction of the size of the human brain. What causes such inefficiency, preventing us from building models of the size of the human brain?

1.2.8. The Von Neumann Bottleneck

If at the conceptual level the neural networks used today take their overall strategy from the brain, on the hardware implementation level however, they use little of the cleverness of nature. Our current processors, on which these neural networks are trained and run, operate in fundamentally different ways than brains. They rely on the *von Neumann architecture*. In this computer architecture, the memory unit where information is stored, is separated from the processing unit where calculations are done. A so-called *bus* moves information back and forth between these two units. Over the course of history of computing, this computer architecture has become the norm, and today, the von Neumann architecture is used in virtually all computer systems: laptops, smartphones, and all kinds of embedded systems. The GPUs, massively used for neural network training today, also rely on the von Neuman architecture, where memory is separated from computing.

The brain on the other hand deeply merges memory and computing by using the same functional unit: the synapse. The human brain is composed of a quadrillion (10^{15}) synapses. In other words, the human brain has 10^{15} nanoprocessors working in parallel.

Training neural networks on von Neumann hardware as we do it today is extraordinarily energy inefficient in comparison with the way brains operate. The necessity to move the data back and forth between the memory and processing units in the von Neumann architecture

is energy intensive and creates considerable latency. This limitation is known as the *von Neumann bottleneck*. How inefficient is it compared to biological systems like the brain? The human brain is composed of 10^{11} neurons and consumes around 20W to conduct all of its activities [Attwell and Laughlin, 2001]. In comparison, training a BERT model (a state-of-the-art natural language processing model) on a modern supercomputer requires 1500 kW.h [Strubell et al., 2019], which is the total amount of energy consumed by a brain in nine years. Besides, a GPU running real-time object detection with YOLO [Redmon et al., 2016], a network smaller than the brain by four orders of magnitude, consumes around 200W.

This striking mismatch holds more broadly with biological systems in general. For example, Kempes et al. [2017] study the energy efficiency of 'cellular computation' in the process of biological translation. What is the amount of energy required (in ATP equivalents) by ribosomes to assemble amino acids into proteins ? They point out that "the best supercomputers perform a bit operation at roughly $5.27 \times 10^{-13} J$, [...] which is about five orders of magnitude less efficient than biological translation."

To sum up, our current neural networks are orders of magnitude less energy efficient than biological systems at processing information, and the von Neumann bottleneck is largely responsible for this inefficiency. If using more and more GPUs may increase speed, and thereby speed up training and inference of neural networks, this strategy however can't improve energy efficiency.

1.2.9. In-Memory Computing

In order to build massively parallel neural networks that are energy efficient and can scale to the size of the human brain, we need to fundamentally rethink the underlying computing hardware. We need to design neural networks so that computations are performed at the physical location of the synapses, where the strength of the connections (the weights of the neural network) are stored and adjusted, just like in the brain. The concept of hardware that merges memory and computing is called *in-memory computing* (or *in-memory processing*), and the field tackling this problem is called *neuromorphic computing*. This field of research, started by Carver Mead in the 1980s [Mead, 1989] aims at mimicking brains at a hardware level, by building physical neurons and synapses onto a chip.

The most common approach to in-memory computing today is to use *programmable resistors* as synapses. Programmable resistors, such as *memristors* [Chua, 1971], are resistors whose conductance can be changed (or 'programmed'). The weights of a neural network can be encoded in the conductance of such devices. In the last decade, important advances in nanotechnology were made, and a number of new technologies have emerged and have been studied as potentially promising programmable resistors [Burr et al., 2017, Xia and Yang, 2019].

Neuromorphic computing thus explores analog computations that fundamentally depart from the standard digital computing paradigm.

1.2.10. Challenges of Analog Computing

Analog processing differs from digital processing in important ways. Whereas digital circuits manipulate binary signals with reliably distinguishable *on* and *off*-states, analog circuits on the other hand manipulate real-valued currents and voltages that are subject to analog noise that bounds the precision with which computation may be performed. More importantly, analog devices suffer from *mismatches*, i.e. small random variations in the physical characteristics of devices, which occurs during their manufacturing. No two devices are exactly alike in their characteristics, and it is impossible to make a perfect clone of one. These variations result in behavioral differences between identically designed devices. Due to the accumulation of the mismatch errors from individual devices, it is very difficult to analytically predict the behavior of a large analog circuit.

A growing field of research in the neuromorphic literature attempts to perform in analog the operations that we normally do in software, so as to implement feedforward neural networks and the backpropagation algorithm efficiently. In this approach, the starting point is an equation of the kind of Eq. 1.1. Many of these operations are then performed in analog and combined to form the computations of a feedforward network. However, because of device mismatches, it is hard to perform such idealized operations, and as we combine many of these operations, the resulting computation may be different from the desired one. Either these idealized operations are performed with low precision, or we may spend a lot of energy trying to improve precision, e.g. by using analog-to-digital conversion.

Not coincidentally, the constraint of device nonidealities and device variability is shared with biology too. No two neurons are exactly the same. This realization demonstrates, in principle, that it is possible to train (biological) neural networks even in the presence of noise and imperfect 'devices'. It invites us to rethink the learning algorithm for neural networks (and the notion of computation altogether).

1.3. A Deep Learning Theory for Neural Networks Grounded in Physics

In this thesis, we propose an alternative theoretical framework for neural network inference and training, with potential implications for neuromorphic computing. Our theoretical framework preserves the key principles that power deep learning today, such as optimization by stochastic gradient descent (SGD), but we use variational formulations of the laws of physics as first principles, so as to directly implement neural networks in physics. We

present two very broad classes of neural network models, called *energy-based models* and *Lagrangian-based models*, whose state or dynamics derive from variational principles. The learning algorithm, called *equilibrium propagation* (EqProp), enables to estimate the gradients of arbitrary loss functions in such physical systems using solely locally available information for each parameter.

1.3.1. Training Physical Systems with Adjustable Parameters by Gradient Descent

Consider a physical system composed of multiple parts whose characteristics and working mechanisms may be only partially known. The system has some 'adjustable parameters', some of them playing the role of 'inputs', and we may read or measure an output or 'response' on some other 'output' part of the system. We can think of this black box system as performing computations and implementing a nonlinear input-to-output mapping function (which may be analytically unknown). We desire to tune the adjustable parameters of the system by gradients descent (as we normally do in deep learning) so that the overall computation improves on the task we want to solve. Now the question is: how can we compute or estimate the gradients in such a physical system, by relying on the physics of the system?

The main theoretical result of the thesis is that, for a large class of physical systems (those whose state or dynamics derive from a variational principle), there is a simple procedure to estimate the parameter gradients, which in many practical situations requires only locally available information for each parameter.

1.3.2. Variational Principles of Physics as First Principles

Rather than Eq. 1.1, our starting point is a *variational equation* of the form

$$\frac{\partial E}{\partial s} = 0, \tag{1.2}$$

where E is a scalar function. If s is the state of the system, then Eq. 1.2 is an equilibrium condition. In this case, we say that the system is an *energy-based model* (EBM) and we call E the *energy function*.

In this thesis, we also introduce the concept of *Lagrangian-based model* (LBM). Variational equations exist not just to characterize equilibrium states, but also entire trajectories. Many physical systems are such that their trajectory derives from a *principle of stationary action* (e.g. a principle of least action). Denoting $\mathrm{s_t}$ the state of the system at time t, this means that the (continuous-time) trajectory $\mathrm{s} = \{\mathrm{s_t}\}_{0 \leq t \leq T}$ over a time interval $[0, T]$

minimizes a functional of the form

$$\mathcal{S} = \int_0^T L(\mathrm{s_t}, \dot{\mathrm{s}}_\mathrm{t}) dt, \tag{1.3}$$

where $\dot{\mathrm{s}}_\mathrm{t}$ is the time derivative of $\mathrm{s_t}$, L is a function called the *Lagrangian function* of the system, and $\mathcal{S}$ is a scalar functional called the *action*. The stationarity of the action tells us that $\frac{\delta \mathcal{S}}{\delta \mathrm{s}} = 0$, which is another variational equation of the kind of Eq. 1.2. These systems, which we call Lagrangian-based models (LBMs), are suitable in particular in the setting with time-varying inputs and can thus play the role of 'recurrent neural networks'.

The learning algorithm presented in this thesis to train EBMs and LBMs is called *equilibrium propagation* (EqProp). EqProp allows to compute gradients with respect to arbitrary loss functions in these EBMs and LBMs. Furthermore, if the energy function (resp. Lagrangian function) of the system has a property called *sum-separability*, meaning that it is the sum of the energies (resp. Lagrangians) of its parts, then computing the loss gradients with EqProp requires only information that is locally available for each parameter (i.e. the learning rule is *local*). Thus, EqProp preserves the key benefit of being compatible with stochastic gradient descent (SGD), while offering the possibility to directly exploit physics to implement and train neural networks.

1.3.3. Universality of Variational Principles in Physics

In the 1650s, Pierre de Fermat proposed the *principle of least time* which states that, between two given points, the light travels the path which takes the least time. He showed that both the laws of reflection and refraction can be derived from this principle. Fermat's least time principle is an instance of what we call more generally a *variational principle*. Today, the variational approach pervades much of modern physics and engineering [Lanczos, 1949], with applications not only in optics, but also in mechanics, electromagnetism, thermodynamics, etc. Even at a fundamental level of description, our universe seems to behave according to variational principles: for example, Einstein's equations of general relativity can be derived from the Einstein-Hilbert action, and in a sense, Feynman's path integral formulation of quantum mechanics can be seen as a generalized principle of least action [Feynman, 1942].

Thus, many physical systems qualify as energy-based models or Lagrangian-based models. This offers in principle a lot of options for implementing our proposed method on physical substrates. In this manuscript we will present one such option in details: *nonlinear resistive networks*.

1.3.4. Rethinking the Notion of Computation

Interestingly, our theoretical framework invites us to rethink not only the von Neumann architecture on which our current processors rely, but also the notion of computation altogether.

Much of computer science deals with computation in the abstract, without worrying about physical implementation [Lee, 2017]. Our computers today rely on the computing paradigm introduced by Turing, where computers operate on digital data and carry out computations algorithmically, via step-by-step (discrete-time) processes. The von Neumann architecture was invented to implement these computations (i.e. to bridge the gap between physics and these abstract computations), but suffers from the speed and energy efficiency problems mentioned earlier (Section 1.2.8).

The theoretical framework presented in this manuscript can be seen as an alternative approach to computing that takes advantage of the ways by which Nature operates. We suggest a novel computing paradigm, which uses the variational formulations of the laws of physics as first principles. Together with the *equilibrium propagation* (EqProp) training procedure, our approach suggests a way to implement the core principles of deep learning by exploiting physics directly. As will become apparent in Section 4.3.2, the process of 'computations' in EqProp is very different from the step-by-step processes of Turing's conventional computing paradigm. Although we may call EqProp a learning 'algorithm', it is not an algorithm in the conventional sense (one that performs step-by-step computations).

1.4. Overview of the Manuscript and Link to Prior Works

The manuscript is organized as follows.

- In Chapter 2, we present EqProp in its original formulation, as a learning algorithm to train energy-based models (EBMs). We show that, provided that the energy function of the system is *sum-separable*, then the learning rule of EqProp is local. This corresponds to Section 3 and Appendix A of Scellier and Bengio [2017].
- In Chapter 3, we use EqProp to train a particular class of EBMs called *gradient systems*. This includes the continuous Hopfield network, a neural network model introduced by Hopfield in the 1980s, studied by both the neurosience community and the neuromorphic community. In this setting, the learning rule of EqProp is a form of contrastive Hebbian learning. The first part of this chapter corresponds to the result established in Scellier and Bengio [2019]. The second part corresponds to sections 2, 4, and 5 of Scellier and Bengio [2017].

- In Chapter 4, we show that a class of analog neural networks called *nonlinear resistive networks* are energy-based models: they possess an energy function called the *co-content* of the circuit, as a reformulation of Kirchhoff's laws. Furthermore the co-content has the sum-separability property. Therefore we can train these nonlinear resistive networks with EqProp using a local learning rule. This chapter corresponds to Kendall et al. [2020].
- In Chapter 5, we present a class of discrete-time neural network models trainable with EqProp, which is useful to accelerate computer simulations. This formulation, which uses notations closer to those used in conventional deep learning, is also more adapted to train more advanced network architectures such as convolutional networks. This chapter corresponds to Ernoult et al. [2019] and Laborieux et al. [2021].
- In Chapter 6, we present on-going developments. In particular, we introduce the concept of *Lagrangian-based models*, a wide class of machine learning models that can serve as recurrent neural networks and can be implemented directly in physics by exploiting the *principle of stationary action*. These Lagrangian-based models can also be trained with an EqProp-like training procedure. We also present an extension of EqProp to stochastic systems, which was introduced in Appendix C of Scellier and Bengio [2017]. Finally, we briefly present the *contrastive meta-learning* framework of Zucchet et al. [2021], which uses the EqProp technique to train the meta-parameters of a meta-learning model.

1.5. Contributions.

This manuscript is based on the following papers: Scellier and Bengio [2017, 2019], Scellier et al. [2018], Ernoult et al. [2019, 2020], Kendall et al. [2020], Laborieux et al. [2021]. My contributions in these papers include:

- the general formulation of EqProp, and the theorems/proofs presented in Scellier and Bengio [2017], as well as the simulations,
- the theoretical result presented in Scellier and Bengio [2019],
- the mathematical formulation of the 'vector field version' of EqProp [Scellier et al., 2018],
- the main theoretical result presented in Ernoult et al. [2019],
- the theoretical results presented in Ernoult et al. [2020],
- the mathematical formulation and the proof of the theorem presented in Kendall et al. [2020],
- the idea of using symmetric difference estimators for the loss gradients in Laborieux et al. [2021].

Chapter 2

Equilibrium Propagation: A Learning Algorithm for Systems Described by Variational Equations

Much of machine learning today is powered by stochastic gradient descent (SGD). The standard method to compute the loss gradients required at each iteration of SGD is the backpropagation (Backprop) algorithm. Equilibrium propagation (EqProp) is an alternative to Backprop to compute the loss gradients. The difference between EqProp and Backprop lies in the class of models that they apply to: while Backprop applies to *differentiable neural networks*, EqProp is broadly applicable to systems described by *variational equations*, i.e. systems whose state or dynamics is a stationary point of a scalar function or functional. Since many physical systems have this property [Lanczos, 1949], EqProp offers the perspective to implement and train machine learning models which use the laws of physics at their core.

In this chapter, we present EqProp in its original formulation [Scellier and Bengio, 2017], as an algorithm to train energy-based models (EBMs). EBMs are systems whose equilibrium states are stationary points of a scalar function called the *energy function*. EBMs are suitable in particular when the input data is static. In most of the manuscript, we consider for simplicity of presentation the supervised learning setting with static input, e.g. the setting of image classification where the input is an image and the target is the category of that image. However, EqProp is applicable beyond this setting. In Section 6.1, we introduce the concept of *Lagrangian-based models* (LBMs) which, by definition, are physical systems whose dynamics derives from a *principle of stationary action*, and we show how EqProp can be applied to such systems. LBMs are suitable in the context of time-varying data, and can thus play the role of 'recurrent neural networks'. We also present an extension of EqProp to stochastic systems (Section 6.2), and to the setting of meta-learning (Section 6.3).

The present chapter is organized as follows.

- In section 2.1, we present the stochastic gradient descent (SGD) algorithm, which is at the heart of current deep learning. We present SGD in the setting of supervised learning, which we will consider in most of the manuscript to illustrate the ideas of the equilibrium propagation training framework. We note however that SGD is also the workhorse of state-of-the-art unsupervised and reinforcement learning algorithms.
- In section 2.2 we define the notion of *energy-based model* (EBM) that we will use throughout the manuscript.
- In section 2.3, we present the general formula for computing the loss gradients in an EBM, and in section 2.4, we present the equilibrium propagation (EqProp) algorithm to estimate the loss gradients. Under the assumption that the energy function of the system satisfies a property called *sum-separability*, the learning rule for each parameter is local.
- In section 2.5, we give a few examples of models trainable with EqProp, which we will study in the next chapters. Besides the well-known Hopfield model, nonlinear resistive networks, flow networks and elastic networks are examples of sum-separable energy-based models and, as such, are trainable with EqProp using a local learning rule.
- In section 2.7, we discuss the general applicability of the framework presented here and the conditions under which EqProp is applicable.

2.1. Stochastic Gradient Descent

In most of the manuscript, we consider the supervised learning setting, e.g. the setting of image classification where the data consists of images together with the labels associated to these images. In this scenario, we want to build a system that is able, given an input x, to 'predict' the label y associated to x. To do this, we design a *parametric* system, meaning a system that depends on a set of *adjustable parameters* denoted θ. Given an input x, the system produces an output $f(\theta, x)$ which represents a 'guess' for the label of x. Thus, the system implements a mapping function $f(\theta, \cdot)$ from an input space (the space of x) to an output space (the space of y), parameterized by θ. The goal is to tune θ so that for most x of interest, the output $f(\theta, x)$ is close to the target y. The 'closeness' between $f(\theta, x)$ and y is measured using a scalar function $C(f(\theta, x), y)$ called the *cost function*. The overall performance of the system is measured by the expected cost $\mathcal{R}(\theta) = \mathbb{E}_{(x,y)}[C(f(\theta, x), y)]$ over examples (x, y) from the data distribution of interest. The goal is then to minimize $\mathcal{R}(\theta)$ with respect to θ.

In deep learning, the core idea and leading approach to tune the set of parameters θ is *stochastic gradient descent* (SGD). The first step consists in gathering a (large) dataset of examples $\mathcal{D}_{\text{train}} = \{(x^{(i)}, y^{(i)})\}_{1 \leq i \leq N}$, called *training set*, which specifies for each input $x^{(i)}$

the correct output $y^{(i)}$. Then, each step of the training process proceeds as follows. First, a sample (x, y) is drawn from the training set. Input x is presented to the system, which produces $f(\theta, x)$ as output. This output is compared with y to evaluate the loss

$$\mathcal{L}(\theta, x, y) = C(f(\theta, x), y). \tag{2.1}$$

Subsequently, the gradient $\frac{\partial \mathcal{L}}{\partial \theta}(\theta, x, y)$ is computed or estimated using some procedure, and the parameters are updated proportionally to the loss gradient:

$$\Delta\theta = -\eta \frac{\partial \mathcal{L}}{\partial \theta}(\theta, x, y), \tag{2.2}$$

where η is a step-size parameter called *learning rate*. This process is repeated multiple times (often millions of times) until convergence (or until desired). Once trained, the performance of the system is evaluated on a separate set of *previously unseen* examples, called *test set* and denoted $\mathcal{D}_{\text{test}} = \{(x^{(i)}_{\text{test}}, y^{(i)}_{\text{test}})\}_{1 \leq i \leq M}$. The *test loss* is $\widehat{\mathcal{R}}_{\text{test}}(\theta) = \frac{1}{M} \sum_{i=1}^{M} \mathcal{L}\left(\theta, x^{(i)}_{\text{test}}, y^{(i)}_{\text{test}}\right)$.

Several variants of SGD have been proposed, which use adaptive learning rates to accelerate the optimization process. This includes the *momentum method* [Sutskever et al., 2013] and *Adam* [Kingma and Ba, 2014]. In some cases, these methods are not only faster, but also achieve better test performance than standard SGD. Besides, common practice is to average the loss gradients over *mini-batches* of data examples before updating the weights – a method sometimes called *mini-batch gradient descent* – but in this manuscript we consider for simplicity of presentation that training examples are processed one at a time.

The SGD algorithm described above powers nearly all of deep learning today. There are, however, two ingredients that we have not specified so far: the 'system' that implements the mapping function $f(\theta, x)$, and the 'procedure' to compute the loss gradient $\frac{\partial \mathcal{L}}{\partial \theta}(\theta, x, y)$. In conventional deep learning, the 'system' is a *differentiable neural network*, and the loss gradients are computed with the *backpropagation algorithm*. In this chapter, we present an alternative framework for optimization by SGD, where the 'system' (i.e. the neural network) is an *energy-based model*, and the procedure to compute the loss gradients is called *equilibrium propagation* (EqProp).

2.2. Energy-Based Models

There exist different definitions for the concept of energy-based model in the literature – see LeCun et al. [2006] for a tutorial. In this manuscript, we reserve the term to refer to the specific class of machine learning models described in this section.

In the context of supervised learning, an *energy-based model* (EBM) is specified by three variables: a parameter variable θ, an input variable x, and a state variable s. An essential ingredient of an EBM is the *energy function*, which is a scalar function E that specifies how the state s depends on the parameter θ and the input x. Given θ and x, the energy

function associates to each *conceivable* configuration s a real number $E(\theta, x, s)$. Among all conceivable configurations, the *effective* configuration of the system is by definition a state $s(\theta, x)$ such that

$$\frac{\partial E}{\partial s}(\theta, x, s(\theta, x)) = 0. \tag{2.3}$$

We call $s(\theta, x)$ an *equilibrium state* of the system. The aim is to minimize the loss at equilibrium:

$$\mathcal{L}(\theta, x, y) = C(s(\theta, x), y). \tag{2.4}$$

In this expression, $s(\theta, x)$ is the 'prediction' from the model, and plays the role of the 'output' $f(\theta, x)$ of the previous section. A conceptual difference between $s(\theta, x)$ and $f(\theta, x)$ is that, in conventional deep learning, $f(\theta, x)$ is usually thought of as the output layer of the model (i.e. the last layer of the neural network), whereas here $s(\theta, x)$ represents the entire state of the system. Another difference is that $f(\theta, x)$ is usually *explicitly* determined by θ and x through an analytical formula, whereas here $s(\theta, x)$ is *implicitly* specified through the variational equation of Eq. (2.3) and may not be expressible by an analytical formula in terms of θ and x. In particular, there exists in general several such states $s(\theta, x)$ that satisfy Eq. (2.3). We further point out that $s(\theta, x)$ need not be a minimum of the energy function E ; it may be a maximum or more generally any saddle point of E.

We note that, just like the energy function E, the cost function C is defined for any conceivable configuration s, not just the equilibrium state $s(\theta, x)$. Although $C(s, y)$ may depend on the entire state s, in practical situations that we will study in the next chapters, $C(s, y)$ depends only on a subset of s that plays the role of 'outputs'.

We also introduce another key concept: the concept of *sum-separability*. Let $\theta = (\theta_1, \ldots, \theta_N)$ be the adjustable parameters of the system. For each θ_k, we denote $\{x, s\}_k$ the information about (x, s) which is locally available to θ_k. We say that the energy function E is *sum-separable* if it is of the form

$$E(\theta, x, s) = E_0(x, s) + \sum_{k=1}^{N} E_k(\theta_k, \{x, s\}_k), \tag{2.5}$$

where $E_0(x, s)$ is a term that is independent of the parameters to be adjusted, and E_k is a scalar function of θ_k and $\{x, s\}_k$, for each $k \in \{1, \ldots, N\}$. Importantly, many physical systems are energy-based models, many of which have the sum-separability property; we give examples in section 2.5.

2.3. Gradient Formula

The central ingredient of the equilibrium propagation training method is the *total energy function* F, defined by $F = E + \beta\, C$, where β is a real-valued variable called *nudging factor*. The intuition here is that we augment the energy of the system by bringing an additional

energy term βC. By varying β, the total energy F is modified, and so is the equilibrium state relative to F. Specifically, assuming that the functions E and C are continuously differentiable, there exists a continuous mapping $\beta \mapsto s_\star^\beta$ such that $s_\star^0 = s(\theta, x)$ and[1]

$$\frac{\partial E}{\partial s}(\theta, x, s_\star^\beta) + \beta \, \frac{\partial C}{\partial s}(s_\star^\beta, y) = 0 \tag{2.6}$$

for any value of the nudging factor β. Theorem 2.1 provides a formula to compute the loss gradients by varying the nudging factor β.

Theorem 2.1 (Gradient formula for energy-based models)**.** *The gradient of the loss is equal to*

$$\frac{\partial \mathcal{L}}{\partial \theta}(\theta, x, y) = \left. \frac{d}{d\beta} \right|_{\beta=0} \frac{\partial E}{\partial \theta}\left(\theta, x, s_\star^\beta\right). \tag{2.7}$$

Furthermore, if the energy function E is sum-separable, then the loss gradient for each parameter θ_k depends only on information that is locally available to θ_k:

$$\frac{\partial \mathcal{L}}{\partial \theta_k}(\theta, x, y) = \left. \frac{d}{d\beta} \right|_{\beta=0} \frac{\partial E_k}{\partial \theta_k}\left(\theta_k, \{x, s_\star^\beta\}_k\right). \tag{2.8}$$

Proof. Eq. (2.7) is a consequence of Lemma 2.2 (Section 2.6) applied to the total energy function[2] F, defined for a fixed input-target pair (x, y) by $F(\theta, \beta, s) = E(\theta, x, s) + \beta \, C(s, y)$, at the point $\beta = 0$. Eq. (2.8) is a consequence of Eq. (2.7) and the definition of sum-separability (Eq. (2.5)). □

2.4. Equilibrium Propagation

We can use Theorem 2.1 to derive a learning algorithm for energy-based models. Let us assume that the energy function is sum-separable. We can estimate the loss gradients using finite differences, for example with

$$\widehat{\nabla}_{\theta_k}(\beta) = \frac{1}{\beta}\left(\frac{\partial E_k}{\partial \theta_k}(\theta_k, \{x, s_\star^\beta\}_k) - \frac{\partial E_k}{\partial \theta_k}(\theta_k, \{x, s_\star^0\}_k)\right) \tag{2.9}$$

to approximate the right-hand side of Eq. (2.8). We arrive at the following two-phase training procedure to update the parameters in proportion to their loss gradients.

Free phase (inference). The nudging factor β is set to zero, and the system settles to an equilibrium state $s_\star^0$, characterized by Eq. (2.3). We call $s_\star^0$ the *free state*. For each parameter θ_k, the quantity $\frac{\partial E_k}{\partial \theta_k}\left(\theta_k, \{x, s_\star^0\}_k\right)$ is measured locally and stored locally.

[1] We note that it is also possible to define $s_\star^\beta$ differently, by the relationship $\frac{\partial E}{\partial s}(\theta, x, s_\star^\beta) + \beta \, \frac{\partial C}{\partial s}(s(\theta, x), y) = 0$, without changing the conclusions of Theorem 2.1. In Chapter 4 we will use this modified definition of $s_\star^\beta$.

[2] With the modified definition of $s_\star^\beta$, the total energy function to consider is $F(\theta, \beta, s) = E(\theta, x, s) + \beta \, C(s(\theta, x), y)$.

Nudged phase. The nudging factor β is set to a nonzero value (positive or negative), and the system settles to a new equilibrium state $s_\star^\beta$, characterized by Eq. (2.6). We call $s_\star^\beta$ the *nudged state*. For each parameter θ_k, the quantity $\frac{\partial E_k}{\partial \theta_k}\left(\theta_k, \{x, s_\star^\beta\}_k\right)$ is measured locally.

Update rule. Finally, each parameter θ_k is updated locally in proportion to its gradient as $\Delta\theta_k = -\eta\widehat{\nabla}_{\theta_k}(\beta)$, where η is a learning rate and $\widehat{\nabla}_{\theta_k}(\beta)$ is the gradient estimator of Eq. (2.9).

The training scheme described above is natural because the free phase and the nudged phase can be related to the standard training procedure for neural networks (the backpropagation algorithm), in which there is an inference phase (forward pass) followed by a gradient computation phase (backward pass). However, due to the approximation of derivatives by finite differences, the gradient estimator prescribed by the above training scheme is biased. As detailed in Appendix A, the mismatch between this gradient estimator ($\widehat{\nabla}_{\theta_k}(\beta)$) and the true gradient ($\frac{\partial \mathcal{L}}{\partial \theta_k}$) is of the order $O(\beta)$. As proposed in Laborieux et al. [2021], this bias can be reduced by means of a symmetric gradient estimator:

$$\widehat{\nabla}^{\text{sym}}_{\theta_k}(\beta) = \frac{1}{2\beta}\left(\frac{\partial E_k}{\partial \theta_k}\left(\theta_k, \{x, s_\star^\beta\}_k\right) - \frac{\partial E_k}{\partial \theta_k}\left(\theta_k, \{x, s_\star^{-\beta}\}_k\right)\right). \tag{2.10}$$

To achieve this, we can modify the above training procedure to include two nudged phases: one with positive nudging ($+\beta$) and one with negative nudging ($-\beta$). The update rule for parameter θ_k is then $\Delta\theta_k = -\eta\widehat{\nabla}^{\text{sym}}_{\theta_k}(\beta)$. The mismatch between this symmetric gradient estimator and the true gradient is only of the order $O(\beta^2)$. We note that higher order methods which use more point values of β are also possible, to further reduce the gradient estimator bias (e.g. with $+2\beta$, $+\beta$, $-\beta$ and -2β).

We call such training procedures *equilibrium propagation* (EqProp), with the intuition that the equilibrium state $s_\star^\beta$ 'propagates' across the system as β is varied.

2.5. Examples of Sum-Separable Energy-Based Models

We give here a few examples of sum-separable energy-based models. As a first example, the Hopfield model is useful to develop intuitions, as it is a well-known and well-studied model in both the machine learning literature and the neuroscience literature ; the case of continuous Hopfield networks will be developed in Chapter 3. We then briefly present resistive networks, which will be developed in Chapter 4. Nonlinear resistive networks are potentially promising for the development of neuromorphic hardware, towards the goal of building fast and energy efficient neural networks. We also briefly present another couple of instances of energy-based physical systems, such as flow networks, and elastic networks. All these systems can be trained with EqProp.

Hopfield networks. In the Hopfield model [Hopfield, 1982] and its continuous version [Cohen and Grossberg, 1983, Hopfield, 1984], neurons are interconnected via bi-directional synapses. Each neuron i is characterised by a scalar s_i, and each synapse connecting neurons i and j is characterised by a number W_{ij} representing the synaptic strength. The energy function of the model, called *Hopfield energy*, is of the form

$$E(\theta, s) = -\sum_{i,j} W_{ij} s_i s_j \tag{2.11}$$

(or a variant of Eq. 2.11). In this expression, the vector of neural states $s = (s_1, s_2, \ldots, s_N)$ represents the state variable of the system, and the vector of synaptic strengths $\theta = \{W_{ij}\}_{i,j}$ represents the parameter variable (the set of adjustable parameters). At inference, neurons stabilize to a minimum of the energy function, where the condition $\frac{\partial E}{\partial s} = 0$ is met. Furthermore, the Hopfield energy is sum-separable with each factor of the form $E_{ij}(W_{ij}, s_i, s_j) = -W_{ij} s_i s_j$. Since the energy gradients are equal to $\frac{\partial E_{ij}}{\partial W_{ij}} = -s_i s_j$, the Hopfield model can be trained with EqProp using a sort of contrastive Hebbian learning rule (Chapter 3).

Resistive networks. A linear resistance network is an electrical circuit composed of nodes interconnected by linear resistors. Let N be the number of nodes in the circuit, and denote $V = (V_1, \ldots, V_N)$ the vector of node voltages. Since the power dissipated in a resistor of conductance g_{ij} is $\mathcal{P}_{ij} = g_{ij} (V_j - V_i)^2$, where V_i and V_j are the terminal voltages of the resistor, the total power dissipated in the circuit is

$$\mathcal{P}(\theta, V) = \sum_{i,j} g_{ij} (V_j - V_i)^2 , \tag{2.12}$$

where $\theta = \{g_{ij}\}_{i,j}$ is the set of conductances of the circuit, which plays the role of 'adjustable parameters'. Notably, linear resistance networks satisfy the so-called *principle of minimum dissipated power*: if the voltages are imposed at a set of input nodes, then the circuit chooses the voltages at other nodes so as to minimize the total power dissipated ($\mathcal{P}$). This implies in particular that $\frac{\partial \mathcal{P}}{\partial V_i} = 0$ for any floating node voltage V_i. Thus, linear resistance networks are energy-based models, with $\mathcal{P}$ playing the role of 'energy function'. Furthermore, the function $\mathcal{P}$ has the sum-separability property, with each factor of the form $\mathcal{P}_{ij}(g_{ij}, V_i, V_j) = g_{ij}(V_i - V_j)^2$, and each gradient equal to $\frac{\partial \mathcal{P}_{ij}}{\partial g_{ij}} = (V_i - V_j)^2$. Crucially, as we will see in Chapter 4, in circuits consisting of arbitrary resistive devices, there exists a generalization of the notion of power function $\mathcal{P}$ called *co-content* [Millar, 1951]. Such circuits, called *nonlinear resistive networks*, can implement analog neural networks, using memristors (to implement the synaptic weights), diodes (to play the role of nonlinearities), voltage sources (to set the voltages of input nodes) and current sources (to inject loss gradients as currents during training).

Flow networks. The EqProp framework may also have implications in other areas of engineering, beyond neuromorphic computing. For example, Stern et al. [2020] study the case of *flow networks*, e.g. networks of nodes interconnected by pipes. This setting is analogous to the case of resistive networks described above. In a flow network, each node i is described by its pressure p_i, and each pipe connecting node i to node j is characterized by its conductance k_{ij}. The total dissipated power in the network, which is minimized, is $\mathcal{P}(\theta, p) = \sum_{i,j} k_{ij} (p_j - p_i)^2$, where $\theta = \{k_{ij}\}$ is the set of parameters to be adjusted, and $p = \{p_{ij}\}$ plays the role of the state variable of the system.

Elastic networks. Stern et al. [2020] also study the case of *central force spring networks*. In this setting, we have a set of N nodes interconnected by linear springs. Each node i is characterized by its 3D position s_i. The elastic energy stored in the spring connecting node i to node j is $E_{ij} = \frac{1}{2} k_{ij} (r_{ij} - \ell_{ij})^2$, where k_{ij} is the spring constant, ℓ_j is the spring's equilibrium length, and $r_{ij} = \|s_i - s_j\|$ is the Euclidean distance between nodes i and j. Thus, the total elastic energy stored in the network, which is minimized, is given by

$$E(\theta, r) = \sum_{i,j} \frac{1}{2} k_{ij} (r_{ij} - \ell_{ij})^2 , \tag{2.13}$$

where $\theta = \{k_{ij}, \ell_{ij}\}$ is the set of adjustable parameters, and $r = \{r_{ij}\}$ plays the role of state variable. The energy gradients in this case are $\frac{\partial E_{ij}}{\partial k_{ij}} = \frac{1}{2} (r_{ij} - \ell_{ij})^2$ and $\frac{\partial E_{ij}}{\partial \ell_{ij}} = k_{ij} (\ell_{ij} - r_{ij})$.

2.6. Fundamental Lemma

In this section, we present the fundamental lemma of the equilibrium propagation framework, from which Theorem 2.1 derives.

Lemma 2.2 (Scellier and Bengio [2017])**.** *Let $F(\theta, \beta, s)$ be a twice differentiable function of the three variables θ, β and s. For fixed θ and β, let s_θ^β be a point that satisfies the stationarity condition*

$$\frac{\partial F}{\partial s}(\theta, \beta, s_\theta^\beta) = 0, \tag{2.14}$$

and suppose that $\frac{\partial^2 F}{\partial s^2}(\theta, \beta, s_\theta^\beta)$ is invertible. Then, in the neighborhood of this point, we can define a continuously differentiable function $(\theta, \beta) \mapsto s_\theta^\beta$ such that Eq. 2.14 holds for any (θ, β) in this neighborhood. Furthermore, we have the following identity:

$$\frac{d}{d\theta} \frac{\partial F}{\partial \beta}(\theta, \beta, s_\theta^\beta) = \frac{d}{d\beta} \frac{\partial F}{\partial \theta}(\theta, \beta, s_\theta^\beta). \tag{2.15}$$

Proof of Lemma 2.2. The first statement follows from the implicit function theorem. It remains to prove Eq. 2.15. Let us consider $F\left(\theta, \beta, s_\theta^\beta\right)$ as a function of (θ, β) (not only through $F(\theta, \beta, \cdot)$ but also through s_θ^β). Using the chain rule of differentiation and the

stationary condition of Eq. (2.14), we have

$$\frac{d}{d\beta}F(\theta,\beta,s_\theta^\beta) = \frac{\partial F}{\partial \beta}\left(\theta,\beta,s_\theta^\beta\right) + \underbrace{\frac{\partial F}{\partial s}\left(\theta,\beta,s_\theta^\beta\right)}_{=0}\cdot\frac{\partial s_\theta^\beta}{\partial \beta}. \tag{2.16}$$

Similarly, we have

$$\frac{d}{d\theta}F(\theta,\beta,s_\theta^\beta) = \frac{\partial F}{\partial \theta}\left(\theta,\beta,s_\theta^\beta\right) + \underbrace{\frac{\partial F}{\partial s}\left(\theta,\beta,s_\theta^\beta\right)}_{=0}\cdot\frac{\partial s_\theta^\beta}{\partial \theta}. \tag{2.17}$$

Combining these equations and using the symmetry of second-derivatives, we get:

$$\frac{d}{d\theta}\frac{\partial F}{\partial \beta}(\theta,\beta,s_\theta^\beta) = \frac{d}{d\theta}\frac{d}{d\beta}F(\theta,\beta,s_\theta^\beta) = \frac{d}{d\beta}\frac{d}{d\theta}F(\theta,\beta,s_\theta^\beta) = \frac{d}{d\beta}\frac{\partial F}{\partial \theta}(\theta,\beta,s_\theta^\beta). \tag{2.18}$$

□

2.7. Remarks

Stationary points. In the static setting presented in this chapter, EqProp applies to any system whose equilibrium states satisfy the stationary condition of Eq 2.3 – what we have called an *energy-based model* (EBM). While early works [Scellier and Bengio, 2017, 2019] proposed to apply EqProp to EBMs whose equilibrium states are minima of the energy function (e.g. Hopfield networks), we stress here that the equilibrium states may more generally be any stationary points (saddle points or maxima) of the energy function. We note that the landscape of the energy function can contain in general exponentially many more stationary points than (local) minima. For instance, the recently introduced 'modern Hopfield networks' [Ramsauer et al., 2020] are EBMs in the sense of Eq 2.3.

Infinite dimensions. In section 2.5, we have given examples of EBMs in which the state variable s has finitely many dimensions. We note however that s may also belong to an infinite dimensional space (mathematically, a Banach space). In this case, the expression $\frac{\partial F}{\partial s}\left(\theta,\beta,s_\theta^\beta\right)\cdot\frac{\partial s_\theta^\beta}{\partial \beta}$ in Lemma 2.2 must be thought of as the differential of the function $F\left(\theta,\beta,\cdot\right)$ at the point s_θ^β, applied to the vector $\frac{\partial s_\theta^\beta}{\partial \beta}$.

Variational principles of physics. In physics, many systems can be described by a variational equation of the form of Eq. 2.3 ; we have given examples in Section 2.5. In fact, the framework presented here transfers directly to time-varying physical systems (Chapter 6). In this case, s must be thought of as the *trajectory* of the system, E as a functional called *action functional*, and the stationary condition of Eq. 2.3 as a *principle of stationary action*.

Singularities. We have proved Theorem 2.1 using Lemma 2.2. Yet the formula of Lemma 2.2 assumes that the Hessian $\frac{\partial^2 E}{\partial s^2}(\theta, x, s(\theta, x))$ is invertible. While this assumption is likely to be valid at most iterations of training, it is also likely that, as θ evolves during training, θ goes through values where the Hessian of the energy is singular for some input x. At such points, it is not clear how the update rule of EqProp behaves. One branch of mathematics that studies these aspects is *Catastrophe Theory*. Although these aspects raise interesting questions, diving into these questions would take us far from the main thrust of this manuscript. In this manuscript, we pretend everything is differentiable.

Beyond supervised learning. Although we have focused on supervised learning, the framework presented in this chapter can be adapted to other machine learning paradigms. For example, we note that the formula of Theorem 2.1 can be directly transposed to compute the loss gradients with respect to input variables of the network:

$$\frac{\partial \mathcal{L}}{\partial x}(\theta, x, y) = \left.\frac{d}{d\beta}\right|_{\beta=0} \frac{\partial E}{\partial x}\left(\theta, x, s_\star^\beta\right). \tag{2.19}$$

This formula may be useful in applications where one wants to do gradient descent in the input space, e.g. image synthesis. This may also be useful in the setting of generative adversarial networks [Goodfellow et al., 2014], in which we need to compute the loss gradients with respect to inputs of the discriminator network, to further propagate error signals in the generator network. The framework presented in this chapter may also be adapted to model-free reinforcement learning algorithms such as temporal difference (TD) learning (e.g. Q-learning). Finally, the EqProp training procedure has also been used in the meta-learning setting to train the meta-parameters of a model, a method called *contrastive meta-learning* [Zucchet et al., 2021]. We briefly present this framework in section 6.3.

Chapter 3

Training Continuous Hopfield Networks with Equilibrium Propagation

In the previous chapter, we have presented equilibrium propagation (EqProp) as a general learning algorithm for energy-based models. In this chapter, we use EqProp to train a class of energy-based models called gradient systems. In particular we apply EqProp to continuous Hopfield networks, a class of of neural networks that has inspired neuroscience and neuromorphic computing since the 1980s. The present chapter is essentially a compilation of Scellier and Bengio [2017, 2019], and is organized as follows.

- In section 3.1, we apply EqProp to a class of continuous-time dynamical systems called *gradient systems*. In a gradient system, the state dynamics descend the gradient of a scalar function (the *energy function*) and stabilise to a minimum of that energy function. Thus, in this setting, equilibrium states correspond to energy minima. We provide an analytical formula for the transient states of the system between the free state and the nudged state of the EqProp training process, and we link these transient states to the recurrent backpropagation algorithm of Almeida [1987] and Pineda [1987].
- In section 3.2, we apply EqProp to the continuous Hopfield model, an energy-based neural network model described by an energy function called the *Hopfield energy*. The gradient dynamics associated with the Hopfield energy yields the neural dynamics in Hopfield networks: neurons are seen as leaky integrator neurons, with the constraint that synapses are bidirectional and symmetric. In addition, the update rule of EqProp for each synapse is local (more specifically Hebbian).
- In section 3.3, we present numerical experiments on deep Hopfield networks trained with EqProp on the MNIST digit classification task.
- In section 3.4, we study the relationship between EqProp and the contrastive Hebbian learning algorithm of Movellan [1991].

3.1. Gradient Systems

In this section, we present a theoretical result which holds for arbitrary energy functions and cost functions. This section, which deals with the concepts of energy and cost functions in the abstract, is largely independent of the rest of this chapter. The reader who is eager to see how Hopfield networks can be trained with EqProp may skip this section and go straight to the next one.

3.1.1. Gradient Systems as Energy-Based Models

We have seen in Chapter 2 that EqProp is an algorithm to train systems that possess *equilibrium states*, i.e. states characterized by a variational equation of the form $\frac{\partial E}{\partial s}(\theta, x, s_\star) = 0$, where $E(\theta, x, s)$ is a scalar function called *energy function*. Recall that θ is the set of adjustable parameters of the system, and x is an input. We have called such systems *energy-based models*. The class of energy-based models that we study here is that of systems whose dynamics spontaneously minimizes the energy function E by following its gradient. In such a system, called *gradient system*, the state follows the dynamics

$$\frac{ds_t}{dt} = -\frac{\partial E}{\partial s}(\theta, x, s_t). \tag{3.1}$$

Here s_t denotes the state of the system at time t. The energy of the system decreases until $\frac{ds_t}{dt} = 0$, and the equilibrium state $s_\star$ reached after convergence of the dynamics is an energy minimum (either local or global). The function E is also sometimes called a *Lyapunov function* for the dynamics of s_t.

In this setting, equilibrium states are *stable*: if the state is slightly perturbed around equilibrium, the dynamics will tend to bring the system back to equilibrium. For this reason, such equilibrium states are also called 'attractors' or 'retrieval states', because the system's dynamics can 'retrieve' them if they are only partially known. Thus, gradient systems can recover incomplete data, by storing 'memories' in their point attractors.

In this manuscript, we are more specifically interested in the supervised learning problem, where the loss to optimize is of the form

$$\mathcal{L} = C(s_\star, y). \tag{3.2}$$

After training is complete, the model can be used to 'retrieve' a label y associated to a given input x.

3.1.2. Training Gradient Systems with Equilibrium Propagation

In a gradient system, EqProp takes the following form. In the first phase, or free phase, the state of the system follows the gradient of the energy (Eq. 3.1). At the end of the first

phase the system is at equilibrium ($s_\star$). In the second phase, or nudged phase, starting from the equilibrium state $s_\star$, a term $-\beta \frac{\partial C}{\partial s}$ (where $\beta > 0$ is a hyperparameter called *nudging factor*) is added to the dynamics of the state and acts as an external force nudging the system dynamics towards decreasing the cost C. Denoting s_t^β the state of the system at time t in the second phase (which depends on the value of the nudging factor β), the dynamics is defined as[1]

$$s_0^\beta = s_\star \qquad \text{and} \qquad \forall t \geq 0, \quad \frac{ds_t^\beta}{dt} = -\frac{\partial E}{\partial s}\left(\theta, x, s_t^\beta\right) - \beta \frac{\partial C}{\partial s}\left(s_t^\beta, y\right). \tag{3.3}$$

The system eventually settles to a new equilibrium state $s_\star^\beta$. Recall from Theorem 2.1 that the gradient of the loss $\mathcal{L}$ can be estimated based on the two equilibrium states $s_\star$ and $s_\star^\beta$. Specifically, in the limit $\beta \to 0$,

$$\lim_{\beta \to 0} \frac{1}{\beta}\left(\frac{\partial E}{\partial \theta}\left(x, s_\star^\beta, \theta\right) - \frac{\partial E}{\partial \theta}\left(\theta, x, s_\star\right)\right) = \frac{\partial \mathcal{L}}{\partial \theta}. \tag{3.4}$$

Furthermore, if the energy function has the sum-separability property (as defined by Eq. 2.5), then the learning rule for each parameter θ_k is local:

$$\lim_{\beta \to 0} \frac{1}{\beta}\left(\frac{\partial E_k}{\partial \theta_k}\left(\theta_k, \{x, s_\star^\beta\}_k\right) - \frac{\partial E_k}{\partial \theta_k}\left(\theta_k, \{x, s_\star\}_k\right)\right) = \frac{\partial \mathcal{L}}{\partial \theta_k}. \tag{3.5}$$

3.1.3. Transient Dynamics

Note that the learning rule of Eqs. 3.4-3.5 only depends on the equilibrium states $s_\star$ and $s_\star^\beta$, not on the specific trajectory that the system follows to reach them. Indeed, as we have seen in the previous chapter, EqProp applies to any energy based model, not just gradient systems. But under the assumption of a gradient dynamics, we can say more about the transient states, when the system gradually moves from the free state ($s_\star$) towards the nudged state ($s_\star^\beta$): we show that the transient states (s_t^β for $t \geq 0$) perform gradient computation with respect to a function called the *projected cost function*.

Recall that in the free phase, the system follows the dynamics of Eq. 3.1. In particular, the state s_t at time $t \geq 0$ depends not just on θ and x, but also on the initial state s_0 at time $t = 0$. Let us define the *projected cost function*

$$L_t(\theta, s_0) = C\left(s_t\right), \tag{3.6}$$

where we omit x and y for brevity of notations. $L_t(\theta, s_0)$ is the cost of the state projected a duration t in the future, when the system starts from s_0 and follows the dynamics of the free phase. Note that L_t depends on θ and s_0 (as well as x) implicitly through s_t. For fixed s_0, the process $(L_t(\theta, s_0))_{t \geq 0}$ represents the successive cost values taken by the state of the

[1] As discussed in the case of Eq. 2.6, we can also define $\frac{ds_t^\beta}{dt} = -\frac{\partial E}{\partial s}\left(\theta, x, s_t^\beta\right) - \beta \frac{\partial C}{\partial s}\left(s_\star, y\right)$ without changing the conclusions of the theoretical results.

system along the free dynamics when it starts from the initial state s_0. In particular, for $t = 0$, the projected cost is simply the cost of the initial state, i.e. $L_0(\theta, s_0) = C(s_0)$. As $t \to \infty$, we have $s_t \to s_\star$ and therefore $L_t(\theta, s_0) \to C(s_\star) = \mathcal{L}(\theta)$, i.e. the projected cost converges to the loss at equilibrium.

The following result shows that the transient states of EqProp (s_t^β) can be expressed in terms of the projected cost function (L_t), when $s_0 = s_\star$.

Theorem 3.1 (Scellier and Bengio [2019])**.** *The following identities hold for any $t \geq 0$:*

$$\lim_{\beta \to 0} \frac{1}{\beta} \left(\frac{\partial E}{\partial \theta} \left(\theta, s_t^\beta \right) - \frac{\partial E}{\partial \theta} \left(\theta, s_\star \right) \right) = \frac{\partial L_t}{\partial \theta} \left(\theta, s_\star \right), \tag{3.7}$$

$$\lim_{\beta \to 0} \frac{1}{\beta} \frac{ds_t^\beta}{dt} = -\frac{\partial L_t}{\partial s} \left(\theta, s_\star \right). \tag{3.8}$$

Furthermore, if the energy function has the sum-separability property (as defined by Eq. 2.5), then

$$\lim_{\beta \to 0} \frac{1}{\beta} \left(\frac{\partial E_k}{\partial \theta_k} \left(\theta_k, \{ s_t^\beta \}_k \right) - \frac{\partial E_k}{\partial \theta_k} \left(\theta_k, \{ s_\star \}_k \right) \right) = \frac{\partial L_t}{\partial \theta_k} \left(\theta, s_\star \right). \tag{3.9}$$

Proof. Eq. 3.7-3.8 follow directly from Lemma 3.2 (next subsection). Eq. 3.9 follows from Eq. 3.7 and the definition of sum-separability. □

The left-hand-side of Eq. 3.7 represents the gradient provided by EqProp if we substitute s_t^β to $s_\star^\beta$ in the gradient formula (Eq. 3.4). This corresponds to a *truncated* version of EqProp, where the second phase (nudged phase) is halted before convergence to the nudged equilibrium state. Eq. 3.7 provides an analytical formula for this truncated gradient in terms of the projected cost function, when $s_0 = s_\star$

The left-hand side of Eq. 3.8 is the temporal derivative of s_t^β rescaled by a factor $\frac{1}{\beta}$. In essence, Eq. 3.8 shows that, in the second phase of EqProp (nudged phase), the temporal derivative of the state *codes* for gradient information (namely the gradients of the projected cost function, when $s_0 = s_\star$).

3.1.4. Recurrent Backpropagation

In this section, we prove Theorem 3.1. In doing so, we also establish a link between EqProp and the recurrent backpropagation algorithm of Almeida [1987] and Pineda [1987], which we briefly present below.

First, let us introduce the temporal processes $(\overline{S}_t, \overline{\Theta}_t)$ and $(\widetilde{S}_t, \widetilde{\Theta}_t)$ defined by

$$\forall t \geq 0, \qquad \overline{S}_t = \frac{\partial L_t}{\partial s} \left(\theta, s_\star \right), \qquad \overline{\Theta}_t = \frac{\partial L_t}{\partial \theta} \left(\theta, s_\star \right), \tag{3.10}$$

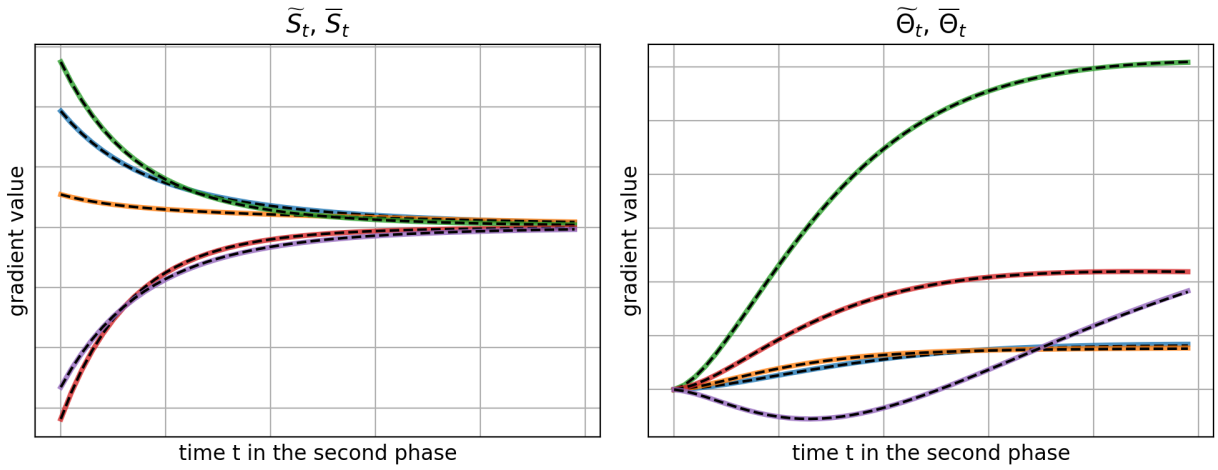

Fig. 4. Illustration of Theorem 3.1 on a toy example. Dashed lines (in black) represent five randomly chosen coordinates of $\widetilde{S}_t$ (left) and five randomly chosen coordinates of $\widetilde{\Theta}_t$ (right). Solid colored lines represent the corresponding coordinates in $\overline{S}_t$ (left) and in $\overline{\Theta}_t$ (right). The processes $\overline{S}_t$, $\widetilde{S}_t$, $\overline{\Theta}_t$ and $\widetilde{\Theta}_t$ are defined by Eqs. 3.10-3.11. The figure was produced by modifying the code[2] of Ernoult et al. [2019].

and

$$\forall t \geq 0, \qquad \widetilde{S}_t = -\lim_{\beta \to 0} \frac{1}{\beta} \frac{ds_t^\beta}{dt}, \qquad \widetilde{\Theta}_t = \lim_{\beta \to 0} \frac{1}{\beta} \left(\frac{\partial E}{\partial \theta} \left(\theta, s_t^\beta \right) - \frac{\partial E}{\partial \theta} \left(\theta, s_\star \right) \right). \tag{3.11}$$

The processes $\overline{S}_t$ and $\widetilde{S}_t$ take values in the state space (space of the state variable s). The processes $\overline{\Theta}_t$ and $\widetilde{\Theta}_t$ take values in the parameter space (space of the parameter variable θ). Using these notations, Theorem 3.1 states that $\overline{S}_t = \widetilde{S}_t$ and $\overline{\Theta}_t = \widetilde{\Theta}_t$ for every $t \geq 0$. These identities are a direct consequence of the following lemma.

Lemma 3.2. *Both the processes $(\overline{S}_t, \overline{\Theta}_t)$ and $(\widetilde{S}_t, \widetilde{\Theta}_t)$ are solutions of the same (linear) differential equation:*

$$S_0 = \frac{\partial C}{\partial s} \left(s_\star \right), \tag{3.12}$$

$$\Theta_0 = 0, \tag{3.13}$$

$$\frac{d}{dt} S_t = -\frac{\partial^2 E}{\partial s^2} \left(\theta, s_\star \right) \cdot S_t, \tag{3.14}$$

$$\frac{d}{dt} \Theta_t = -\frac{\partial^2 E}{\partial \theta \partial s} \left(\theta, s_\star \right) \cdot S_t. \tag{3.15}$$

By uniqueness of the solution, the processes $(\overline{S}_t, \overline{\Theta}_t)$ and $(\widetilde{S}_t, \widetilde{\Theta}_t)$ are equal.

We refer to Scellier and Bengio [2019] for a proof of this result. Since the differential equation of Lemma 3.2 is linear with constant coefficients, we can express S_t and Θ_t using

[2] https://github.com/ernoult/updatesEPgradientsBPTT

closed-form formulas. Specifically $S_t = \exp\left(-t\frac{\partial^2 E}{\partial s^2}(\theta, s_\star)\right) \cdot \frac{\partial C}{\partial s}(s_\star)$ and $\Theta_t = \frac{\partial^2 E}{\partial \theta \partial s}(\theta, s_\star) \cdot \left(\frac{\partial^2 E}{\partial s^2}(\theta, s_\star)\right)^{-1} \cdot \left[\mathrm{Id} - \exp\left(-t\frac{\partial^2 E}{\partial s^2}(\theta, s_\star)\right)\right] \cdot \frac{\partial C}{\partial s}(s_\star)$.

Lemma 3.2 also suggests an alternative procedure to compute the parameter gradients of the loss $\mathcal{L}$ numerically. This procedure, known as *Recurrent Backpropagation* (RBP), was introduced independently by Almeida [1987] and Pineda [1987]. Specifically, RBP consists of the following two phases. The first phase is the same as the free phase of EqProp: s_t follows the free dynamics (Eq. 3.1) and relaxes to the equilibrium state $s_\star$. The state $s_\star$ is necessary for evaluating $\frac{\partial^2 E}{\partial s^2}(\theta, s_\star)$ and $\frac{\partial^2 E}{\partial \theta \partial s}(\theta, s_\star)$, which the second phase requires. In the second phase, S_t and Θ_t are computed iteratively for increasing values of t using Eq. 3.12-3.15. Finally, Θ_t provides the desired loss gradient in the limit $t \to \infty$. To see this, we first note that Lemma 3.2 tells us that the vector Θ_t computed by this procedure is equal to $\widetilde{\Theta}_t$ for any $t \geq 0$. Then by definition, $\widetilde{\Theta}_t = \lim_{\beta \to 0} \frac{1}{\beta}\left(\frac{\partial E}{\partial \theta}\left(\theta, s_t^\beta\right) - \frac{\partial E}{\partial \theta}(\theta, s_\star)\right)$. It follows that, as $t \to \infty$, we have $\widetilde{\Theta}_t \to \lim_{\beta \to 0} \frac{1}{\beta}\left(\frac{\partial E}{\partial \theta}\left(\theta, s_\star^\beta\right) - \frac{\partial E}{\partial \theta}(\theta, s_\star)\right) = \frac{\partial \mathcal{L}}{\partial \theta}$.

An important benefit of EqProp over RBP it that EqProp requires only one kind of dynamics for both phases of training. RBP requires a special computational circuit in the second phase for computing the gradients.

The original RBP algorithm was described for a general state-to-state dynamics. Here, we have presented RBP in the particular case of gradient dynamics. We refer to LeCun et al. [1988] for a more general derivation of RBP based on the adjoint method.

3.2. Continuous Hopfield Networks

In the previous section we have presented a theoretical result which is generic and involves the energy function E and cost function C in their abstract form. In this section, we study EqProp in the context of a neural network model called the continuous Hopfield model [Hopfield, 1984].

3.2.1. Hopfield Energy

A Hopfield network is a neural network with the following characteristics. The state of a neuron i is described by a scalar s_i, loosely representing its membrane voltage. The state of a synapse connecting neuron i to neuron j is described by a real number W_{ij} representing its efficacy (or 'strength'). The notation $\sigma(s_i)$ is further used to denote the firing rate of neuron i. The function σ is called *activation function* ; it takes a real number as input, and returns a real number as output. Using the formalism of the previous section, the state of the system is the vector $s = (s_1, s_2, \ldots, s_N)$ where N is the number of neurons in the network, and the set of parameters to be adjusted is $\theta = \{W_{ij}\}_{ij}$. As we will see shortly, one biologically unrealistic requirement of the Hopfield model is that synapses are assumed to

be bidirectional and symmetric: the synapse connecting i to j shares the same weight value as the synapse connecting j to i, i.e. $W_{ij} = W_{ji}$.

Hopfield Energy. Hopfield [1984] introduced the following energy function[3]:

$$E(\theta, s) = \frac{1}{2}\sum_i s_i^2 - \sum_{i<j} W_{ij}\sigma(s_i)\sigma(s_j), \tag{3.16}$$

which we will call the *Hopfield energy*. We calculate

$$-\frac{\partial E}{\partial s_i} = \sigma'(s_i)\left(\sum_{j\neq i} W_{ij}\sigma(s_j)\right) - s_i. \tag{3.17}$$

Thus, the gradient dynamics for neuron s_i with respect to the Hopfield energy is given by the formula $\frac{ds_i}{dt} = \sigma'(s_i)\left(\sum_{j\neq i} W_{ij}\sigma(s_j)\right) - s_i$. This dynamics is reminiscent of the leaky integrator neuron model, a simplified neuron model commonly used in neuroscience. The main difference with the standard leaky-integrator neuron model is the fact that synaptic weights are constrained to be bidirectional and symmetric, a biologically unrealistic constraint often referred to as the *weight transport problem*. Another difference is the presence of the term $\sigma'(s_i)$ which modulates the total input to neuron i.

Squared Error. In the supervised setting that we study here, a set of neurons are input neurons, denoted x, and are always clamped to their input values. Among the 'free' neurons (s), a subset of them are *output neurons* (denoted o), meaning that they represent the network's output. The network's prediction is the state of output neurons at equilibrium (denoted $o_\star$). We call all other neurons the *hidden neurons* and denote them h. Thus, the state of the network is $s = (h, o)$. The cost function considered here is the squared error

$$C(s, y) = \frac{1}{2}\left\|o - y\right\|^2, \tag{3.18}$$

which measures the discrepancy between the state of output neurons (o) and their target values (y).

Total Energy. One of the novelties of EqProp with respect to prior learning algorithms for energy-based models is the *total energy function* F, which takes the form $F = E + \beta\, C$, where β is a real-valued scalar (the *nudging factor*). The function C not only represents the cost to minimize, but also contributes to the total energy of the system by acting like an external potential for the output neurons (o). Thus, the total energy F is the sum of two potential energies: an 'internal potential' (E) that models the interactions within the network, and an 'external potential' ($\beta\, C$) that models how the targets influence the output neurons. The resulting gradient dynamics $\frac{ds_t}{dt} = -\frac{\partial E}{\partial s} - \beta\frac{\partial C}{\partial s}$ consists of two 'forces' which

[3]The energy function of Eq. 3.16 is in fact the one proposed by Bengio et al. [2017]. The energy function introduced by Hopfield is slightly different, but this technical detail is not essential for our purpose.

act on the temporal derivative of s_t. The 'internal force' (induced by E) is that of a leaky integrator neuron (Eq. 3.17). The 'external force' (induced by $\beta\ C$) on $s = (h, o)$ takes the form:

$$-\beta\frac{\partial C}{\partial h} = 0 \qquad \text{and} \qquad -\beta\frac{\partial C}{\partial o} = \beta(y - o). \tag{3.19}$$

This external force acts on output neurons only: it can pull them (if $\beta \geq 0$) towards their target values (y), or repel them (if $\beta \leq 0$). The nudging factor β controls the strength of this interaction between output neurons and targets. In particular, when $\beta = 0$, the output neurons are not sensitive to the targets.

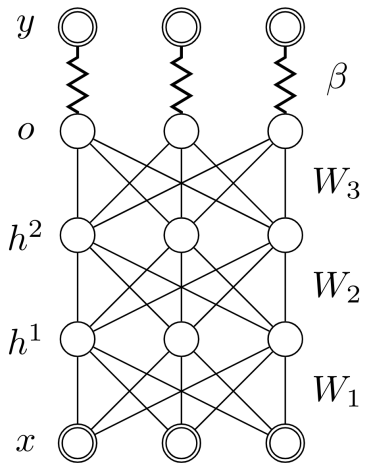

Fig. 5. A deep Hopfield network (DHN). Input x is clamped. Neurons s include "hidden layers" h_1 and h_2, and "output layer" o (the layer where the prediction is read). Target y has the same dimension as output o. Connections between neurons are bidirectional and have symmetric weights. Such a network can be trained with EqProp. In the nudged phase (second phase of training), the nudging factor β scales the "external force" $\beta(y - o)$ that attracts output neurons (o) towards their target values (y).

3.2.2. Training Continuous Hopfield Networks with Equilibrium Propagation

Consider a deep Hopfield network (DHN) of the kind depicted in Figure 5. For each training example (x, y) in the dataset, EqProp training proceeds as follows.

Free Phase. At inference, inputs x are clamped, and both the hidden neurons (h^1 and h^2) and output neurons (o) evolve freely, following the gradient of the energy. The hidden and output neurons subsequently stabilize to an energy minimum, called *free state* and denoted $s_\star = (h^1_\star, h^2_\star, o_\star)$. The state of output neurons at equilibrium ($o_\star$) plays the role of prediction for the model.

Nudged Phase. After relaxation to the free state $s_\star$, the target y is observed, and the nudging factor β takes on a positive value, gradually driving the state of output neurons (o) towards y. Since the external force only acts on the output neurons, the hidden layers (h^1 and h^2) are initially at equilibrium at the beginning of the nudged phase. The perturbation introduced at output neurons gradually propagates backwards along the layers of the network, until the system settles to a new equilibrium state ($s_\star^\beta$).

Proposition 3.3 (Scellier and Bengio [2017])**.** *Denote s_i^0 and s_i^β the free state and nudged state of neuron i, respectively. Then, we have the following formula to estimate the gradient of the loss $\mathcal{L} = \frac{1}{2}\|o_\star - y\|^2$:*

$$\lim_{\beta \to 0} \frac{1}{\beta} \left(\sigma\left(s_i^\beta\right) \sigma\left(s_j^\beta\right) - \sigma\left(s_i^0\right) \sigma\left(s_j^0\right) \right) = -\frac{\partial \mathcal{L}}{\partial W_{ij}}. \tag{3.20}$$

Proof. This is a direct consequence of the main Theorem 2.1, applied to the Hopfield energy function (Eq. 3.16) and the squared error cost function (Eq. 3.18). Notice that the Hopfield energy has the sum-separability property (as defined by Eq. 2.5), with each factor of the form $E_{ij}(W_{ij}, s_i, s_j) = -W_{ij}\sigma(s_i)\sigma(s_j)$. □

Proposition 3.3 suggests for each synapse W_{ij} the update rule

$$\Delta W_{ij} = \frac{\eta}{\beta} \left(\sigma\left(s_i^\beta\right) \sigma\left(s_j^\beta\right) - \sigma\left(s_i^0\right) \sigma\left(s_j^0\right) \right), \tag{3.21}$$

where η is a learning rate. This learning rule is a form of contrastive Hebbian learning (CHL), with a Hebbian term at one equilibrium state, and an anti-Hebbian term at the other equilibrium state. We will discuss in Section 3.4 the relationship between EqProp and the CHL algorithm of Movellan [1991].

3.2.3. 'Backpropagation' of Error Signals

It is interesting to note that EqProp is similar in spirit to the backpropagation algorithm [Rumelhart et al., 1988]. The free phase of EqProp, which corresponds to inference, plays the role of the forward pass in a feedforward net. The nudged phase of EqProp is similar to the backward pass of backpropagation, in that the target output is revealed and it involves the propagation of loss gradient signals. This analogy is even more apparent in a layered network like the one depicted in Fig. 5: in the nudged phase of EqProp, error signals (back-)propagate across the layers of the network, from output neurons to input neurons. Theorem 3.1 gives a more quantitative description of how gradient computation is performed in the nudged phase, with the temporal derivatives of neural activity carrying gradient signals. Thus, like backprop, the learning process in EqProp is driven by an error signal; but unlike backprop, neural computation in EqProp corresponds to both inference and error back-propagation.

The idea that error signals in neural networks can be encoded in the temporal derivatives of neural activity was also explored by Hinton and McClelland [1988], Movellan [1991], O'Reilly [1996], and has been recently formulated as a hypothesis for neuroscience [Lillicrap et al., 2020].

Because error signals are propagated in the network via the neural dynamics, synaptic plasticity can be driven directly by the dynamics of the neurons. Indeed, the global update of EqProp (Eq. 3.21) is equal to the temporal integration of infinitesimal updates

$$dW_{ij} = \frac{\eta}{\beta} d\left(\sigma(s_i)\sigma(s_j)\right) \tag{3.22}$$

over the nudged phase, when the neurons gradually move from their free state ($s_\star$) to their nudged state ($s_\star^\beta$). This suggests an alternative method to implement the global weight update: in the first phase, when the neurons relax to the free state, no synaptic update occurs ($\Delta W_{ij} = 0$) ; in the second phase, the real-time update of Eq. 3.22 is performed when the neurons evolve from their free state to their nudged state. This idea is formalized and tested numerically in Ernoult et al. [2020].

From a biological perspective, perhaps the most unrealistic assumption in this model of credit assignment is the requirement of symmetric weights. This constraint can be relaxed at the cost of computing a biased gradient [Scellier et al., 2018, Ernoult et al., 2020, Laborieux et al., 2021, Tristany et al., 2020].

3.3. Numerical Experiments on MNIST

In this section, we present the experimental results of Scellier and Bengio [2017]. In these simulations, we train deep Hopfield networks of the kind depicted in Fig. 5. Our networks have no skip-layer connections and no lateral connections[4]. We recall that these Hopfield networks, unlike feedforward networks, are recurrently connected, with bidirectional and symmetric connections (i.e. the synapse from neuron i to neuron j shares the same weight value as the synapse from neuron j to neuron i).

We train these Hopfield networks on the MNIST digits classification task [LeCun et al., 1998]. The MNIST dataset (the 'modified' version of the National Institute of Standards and Technology dataset) of handwritten digits is composed of 60,000 training examples and 10,000 test examples. Each example x in the dataset is a 28×28 gray-scaled image and comes with a label $y \in \{0, 1, \ldots, 9\}$ indicating the digit that the image represents. Given an input x, the network's prediction $\widehat{y}$ is the index of the output neuron (among the 10 output

[4]We stress that the models trainable by EqProp are not limited to the chain-like architecture of Fig. 5. Other works have studied the effect of adding skip-layer connections [Gammell et al., 2020] and introducing sparsity [Tristany et al., 2020].

neurons) whose activity at equilibrium is maximal, that is

$$\widehat{y} = \underset{i \in \{0,1,\dots,9\}}{\arg\max}\ o_{\star,i}. \tag{3.23}$$

The network is optimized by stochastic gradient descent (SGD). The process to perform one training iteration on a sample of the training set (i.e. to compute the corresponding gradient and to take one step of SGD) is the one described in section 3.2.2. For efficiency of the experiments, we use minibatches of 20 training examples.

3.3.1. Implementation Details

The hyperparameters chosen for each model are shown in Table 1. The code is available[5].

Architecture. We train deep Hopfield networks with 1, 2 and 3 hidden layers. The input layer consists of $28 \times 28 = 784$ neurons. The hidden layers consist of 500 hidden neurons each. The output layer consists of 10 output neurons.

Weight initialization. The weights of the network are initialized[6] according to the Glorot-Bengio initialization scheme [Glorot and Bengio, 2010], i.e. each weight matrix is initialized by drawing i.i.d. samples uniformly at random in the range $[L, U]$, where $L = -\frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}}$ and $U = \frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}}$, with n_i the fan-in and n_{i+1} the fan-out of the weight matrix.

Implementation of the neural dynamics. Recall that, for a fixed input-target pair (x, y), the total energy is $F(\theta, \beta, s) = E(\theta, x, s) + \beta\ C(s, y)$. We implement the gradient dynamics $\frac{ds_t}{dt} = -\frac{\partial F}{\partial s}(\theta, \beta, s_t)$ using the Euler scheme, meaning that we discretize time into short time lapses of duration ϵ and iteratively update the state of the network (hidden and output neurons) according to

$$s_{t+1} = s_t - \epsilon \frac{\partial F}{\partial s}(\theta, \beta, s_t)\,. \tag{3.24}$$

This process can be thought of as one step of gradient descent (in the state space) on the total energy F, with learning rate ϵ. In practice we find that it is necessary to restrict the space for each state variable (i.e. each neuron) to a bounded interval ; we choose the interval $[0, 1]$. This amounts to use the modified version of the Euler scheme:

$$s_{t+1} = \min\left(\max\left(0, s_t - \epsilon \frac{\partial F}{\partial s}(\theta, \beta, s_t)\right), 1\right). \tag{3.25}$$

We choose $\epsilon = 0.5$ in the simulations. The number of iterations in the free phase is denoted T. The number of iterations in the nudged phase is denoted K.

[5] https://github.com/bscellier/Towards-a-Biologically-Plausible-Backprop

[6] Little is known about how to initialise the weights of recurrent neural networks with static input. More exploration is needed to find appropriate initialisation schemes for such networks.

Number of iterations in the free phase (T). We find experimentally that for the network to be successfully trained, it is necessary that the equilibrium state be reached with very high precision in the free phase (otherwise the gradient estimate of EqProp is unreliable). As a consequence, we require a large number of iterations (denoted T) to reach this equilibrium state. Moreover we find that T grows fast as the number of layers increases (see Table 1). Nevertheless, we will see in Chapter 5 that we can experimentally cut down the number of iterations by a factor five by rewriting the free phase dynamics differently. Importantly, we stress that the large number of time steps required in the free phase is only a concern for computer simulations ; we will see in Chapter 4 that inference can potentially be extremely fast if performed appropriately on analog hardware (by using the physics of the circuit, rather than numerical optimization on conventional computers).

Number of iterations in the nudged phase (K). During the second phase of training, we find experimentally that full relaxation to the nudged equilibrium state is not necessary. This observation is also partly justified by Theorem 3.1, which gives an explicit formula for the 'truncated gradient' provided by EqProp when the nudged phase is halted before convergence. As a heuristic, we choose K (the number of iterations in the nudged phase) proportional to the number of layers, so that the 'error signals' are able to propagate from output neurons back to input neurons.

Nudging factor (β). In spite of its intrinsic bias (Lemma A.1), we find that the one-sided gradient estimator performs well on MNIST (as also observed by Ernoult et al. [2019]). We choose $\beta = 1$ in the experiments. Although it is not crucial, we find that the test accuracy is slightly improved by choosing the sign of β at random in the nudged phase of each training iteration (with probability $p(\beta = 1) = 1/2$ and $p(\beta = -1) = 1/2$). Randomizing β indeed has the effect of cancelling on average the $O(\beta)$-error term of the one-sided gradient estimator.

While this is not necessary on MNIST, we will see in Chapter 5 that on a more complex task such as CIFAR-10, unbiasing the gradient estimator is necessary, and that the symmetric nudging estimator (Eq. 2.10) further helps stabilize training and improve test accuracy.

Learning rates. We find experimentally that we need different learning rates for the weight matrices of different layers. We choose these learning rates heuristically as follows. Denote by $h^0, h^1, \cdots, h^N$ the layers of the network (where $h^0 = x$ and $h^N = o$) and by W_k the weight matrix between the layers h^{k-1} and h^k. We choose the learning rate α_k for W_k proportionally to $\frac{\|W_k\|}{\mathbb{E}\left[\left\|\nabla_{W_k}\right\|\right]}$, where $\mathbb{E}\left[\|\nabla_{W_k}\|\right]$ represents the norm of the EqProp gradient for layer W_k, averaged over training examples.

3.3.2. Experimental Results

Table 1 (top) presents the experimental results of Scellier and Bengio [2017]. These experiments aim at demonstrating that the EqProp training scheme is able to perfectly (over)fit the training dataset, i.e. to get the error rate on the training set down to 0.00%. To achieve this, we use the following trick to reach the equilibrium state of the first phase more easily: at each epoch of training, for each example in the training set, we store the corresponding equilibrium state (i.e. the state of the hidden and output neurons at the end of the free phase), and we use this configuration as a starting point for the next free phase relaxation on that example. This method, which is similar to the PCD (Persistent Contrastive Divergence) algorithm for sampling from the equilibrium distribution of the Boltzmann machine [Tieleman, 2008], enables to speed up the first phase and reach the equilibrium state with higher precision.

However, this technique hurts generalization performance. Table 1 (bottom) shows the experimental results of Ernoult et al. [2019], which do not use this technique: during training, for each training example in the dataset, the state of the network is initialized to zero at the beginning of each free phase relaxation. The resulting test error rate is lower, though the number of iterations required in the free phase to converge to equilibrium is larger.

Model	cached	Test er.	Train er.	T	K	ϵ	β	Epochs	α_1	α_2	α_3	α_4
DHN-1h	Y	~ 2.5 %	0.00 %	20	4	0.5	1.0	25	0.1	0.05		
DHN-2h	Y	~ 2.3 %	0.00 %	100	6	0.5	1.0	60	0.4	0.1	0.01	
DHN-3h	Y	~ 2.7 %	0.00 %	500	8	0.5	1.0	150	0.128	0.032	0.008	0.002
DHN-1h	N	2.06 %	0.13 %	100	12	0.2	0.5	30	0.1	0.05		
DHN-2h	N	2.01 %	0.11 %	500	40	0.2	0.8	50	0.4	0.1	0.01	

Table 1. "DHN-#h" stands for Deep Hopfield Network with # hidden layers. 'cached' refers to whether or not the equilibrium states are cached and reused as a starting point at the next free phase relaxation. T is the number of iterations in the free phase. K is the number of iterations in the nudged phase. ϵ is the step size for the dynamics of the state variable s. β is the value of the nudging factor in the nudged phase. α_k is the learning rate for updating the parameters in layer k. **Top.** Experimental results of Scellier and Bengio [2017] with the caching trick. Test error rates and train error rates are reported on single trials. **Bottom.** Experimental results of Ernoult et al. [2019] without the caching trick. Test error rates and train error rates are averaged over five trials.

Since these early experiments, thanks to new insights, new ideas and more perseverance, new results have been obtained which improve in terms of simulation speed, test accuracy, and complexity of the task solved. We present these more recent experimental results in Chapter 5. In addition, we stress that the real potential of EqProp is more likely to shine on neuromorphic substrates (Chapter 4), rather than on digital computers.

3.4. Contrastive Hebbian Learning (CHL)

In the setting of continuous Hopfield networks studied in this Chapter, EqProp is similar to the generalized recirculation algorithm (GeneRec) [O'Reilly, 1996]. The main novelty of EqProp with respect to GeneRec is the formalism based on the concepts of nudging factor (β) and total energy function (F), which enables to formulate a general framework for training energy-based models (Chapter 2) and Lagrangian-based models (Chapter 6), applicable not just to the continuous Hopfield model, but also many more network models, including nonlinear resistive networks (Chapter 4) and convolutional networks (Chapter 5).

EqProp is also similar in spirit to the contrastive Hebbian learning algorithm (CHL), which we present in this section. The CHL algorithm was originally introduced in the case of the Boltzmann machine [Ackley et al., 1985] and then extended to the case of the continuous Hopfield network [Movellan, 1991, Baldi and Pineda, 1991]. We note that Boltzmann machines may be trained with EqProp, via the stochastic version presented in Section 6.2.

3.4.1. Contrastive Hebbian Learning in the Continuous Hopfield Model

Like EqProp, the CHL algorithm proceeds in two phases and uses a free phase. But unlike EqProp, it uses a *clamped phase* as a second phase for training, instead of a *nudged phase*. Bringing this modification to the EqProp training procedure described in section 3.2.2, we arrive at the following algorithm, proposed by Movellan [1991].

Free phase. As in EqProp, the first phase is a free phase (also called 'negative phase'): inputs x are clamped, and both the hidden and output neurons evolve freely, following the gradient of the energy function. The hidden and output neurons stabilize to an energy minimum called free state and denoted $(h_\star^-, o_\star^-)$. We write $s^- = (x, h_\star^-, o_\star^-)$. At the free state, every synapse undergoes an anti-Hebbian update. That is, for any synapse W_{ij} (connecting neuron i to neuron j), we perform the weight update $\Delta W_{ij} = -\eta\, \sigma\left(s_i^-\right) \sigma\left(s_j^-\right)$.

Clamped phase. The second phase is a 'clamped phase' (also called 'positive phase'): not only inputs are clamped, but also outputs are now clamped to their target value y. The hidden neurons evolve freely and stabilize to another energy minimum $h_\star^+$. We write $s^+ = (x, h_\star^+, y)$ and call this configuration the *clamped state*. At the clamped state, every synapse undergoes a Hebbian update. That is, for any synapse W_{ij} (connecting neuron i to neuron j), we perform the weight update $\Delta W_{ij} = +\eta\, \sigma\left(s_i^+\right) \sigma\left(s_j^+\right)$.

Global update. Putting the weight updates of the free phase and clamped phase together, we get the global update of the CHL algorithm:

$$\Delta W_{ij} = \eta \left(\sigma \left(s_i^+ \right) \sigma \left(s_j^+ \right) - \sigma \left(s_i^- \right) \sigma \left(s_j^- \right) \right). \tag{3.26}$$

3.4.2. An Intuition Behind Contrastive Hebbian Learning

Both CHL and EqProp have the desirable property that learning stops when the network correctly predicts the target. Specifically, in CHL, when the equilibrium state of the free phase (the free state) matches the equilibrium state of the clamped phase (the clamped state), the two terms of the weight update (Eq. 3.26) cancel out, thus yielding an effective weight update of zero. In other words, if the network already provides the correct output, then no learning occurs.

It is instructive to verify that EqProp preserves this property, even in its general formulation (Chapter 2). Suppose that the equilibrium state $(s_\star^0)$ corresponding to an input x provides the correct answer (y), i.e. suppose that $s_\star^0$ is a minimum of the function $s \mapsto C(s, y)$. This implies that $\frac{\partial C}{\partial s}(s_\star^0, y) = 0$. Using the fact that $\frac{\partial E}{\partial s}(\theta, x, s_\star^0) = 0$ by definition of $s_\star^0$, we get $\frac{\partial E}{\partial s}(\theta, x, s_\star^0) + \beta \, \frac{\partial C}{\partial s}(s_\star^0, y) = 0$ for any value of β. This implies that $s_\star^\beta = s_\star^0$ for any β, by definition of $s_\star^\beta$. As a consequence, the two terms in the learning rule of EqProp cancel out. We note that this property remains true in the case of the symmetric difference estimator (Eq. 2.10).

3.4.3. A Loss Function for Contrastive Hebbian Learning

The global learning rule of the CHL algorithm rewrites in terms of the energy function (the Hopfield energy) as

$$\Delta\theta = \eta \left(-\frac{\partial E}{\partial \theta} \left(\theta, x, h_\star^+, y \right) + \frac{\partial E}{\partial \theta} \left(\theta, x, h_\star^-, o_\star^- \right) \right). \tag{3.27}$$

Here $(x, h_\star^+, y)$ is the clamped state, and $(x, h_\star^-, o_\star^-)$ is the free state. In this form, the CHL update rule stipulates to decrease the energy value of the clamped state and to increase the energy value of the free state. Since low-energy configurations correspond to preferred states of the model under the gradient dynamics, the CHL update rule thus increases the likelihood that the model produces the correct output (y), and decreases the likelihood that it generates again the same output $(o_\star)$.

Proposition 3.4 (Movellan [1991])**.** *The CHL update rule (Eq. 3.27) is equal to*

$$\Delta\theta = -\eta \frac{\partial \mathcal{L}^{\mathrm{CHL}}}{\partial \theta}(\theta, x, y), \tag{3.28}$$

where $\mathcal{L}^{\mathrm{CHL}}$ *is the loss defined by*

$$\mathcal{L}^{\mathrm{CHL}}(\theta, x, y) = E\left(\theta, x, h_\star^+, y\right) - E\left(\theta, x, h_\star^-, o_\star^-\right). \tag{3.29}$$

The loss $\mathcal{L}^{\mathrm{CHL}}$ has the problem that the two phases of the CHL algorithm may stabilize in different modes of the energy function. Movellan [1991] points out that when this happens, the weight update is inconsistent and learning usually deteriorates. Similarly, Baldi and Pineda [1991] note abrupt discontinuities due to basin hopping phenomena.

EqProp solves this problem by optimizing the loss $\mathcal{L} = C(s_\star, y)$, whose gradient can be estimated using nudged states $(s_\star^\beta)$ that are infinitesimal continuous deformations of the free state $(s_\star)$, and are thus in the same 'mode' of the energy landscape.

Chapter 4

Training Nonlinear Resistive Networks with Equilibrium Propagation

In the previous chapter, we have discussed the bio-realism of EqProp in the setting of Hopfield networks. Learning in this context is achieved using solely leaky integrator neurons (in both phases of training) and a local (Hebbian) weight update. These bio-realistic features are of interest not only for neuroscience, but also for neuromorphic computing, towards the goal of building fully analog neural networks supporting on-chip learning. Recently, several works have proposed analog implementations of EqProp in the context of Hopfield networks [Zoppo et al., 2020, Foroushani et al., 2020, Ji and Gross, 2020] and spiking variants [O'Connor et al., 2019, Martin et al., 2020].

Here we investigate a different approach to implement EqProp on neuromorphic chips. We emphasize that EqProp is not limited to the Hopfield model and the gradient systems of Chapter 3, but more broadly applies to any system whose equilibrium state $s_\star$ is a solution of a variational equation $\frac{\partial E}{\partial s}(s_\star) = 0$, where $E(s)$ is a scalar function – what we have called an *energy-based model* (EBM) in Chapter 2. Importantly, many physical systems can be described by variational principles, as a reformulation of the physical laws characterizing their state. This suggests a path to build highly efficient energy-based models grounded in physics, with EqProp as a learning algorithm for training.

In this chapter, we exploit the fact that a broad class of analog neural networks called *nonlinear resistive networks* can be described by such a variational principle. Nonlinear resistive network are electrical circuits consisting of nodes interconnected by (linear or nonlinear) resistive elements. These circuits can serve as analog neural networks, in which the weights to be adjusted are implemented by the conductances of programmable resistive devices such as memristors [Chua, 1971], and the nonlinear transfer functions (or 'activation functions') are implemented by nonlinear components such as diodes. The 'energy function' in these nonlinear resistive networks is a quantity called the *co-content* [Millar, 1951] or *total pseudo-power* [Johnson, 2010] of the circuit, and its existence can be derived directly from Kirchhoff's laws.

Moreover, this energy function has the sum-separability property: the total pseudo-power of the circuit is the sum of the pseudo-powers of its individual elements. As a consequence, we can train these analog networks with EqProp, and the update rule for each conductance, which follows the gradient of the loss, is local. Specifically, we show mathematically that the gradient with respect to a conductance can be estimated using solely the voltage drop across the corresponding resistor. This theoretical result provides a principled method to train end-to-end analog neural networks by stochastic gradient descent, thus suggesting a path towards the development of ultra-fast, compact and low-power learning-capable neural networks.

The present chapter, which is essentially a rewriting of Kendall et al. [2020], is articulated as follows.

- In section 4.1, we briefly present a class of analog neural networks called *nonlinear resistive networks*, as well as the concept of *programmable resistors* that play the role of synapses.
- In section 4.2, we show that these nonlinear resistive networks are energy-based models: at inference, the configuration of node voltages chosen by the circuit corresponds to the minimum of a mathematical function (the *energy function*) called the *co-content* (or *total pseudo-power*) of the circuit, as a consequence of Kirchhoff's laws (Lemma 4.1). This suggests an implementation of energy-based neural networks grounded in electrical circuit theory, which also bridges the conceptual gap between energy functions (at a mathematical level[1]), and physical energies[2] (at a hardware level).
- In section 4.3, we show how these nonlinear resistive networks can be trained with EqProp, and we derive the formula for updating the conductances (the synaptic weights) in proportion to their loss gradients, using solely the voltage drops across the corresponding resistive devices (Theorem 4.2).
- In section 4.4, as a proof of concept of what is possible with this neuromorphic hardware methodology, we propose an analog network architecture inspired by the deep Hopfield network, which alternates linear and nonlinear processing stages (Fig. 7).
- In section 4.5, we present numerical simulations on the MNIST dataset, using a SPICE-based framework to simulate the circuit's dynamics.

By explicitly decoupling the training procedure (EqProp in Section 4.3) from the specific neural network architecture presented (Section 4.4), we stress that this optimization method is applicable to any resistive network architecture, not just the one of Section 4.4. This modular approach thus offers the possibility to explore the design space of analog network

[1]In an energy-based model, the *energy function* is a mathematical abstraction of the model, not a physical energy.
[2]Specifically the power dissipated in resistive devices.

architectures trainable with EqProp, in essentially the same way as deep learning researchers explore the design space of differentiable neural networks trainable with backpropagation.

4.1. Nonlinear Resistive Networks as Analog Neural Networks

Nonlinear resistive networks are electrical circuits consisting of arbitrary two-terminal resistive elements – see Muthuswamy and Banerjee [2018, Chapter 3] for an introduction. We can use such circuits to build neural networks. In the supervised learning scenario, we use a subset of the nodes of the circuit as input nodes, and another subset of the nodes as output nodes. We use voltage sources to impose the voltages at input nodes: after the circuit has settled to steady state, the voltages of output nodes indicate the 'prediction'. The circuit thus implements an input-to-output mapping function, with the node voltages representing the state of the network. This mapping function can be nonlinear if we include nonlinear resistive elements such as diodes in the circuit, and the conductance values of resistors can be thought of as parameterizing this mapping function.

A *programmable resistor* is a resistor whose conductance can be changed (or 'programmed'), and thus can play the role of a 'weight' to be adjusted. Programmable resistors can thus implement the synapses of a neural network. In the last decade, many technologies have emerged, and have been proposed and studied as programmable resistors. We refer to Burr et al. [2017] and Xia and Yang [2019] for reviews on existing technologies, their working mechanisms, and how they are used for neuromorphic computing. For convenience, in most of this chapter we will think of programmable resistors as ideally tunable, which is a convenient concept to formalize mathematically the goal of learning in nonlinear resistive networks. However, this is an ideal and unrealistic assumption: in practice, far from being ideally tunable, these programmable resistive devices currently present important challenges for the coming decade of research to solve. We refer to Chang et al. [2017] for an analysis of these challenges to be overcome. In this manuscript, we will not discuss how the programming of a conductance can be done and implemented in hardware.

We note that nonlinear resistive networks have been studied as neural network models since the 1980s [Hutchinson et al., 1988, Harris et al., 1989].

4.2. Nonlinear Resistive Networks are Energy-Based Models

In this section we show that, in a nonlinear resistive network, the steady state of the circuit imposed by Kirchhoff's laws is a stationary point of a function called the *co-content*, or *total pseudo-power* (Lemma 4.1). Thus, nonlinear resistive networks are energy-based

models whose energy function is the total pseudo-power. Furthermore, the total pseudo-power has the sum-separability property, being by definition the sum of the pseudo-powers of its individual components.

We first present in section 4.2.1 the case of linear resistance networks. Although this model is functionally not very useful (as a neural network model), studying it is helpful to gain understanding of the working mechanisms of analog neural networks: it helps understand the limits of linear resistances and the need to introduce nonlinear elements (section 4.2.2). In section 4.2.3, we derive the general result for nonlinear resistive networks.

4.2.1. Linear Resistance Networks

A *linear resistance network* is an electrical circuit whose nodes are linked pairwise by *linear resistors*, i.e. resistors that satisfy Ohm's law. We recall that, in a linear resistor, Ohm's law states that $I_{ij} = g_{ij}(V_i - V_j)$, where I_{ij} is the current through the resistor, g_{ij} is its conductance ($g_{ij} = \frac{1}{R_{ij}}$ where R_{ij} is the resistance), and V_i and V_j are its terminal voltages.

Consider the following question: we impose the voltages at a set of input nodes, and we want to know what are the voltages at other nodes of the circuit. We can answer this question by writing Ohm's law in every branch, Kirchhoff's current law at every node, and by solving the set of equations obtained for all node voltages and all branch currents. But there is a more elegant way to characterize the steady state of the circuit. Kirchhoff's current law gives $\sum_j I_{ij} = 0$ for every node i. Combined with Ohm's law, we get $\sum_j g_{ij}(V_i - V_j) = 0$. Now note that the left-hand side of this expression is equal to $\frac{1}{2}\frac{\partial \mathcal{P}}{\partial V_i}$, where $\mathcal{P}(V_1, V_2, \ldots, V_N)$ is the functional defined by

$$\mathcal{P}(V_1, V_2, \ldots, V_N) = \sum_{i<j} g_{ij} \left(V_j - V_i\right)^2 . \tag{4.1}$$

This means that, among all *conceivable* configurations of node voltages, the configuration that is physically realized is a stationary point of the functional $\mathcal{P}(V_1, V_2, \ldots, V_N)$. Therefore, linear resistance networks are energy-based models, with the configuration of node voltages $V = (V_1, V_2, \ldots, V_N)$ playing the role of state variable, and the functional $\mathcal{P}(V_1, V_2, \ldots, V_N)$ playing the role of energy function.

The functional $\mathcal{P}(V_1, V_2, \ldots, V_N)$ is called the *power functional*, because it represents the total power dissipated in the circuit, with $\frac{1}{2} g_{ij} \left(V_j - V_i\right)^2$ being the power dissipated in the resistor connecting node i to node j. Since $\mathcal{P}$ is convex, the steady state of the circuit is not just a stationary point of $\mathcal{P}$, but also the global minimum. This well-known result of circuit theory is called the *principle of minimum dissipated power*: if we impose the voltages at a set of input nodes, the circuit will choose the voltages at other nodes so as to minimize the total power dissipated in the resistors (Fig. 6).

However, linear resistance networks are not very useful as neural network models since they cannot implement nonlinear operations. Rewriting Kirchhoff's current law at node i, we get $V_i = \frac{\sum_j g_{ij} V_j}{\sum_j g_{ij}}$. This operation resembles the usual multiply-accumulate operation of artificial neurons in conventional deep learning, but with the notable difference that there is no nonlinear activation function. Another difference is the presence of the factor $G_i = \sum_j g_{ij}$ at the denominator, which replaces the usual weighted sum by a weighted mean: each floating node voltage V_i is a weighted mean of its neighbors.

From this analysis, it appears that nonlinear elements such as diodes are necessary to perform nonlinear operations. In the rest of this section, we generalize the result of this subsection to the setting of nonlinear resistive networks.

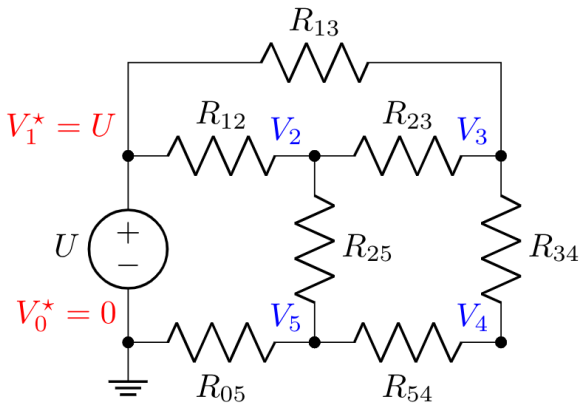

Fig. 6. Principle of minimum dissipated power. In a linear resistance network, if we impose the voltages at a set of input nodes ($V_0 = 0$ and $V_1 = U$ here), the voltages at other nodes (V_2, V_3, V_4 and V_5 here) is such that the total power dissipated in the resistors is minimized. A generalization of this result to nonlinear resistive networks exists (Lemma 4.1).

4.2.2. Two-Terminal Resistive Elements

In this subsection, we follow the method of Johnson [2010] to generalize the notion of 'power dissipated in a linear resistor' to arbitrary two-terminal resistive elements.

Current-voltage characteristic. Consider a two-terminal resistive element with terminals i and j, characterised by a well-defined and continuous current-voltage characteristic γ_{ij}. The function γ_{ij} takes as input the voltage drop $\Delta V_{ij} = V_i - V_j$ across the component and returns the current $I_{ij} = \gamma_{ij}(\Delta V_{ij})$ moving from node i to node j in response to ΔV_{ij}. Since the current flowing from i to j is the negative of the current flowing from j to i, we have by definition:

$$\forall i, j, \qquad \gamma_{ij}(\Delta V_{ij}) = -\gamma_{ji}(\Delta V_{ji}) \tag{4.2}$$

where $\Delta V_{ji} = -\Delta V_{ij}$.

For example, the current-voltage characteristic of a linear resistor of conductance g_{ij} linking node i to node j is, by Ohm's law, $I_{ij} = g_{ij}\Delta V_{ij}$. By definition of γ_{ij}, this implies that

$$\gamma_{ij}\left(\Delta V_{ij}\right) = g_{ij}\Delta V_{ij}. \tag{4.3}$$

Pseudo-power. For each two-terminal element with current-voltage characteristic $I_{ij} = \gamma_{ij}(\Delta V_{ij})$, we define $p_{ij}(\Delta V_{ij})$ as the primitive function of $\gamma_{ij}(\Delta V_{ij})$ that vanishes at 0, i.e.

$$p_{ij}(\Delta V_{ij}) = \int_0^{\Delta V_{ij}} \gamma_{ij}(v)dv. \tag{4.4}$$

The quantity $p_{ij}\left(\Delta V_{ij}\right)$ has the physical dimensions of power, being a product of a voltage and a current. We call $p_{ij}\left(\Delta V_{ij}\right)$ the *pseudo-power* along the branch from i to j, following the terminology of Johnson [2010]. Note that as a consequence of Eq. 4.2 we have

$$\forall i, j, \qquad p_{ij}(\Delta V_{ij}) = p_{ji}(\Delta V_{ji}), \tag{4.5}$$

i.e. the pseudo-power from i to j is equal to the pseudo-power from j to i. We call this property the *pseudo-power symmetry*.

For example, in the case of a linear resistor of conductance g_{ij} linking node i to node j, the pseudo-power corresponding to the current-voltage characteristic of Eq. 4.3 is:

$$p_{ij}(\Delta V_{ij}) = \frac{1}{2} g_{ij}\Delta V_{ij}^2. \tag{4.6}$$

In this case, the pseudo-power is half the physical power dissipated in the resistor.

4.2.3. Nonlinear Resistive Networks

A *nonlinear resistive network* is a circuit consisting of interconnected two-terminal resistive elements. We number the nodes of the circuit $i = 1, 2, \ldots, N$.

Configuration. We call a vector of voltage values $V = (V_1, V_2, \ldots, V_N)$ a *configuration*. Importantly, a configuration can be any vector of voltage values, even those that are not compatible with Kirchhoff's current law (KCL).

Total pseudo-power (also called co-content). Recall the definition of the pseudo-power of a two-terminal element (Eq. 4.4). We define the *total pseudo-power* of a configuration $V = (V_1, V_2, \ldots, V_N)$ as the sum of pseudo-powers along all branches:

$$\mathcal{P}(V_1, \cdots, V_N) = \sum_{i<j} p_{ij}(V_i - V_j). \tag{4.7}$$

We note that the pseudo-power symmetry (Eq. 4.5) guarantees that this definition does not depend on node ordering. In the case of a linear resistance network, the total pseudo-power of the circuit is half the power functional of Eq. 4.1.

We stress that $\mathcal{P}$ is a mathematical function defined on any configuration $V_1, V_2, \ldots, V_N$, even those that are not compatible with KCL.

Steady state. We denote $V_1^\star$, $V_2^\star$, ..., $V_N^\star$ the configuration of node voltages imposed by Kirchhoff's current law (KCL), and we call $V^\star = (V_1^\star, V_2^\star, \ldots, V_N^\star)$ the *steady state* of the circuit. Specifically, for every (internal or output) floating node i, KCL implies $\sum_{j=1}^N I_{ij} = 0$, which rewrites

$$\sum_{j=1}^{N} \gamma_{ij} \left(V_i^\star - V_j^\star \right) = 0. \tag{4.8}$$

The following result, known since Millar [1951], shows that the circuit is an energy-based model, whose energy function is the total pseudo-power.

Lemma 4.1. *The steady state of the circuit, denoted $(V_1^\star, V_2^\star, \ldots, V_N^\star)$, is a stationary point*[3] *of the total pseudo-power: for every floating node i, we have*

$$\frac{\partial \mathcal{P}}{\partial V_i} (V_1^\star, V_2^\star, \ldots, V_N^\star) = 0. \tag{4.9}$$

Proof of Lemma 4.1. We use the definition of the total pseudo-power (Eq. 4.7), the pseudo-power symmetry (Eq. 4.5), the definition of the pseudo-power (Eq. 4.4) and the fact that the steady state of the circuit satisfies Kirchhoff's current law (Eq. 4.8). For every floating node i we have:

$$\frac{\partial \mathcal{P}}{\partial V_i} (V_1^\star, V_2^\star, \ldots, V_N^\star) = \sum_j \frac{\partial p_{ij}}{\partial V_i} (V_i^\star - V_j^\star) = \sum_j \gamma_{ij} (V_i^\star - V_j^\star) = 0. \tag{4.10}$$

□

Equipped with this result, we can now derive a procedure to train nonlinear resistive networks with EqProp.

[3]With further assumptions on the current-voltage characteristics γ_{ij}, Christianson and Erickson [2007], as well as Johnson [2010], show that the function $\mathcal{P}$ is convex, so that the steady state is the global minimum of $\mathcal{P}$. However, in the context of EqProp, all one needs is the first order condition, i.e. the fact that the steady state is a stationary point of $\mathcal{P}$, not necessarily a minimum.

4.3. Training Nonlinear Resistive Networks with Equilibrium Propagation

4.3.1. Supervised Learning Setting

In the supervised learning setting, a subset of the nodes of the circuit are *input nodes*, at which input voltages (denoted X) are sourced. All other nodes – the *internal nodes* and *output nodes* – are left floating: after the voltages of input nodes have been set, the voltages of internal and output nodes settle to their steady state. The output nodes, denoted $\widehat{Y}$, represent the readout of the system, i.e. the model prediction. The architecture and the components of the circuit determine the $X \mapsto \widehat{Y}$ mapping function. Specifically, the conductances of the programmable resistors, denoted θ, parameterize this mapping function. That is, $\widehat{Y}$ can be written as a function of X and θ in the form $\widehat{Y}(\theta, X)$. Training such a circuit consists in adjusting the values of the conductances (θ) so that the voltages of output nodes ($\widehat{Y}$) approach the target voltages (Y). Formally, we cast the goal of training as an optimization problem in which the loss to be optimized (corresponding to an input-target pair (X, Y)) is of the form:

$$\mathcal{L}(\theta, X, Y) = C\left(\widehat{Y}(\theta, X), Y\right). \tag{4.11}$$

We have seen that nonlinear resistive networks are energy-based models (Lemma 4.1) and that the energy function (the total pseudo-power) is sum-separable, by definition (Eq.!4.7). This enables us to use EqProp in such analog neural networks to compute the gradient of the loss. Theorem 4.2 below provides a formula for computing the loss gradient with respect to a conductance using solely the voltage drop across the corresponding resistor.

4.3.2. Training Procedure

Given an input X and associated target Y, EqProp proceeds in the following two phases.

Free phase. At inference, input voltages are sourced at input nodes (X), while all other nodes of the circuit (the internal nodes and output nodes) are left floating. All internal and output node voltages are stored[4]. In particular, the voltages of output nodes ($\widehat{Y}$) corresponding to prediction are compared with the target (Y) to compute the loss $\mathcal{L} = C(\widehat{Y}, Y)$.

[4]On practical neuromorphic hardware, this can be achieved using a capacitor or sample-and-hold amplifier (SHA) circuit, for instance. We note that we only need one SHA per node (neuron), not per synapse. We will not discuss these aspects of implementation here.

Nudged phase. For each output node $\widehat{Y}_k$, a current $I_k = -\beta \frac{\partial C}{\partial \widehat{Y}_k}$ is sourced at $\widehat{Y}_k$, where β is a positive or negative scaling factor (the *nudging factor*). All internal node voltages and output node voltages are measured anew.

Theorem 4.2 (Kendall et al. [2020])**.** *Consider a two-terminal component whose terminals are i and j. Denote ΔV_{ij}^0 the voltage drop across this two-terminal component in the free phase (when no current is sourced at output nodes), and ΔV_{ij}^β the voltage drop in the nudged phase (when a current $I_k = -\beta \frac{\partial C}{\partial \widehat{Y}_k}$ is sourced at each output node $\widehat{Y}_k$). Let w_{ij} denote an adjustable parameter of this component, and p_{ij} its pseudo-power (which depends on w_{ij}). Then, the gradient of the loss $\mathcal{L} = C\left(\widehat{Y}, Y\right)$ with respect to w_{ij} can be estimated as*

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \lim_{\beta \to 0} \frac{1}{\beta} \left(\frac{\partial p_{ij}\left(\Delta V_{ij}^\beta\right)}{\partial w_{ij}} - \frac{\partial p_{ij}\left(\Delta V_{ij}^0\right)}{\partial w_{ij}} \right). \tag{4.12}$$

In particular, if the component is a linear resistor of conductance g_{ij}, then the loss gradient with respect to g_{ij} can be estimated as

$$\frac{\partial \mathcal{L}}{\partial g_{ij}} = \lim_{\beta \to 0} \frac{1}{2\beta} \left(\left(\Delta V_{ij}^\beta\right)^2 - \left(\Delta V_{ij}^0\right)^2 \right). \tag{4.13}$$

Proof. For simplicity, we have stated Theorem 4.2 in the case where the cost function $C(\widehat{Y}, Y)$ depends only on output node voltages ($\widehat{Y}$). But this result can be directly generalized to the case of a cost function $C(V, Y)$ that depends on any node voltages (V), not just output node voltages. In this case, in the nudged phase of EqProp, currents $I_k = -\beta \frac{\partial C}{\partial V_k}$ must be sourced at every node V_k (not just at output nodes).

Let θ denote the vector of adjustable parameters (e.g. the conductances), X the voltages of input nodes, and V the voltages of floating nodes (which includes the internal nodes and output nodes). Further let $\mathcal{P}(\theta, X, V)$ denote the total pseudo-power of the circuit in the free phase. By Lemma 4.1, the steady state $V_\star$ of the free phase is such that $\frac{\partial \mathcal{P}}{\partial V}(\theta, X, V_\star) = 0$. In the nudged phase, when a current $I_k = -\beta \, \frac{\partial C}{\partial V_k}(V_\star, Y)$ is sourced at every floating node V_k, Kirchhoff's current law at the steady state $V_\star^\beta$ implies that $\frac{\partial \mathcal{P}}{\partial V}(\theta, X, V_\star^\beta) + \beta \, \frac{\partial C}{\partial V}(V_\star, Y) = 0$. Furthermore, the total pseudo-power (Eq. 4.7) has the sum-separability property: an adjustable parameter w_{ij} of a component whose terminals are i and j contributes to $\mathcal{P}(\theta, X, V)$ only through the pseudo-power $p_{ij}(V_i - V_j)$ of that component. Therefore, Eq. 4.12 follows from the main Theorem 2.1.

In the case of a linear resistor, the adjustable parameter is $w_{ij} = g_{ij}$ and the pseudo-power is given by Eq. 4.6. Thus, Eq. 4.13 follows from Eq. 4.12 and the fact that $\frac{\partial p_{ij}(\Delta V_{ij})}{\partial g_{ij}} = \frac{1}{2}\left(\Delta V_{ij}\right)^2$. □

As explained in the general setting (Section 2.4), it is possible to reduce the bias and the variance of the gradient estimator by performing two nudged phases: one with a positive nudging ($+\beta$) and one with a negative nudging ($-\beta$).

Although the framework we have presented here is deterministic, we note that analog circuits in practice are affected by noise. In section 6.2 we present a stochastic version of EqProp which can model such forms of noise and incorporate effects of thermodynamics.

4.3.3. On the Loss Gradient Estimates

Computing the sign of the gradients. Theorem 4.2 provides a formula for computing the gradient of a given device, assuming that the pseudo-power gradient ($\frac{\partial p_{ij}}{\partial w_{ij}}$) of this device is known, and that its terminal voltages can be measured, stored[5] and retrieved with arbitrary precision. In practice however, these conditions are too stringent.

A piece of good news is that there is empirical evidence that training neural networks by stochastic gradient descent (SGD) works well, even if for each weight, only the sign of the weight gradient is known. Variants of SGD which use the sign of the gradient rather than its exact value work well in practice. At each step of this training procedure, the weight update for θ_k takes the form $\Delta\theta_k = -\eta \text{ sign}\left(\frac{\partial \mathcal{L}}{\partial \theta_k}\right)$. The effectiveness of this optmization method has been shown empirically in the context of differentiable neural networks trained with backpropagation [Bernstein et al., 2018].

In the context of nonlinear resistive networks trained with EqProp, if we aim to get the correct sign (rather than its exact value) of the gradient for a given resistor, precise knowledge of the voltage values at the terminals is not necessary. As a corollary of Theorem 4.2, the sign of the gradient can be obtained by comparing $\left|\Delta V_{ij}^0\right|$ and $\left|\Delta V_{ij}^\beta\right|$, i.e. the absolute values of the voltages across the resistor in the free phase and the nudged phase[6]. This means that, if we aim to compute the sign of the gradient, we only need to perform a 'compare' operation reliably.

Robustness to characteristics variability. A large body of work aims at implementing the backpropagation algorithm in analog [Burr et al., 2017, Xia and Yang, 2019]. However, the weight gradients computed by backpropagation are sensitive to characteristics variability of analog devices. This is because the mathematical derivation of the backpropagation algorithm relies on a *global coordination of elementary operations*: if any of the elementary

[5]The node voltages must be measured and stored at the end of the first phase, since they are no longer physically available after the second phase, at the moment of the weight update. We can achieve this with a sample and hold amplifier circuit.

[6]Equivalently, we can compare $\left|I_{ij}^0\right|$ and $\left|I_{ij}^\beta\right|$, i.e. the currents through the resistor in the free phase and the nudged phase.

operations of the algorithm is inaccurate, then the gradients computed are inaccurate (i.e. biased).

Although there is no experimental evidence for this fact yet, there are reasons to believe that the gradient estimates of EqProp are more robust to device mismatches than the gradients of Backprop. The reason is that, in EqProp, the same circuit is used in both phases of training. Intuitively, any device mismatch will affect the steady states of both phases (free phase and nudged phase), and since the gradient estimate depends on the difference between the measurements of the two phases, the effects of the mismatch will cancel out. More precisely, Theorem 4.2 tells us that the quality of the gradient estimate for a given device does not depend on the characteristics of other devices in the circuit.

We note that this argument only holds for the computation/estimation of the weight gradients. In EqProp like in Backprop, the challenge of weight update asymmetry of programmable resistors remains.

4.4. Example of a Deep Analog Neural Network Architecture

The theory of Section 4.3 applies to any nonlinear resistive network. In this section, as an example of what is possible with this general method, we present the neural network architecture proposed by Kendall et al. [2020], inspired by the deep Hopfield network model (Figure 7). It is composed of multiple layers, alternating linear and non-linear processing stages. The linear transformations are performed by crossbar arrays of programmable resistors, that play the role of weight matrices that parameterize the transformations. The nonlinear transfer function is implemented using a pair of diodes, followed by a linear amplifier. These crossbar arrays of programmable resistors and these nonlinear transfer functions are alternated to form a deep network.

4.4.1. Antiparallel Diodes

We propose to implement the neuron nonlinearities (or 'activation functions') as shunt conductances. To do this, we place two diodes antiparallel between the neuron's node and ground. Each diode is placed in series with a voltage source, used to shift the bounds of the activation function. The diodes ensure that the neuron's voltage remains bounded even as its input current grows large, because for any additional input current, one of the diodes turns on and sinks the extra current to ground.

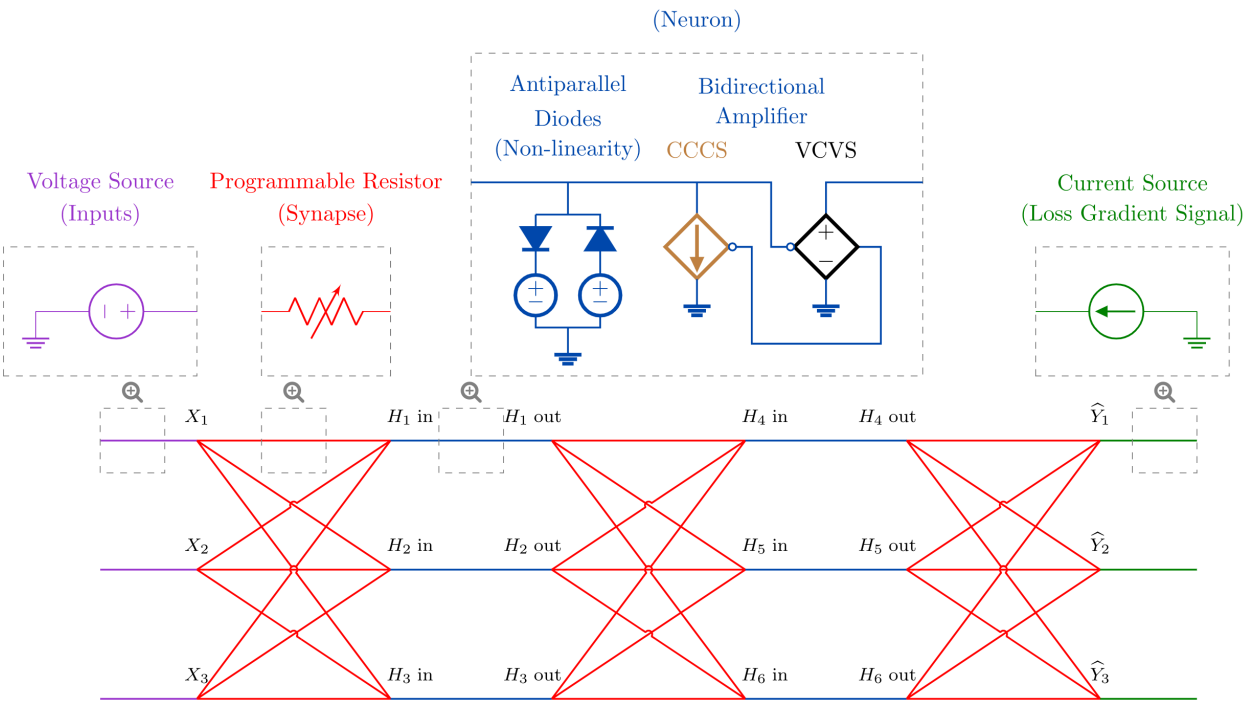

Fig. 7. Deep analog neural network with three input nodes (X_1, X_2 and X_3), two layers of three hidden neurons each (H_1, H_2, H_3, and H_4, H_5, H_6) and three output nodes ($\widehat{Y}_1$, $\widehat{Y}_2$ and $\widehat{Y}_3$). Blue branches and red branches represent neurons and synapses, respectively. Each synapse is a programmable resistor, whose conductance represents a parameter to be adjusted. Each neuron is formed of a nonlinear transfer function and a bidirectional amplifier. The nonlinear transfer function is implemented by a pair of antiparallel diodes (in series with voltage sources), which forms a sigmoidal function in its voltage response (section 4.4.1). The bidirectional amplifier consists of a current-controlled current source (CCCS, shown in brown) and a voltage-controlled voltage source (VCVS, shown in black), allowing signals to propagate in both directions without a decay in amplitude (section 4.4.2). Output nodes are linked to current sources (shown in green) which serve to inject loss gradient signals during training (section 4.4.4). **Equilibrium Propagation** (EqProp) allows to compute the gradient of a loss $\mathcal{L} = C(\widehat{Y}, Y)$, where Y is the desired target (section 4.3). In the free phase (inference), input voltages are sourced at input nodes and the current sources are set to zero current. In the nudged phase (training), for each output node $\widehat{Y}_k$ the corresponding current source is set to $I_k = -\beta \frac{\partial C}{\partial \widehat{Y}_k}$, where β is a scaling factor called nudging factor (a hyperparameter). The update rule to adjust the conductances of programmable resistors is local (Theorem 4.2).

4.4.2. Bidirectional Amplifiers

In a circuit composed only of resistors and diodes, voltages decay through the resistive layers. This *vanishing signals effect* can be explained by the fact that currents always flow from high electric potential to low electric potential. Thus, extremal voltage values are necessarily reached at input nodes, whose voltages are set.

To counter signal decay, one option is to use voltage-controlled voltage sources (VCVS) to amplify the voltages of hidden neurons in the forward direction. Current-controlled current sources (CCCS) can also be used to amplify currents in the backward direction, to better propagate error signals in the nudged phase. We call such a combination of a forward-directed VCVS and a backward-directed CCCS a 'bidirectional amplifier'.

4.4.3. Positive Weights

Unlike conventional neural networks trained in software whose weights are free to take either positive or negative values, one constraint of analog neural networks is that the conductances of programmable resistors (which represent the weights) are positive. Several approaches are proposed in the literature to overcome this structural constraint. One approach consists in decomposing each weight as the difference of two (positive) conductances [Wang et al., 2019]. Another approach is to shift the mean of the weight matrix by a constant factor [Hu et al., 2016].

A third approach proposed here consists in doubling the number of input nodes, and to duplicate input values by inverting one set. We also double the number of output nodes so that, in a classification task with K classes, the network has two output nodes for each class k, denoted $\widehat{Y}_k^+$ and $\widehat{Y}_k^-$, with $\widehat{Y}_k^+ - \widehat{Y}_k^-$ representing a score assigned to class k. The prediction of the model is then

$$\widehat{Y}_{\text{pred}} = \underset{0\leq k\leq K}{\arg\max} \left(\widehat{Y}_k^+ - \widehat{Y}_k^-\right). \tag{4.14}$$

We optimize the loss associated to the squared error cost function, i.e. $\mathcal{L} = C(V_\star, Y)$, where the target vector $Y = (Y_1, Y_2, \ldots, Y_K)$ is the one-hot code of the class label, and

$$C(\widehat{Y}, Y) = \frac{1}{2}\sum_{k=1}^{K}\left(\widehat{Y}_k^+ - \widehat{Y}_k^- - Y_k\right)^2. \tag{4.15}$$

4.4.4. Current Sources

The nudged phase requires to inject currents I_k^+ and I_k^- at output nodes $\widehat{Y}_k^+$ and $\widehat{Y}_k^-$. These currents must be proportional to the gradients of output node voltages $\widehat{Y}_k^+$ and $\widehat{Y}_k^-$, i.e.

$$I_k^+ = -\beta\frac{\partial C}{\partial \widehat{Y}_k^+} = \beta\left(Y_k + \widehat{Y}_k^- - \widehat{Y}_k^+\right), \qquad I_k^- = -\beta\frac{\partial C}{\partial \widehat{Y}_k^-} = \beta\left(\widehat{Y}_k^+ - \widehat{Y}_k^- - Y_k\right), \tag{4.16}$$

where the nudging factor β has the physical dimensions of a conductance. We can inject these currents in the nudged phase using current sources. In the free phase, these current sources are set to zero current and do not influence the voltages of output nodes, acting like open circuits.

4.5. Numerical Simulations on MNIST

Kendall et al. [2020] present simulations on the MNIST digits classification task, performed using the high-performance SPICE-class parallel circuit simulator *Spectre* [Cadence Design Systems, Inc., 2020]. SPICE (simulation program with integrated circuit emphasis) is a framework for realistic simulations of circuit dynamics [Vogt et al., 2020]. Specifically, SPICE is used in the simulations to perform the free phase and the nudged phase of the EqProp training process. The other operations are performed in Python: this includes weight initialization (before training starts), calculating loss and gradient currents (between the free phase and the nudged phase), weight gradient calculation (at the end of the nudged phase) and performing the weight updates (resistances are updated in software). We refer to Kendall et al. [2020] for full details of the implementation and simulation results.

Simulations are performed on a small network with a single hidden layer of 100 neurons. Training is stopped after 10 epochs, when the SPICE network achieves a test error rate of 3.43%. For comparison, LeCun et al. [1998] report results with different kinds of linear classifiers and logistic regression models (corresponding to different pre-processing methods), all performing $> 7\%$ test error, which is significantly worse than the SPICE network. This demonstrates that the SPICE network benefits from the non-linearities offered by the diodes.

Chapter 5

Training Discrete-Time Neural Network Models with Equilibrium Propagation

In the previous chapters, we have presented EqProp in its general formulation (Chapter 2), and we have applied it to gradient systems (such as the continuous Hopfield model, Chapter 3), and to physical systems that can be described by a variational principle (such as nonlinear resistive networks, Chapter 4). Although EqProp is a potentially promising tool for training neuromorphic hardware, developing such hardware is still in the future. Whereas in the previous two chapters we have mostly focused on neuroscience and neuromorphic considerations, it is also essential to demonstrate the potential of EqProp to solve practical tasks. In this chapter, we focus on the scalability of EqProp in software, to demonstrate its usefulness as a learning strategy.

When simulated on digital computers, the models presented in the previous chapters are very slow and require long inference times to converge to equilibrium. More importantly, these models have thus far not been proved to scale to tasks harder than MNIST. In this chapter, we present a class of models trainable with EqProp, specifically aimed at accelerating simulations in software, and at scaling EqProp training to larger models and more challenging tasks. As a consequence of this change of perspective, some of the techniques introduced in this chapter can be viewed as a step backward from biorealism and neuromorphic considerations (e.g. the use of shared weights in the convolutional network models). However, the introduction of such techniques allows us to broaden the scope of EqProp and to benchmark it against more advanced models of deep learning.

The present chapter, which is essentially a compilation and a rewriting of Ernoult et al. [2019] and Laborieux et al. [2021], is organized as follows.

- In Section 5.1, we present a discrete-time formulation of EqProp, which allows training neural network models closer to those used in conventional deep learning.

- In Section 5.2, we present discrete-time neural network models trainable with EqProp, including a fully-connected model (close in spirit to the Hopfield model) and a convolutional one. In contrast with previous chapters where we have only considered the squared error as a cost function, we present here a method to optimize the cross-entropy loss commonly used for classification tasks.
- In Section 5.3, we present the experimental results of Ernoult et al. [2019] and Laborieux et al. [2021]. Compared to the experiments of Chapter 3, discrete-time models allow one to reduce the computational cost of inference, and enable to scale EqProp to deeper architectures and more challenging tasks. In particular, a ConvNet model trained with EqProp achieves 11.68% test error rate on CIFAR-10. Furthermore, these experiments highlight the importance of reducing the bias and variance of the loss gradient estimators on complex tasks. We also discuss some challenges to overcome in order to unlock the scaling of EqProp to larger models and harder tasks, as well as some promising avenues towards this goal.
- In Section 5.4, we present a theoretical result linking the transient states in the second phase of EqProp to the partial derivatives of the loss to optimize (Theorem 5.1 and Fig. 8). This property, which we call the *gradient descending dynamics* (GDD) property, is useful in practice as it prescribes a criterion to decide when the dynamics of the first phase of training has converged to equilibrium.

5.1. Discrete-Time Dynamical Systems with Static Input

In this section, we apply EqProp to a class of discrete-time dynamical systems, as proposed by Ernoult et al. [2019].

5.1.1. Primitive Function

Consider an energy function of the form

$$E(\theta, x, s) = \frac{1}{2}\|s\|^2 - \Phi(\theta, x, s), \tag{5.1}$$

where Φ is a scalar function that we will choose later. With this choice of energy function, the equilibrium condition $\frac{\partial E}{\partial s}(\theta, x, s_\star) = 0$ of Eq. 2.3 rewrites as a fixed point condition:

$$s_\star = \frac{\partial \Phi}{\partial s}(\theta, x, s_\star). \tag{5.2}$$

Assuming that the function $s \mapsto \frac{\partial \Phi}{\partial s}(\theta, x, s)$ is contracting, by the contraction mapping theorem, the sequence of states s_1, s_2, s_3, ... defined by

$$s_{t+1} = \frac{\partial \Phi}{\partial s}(\theta, x, s_t) \tag{5.3}$$

converges to $s_\star$. This dynamical system can be viewed as a recurrent neural network (RNN) with static input x (meaning that the same input x is fed to the RNN at each time step) and transition function $F = \frac{\partial \Phi}{\partial s}$. Because Φ is a primitive function of the transition function F, we call Φ the *primitive function* of the system. In light of Eq. 5.2, in this chapter we will call $s_\star$ a *fixed point* (rather than an equilibrium state).

The question of necessary and sufficient conditions on Φ for the dynamics of Eq. 5.3 to converge to a fixed point is out of the scope of the present manuscript. We refer to Scarselli et al. [2009] where conditions on the transition function are discussed.

5.1.2. Training Discrete-Time Dynamical Systems with Equilibrium Propagation

Recall that we want to optimize a loss of the form

$$\mathcal{L} = C(s_\star, y), \tag{5.4}$$

where $C(s, y)$ is a scalar function called *cost function*, defined for any state s. In the discrete-time setting, EqProp takes the following form.

Free Phase. In the free phase, the dynamics of Eq. 5.3 is run for T time steps, until the sequence of states $s_1, s_2, s_3, \ldots, s_T$ has converged. At the end of the free phase, the network is at the *free fixed point* $s_\star$ characterized by Eq. 5.2, i.e. $s_T = s_\star$.

Nudged Phase. In the nudged phase, starting from the free fixed point $s_\star$, an additional term $-\beta \frac{\partial C}{\partial s}$ is introduced in the dynamics of the neurons, where β is a positive or negative scalar, called *nudging factor*. This term acts as an external force nudging the system dynamics towards decreasing the cost function C. Denoting $s_0^\beta, s_1^\beta, s_2^\beta, \ldots$ the sequence of states in the second phase (which depends on the value of β), we have

$$s_0^\beta = s_\star \qquad \text{and} \qquad \forall t \geq 0, \quad s_{t+1}^\beta = \frac{\partial \Phi}{\partial s}\left(\theta, x, s_t^\beta\right) - \beta \frac{\partial C}{\partial s}\left(s_t^\beta, y\right). \tag{5.5}$$

The network eventually settles to a new fixed point $s_\star^\beta$, called *nudged fixed point*.

Update Rule. In this context, the formula for estimating the loss gradients using the two fixed points $s_\star$ and $s_\star^\beta$ takes the form

$$\lim_{\beta \to 0} \frac{1}{\beta} \left(\frac{\partial \Phi}{\partial \theta} \left(\theta, x, s_\star^\beta \right) - \frac{\partial \Phi}{\partial \theta} \left(\theta, x, s_\star \right) \right) = - \frac{\partial \mathcal{L}}{\partial \theta}. \tag{5.6}$$

Furthermore, if the primitive function Φ has the sum-separability property, i.e. if it is of the form $\Phi(\theta, x, s) = \Phi_0(x, s) + \sum_{k=1}^N \Phi_k(\theta_k, x, s)$ where $\theta = (\theta_1, \theta_2, \dots, \theta_N)$, then

$$\lim_{\beta \to 0} \frac{1}{\beta} \left(\frac{\partial \Phi_k}{\partial \theta_k} \left(\theta_k, x, s_\star^\beta \right) - \frac{\partial \Phi_k}{\partial \theta_k} \left(\theta_k, x, s_\star \right) \right) = - \frac{\partial \mathcal{L}}{\partial \theta_k}. \tag{5.7}$$

Eq. 5.6 follows directly from Theorem 2.1 and the definition of Φ in terms of E (Eq. 5.1). Eq. 5.7 follows from Eq. 5.6 and the definition of sum-separability.

In the discrete-time setting, as in the other settings, we can reduce the bias and the variance of the gradient estimate by using a symmetrized gradient estimator (see Eq. 2.10). This requires two nudged phases: one with a positive nudging ($+\beta$) and one with a negative nudging ($-\beta$).

5.1.3. Recovering Gradient Systems

We note that if we choose a primitive function Φ of the form $\Phi(\theta, x, s) = \frac{1}{2}\|s\|^2 - \epsilon \, \widetilde{E}(\theta, x, s)$, where ϵ is a positive hyperparameter and $\widetilde{E}$ is a scalar function, then the dynamics of Eq. 5.3 rewrites $s_{t+1} = s_t - \epsilon \frac{\partial \widetilde{E}}{\partial s} (\theta, x, s_t)$. This is the Euler scheme with discretization step ϵ of the gradient dynamics $\frac{d}{dt} s_t = - \frac{\partial \widetilde{E}}{\partial s} (\theta, x, s_t)$, which was used in the simulations of Chapter 3. In this sense, the setting of gradient systems of Chapter 3 can be seen as a particular case of the discrete-time formulation presented in this chapter.

5.2. RNN Models with Static Input

The algorithm presented in the previous section is generic and holds for arbitrary primitive function Φ and cost function C. In this section, we present two models corresponding to different choices of primitive function. The first model is a vanilla RNN with static input and symmetric weights (a variant of the Hopfield model). The second model is a convolutional RNN model. We also propose different choices of cost function: whereas in Chapter 3 and Chapter 4 we have only considered the squared error between outputs and targets, here we also present an implementation of the cross-entropy cost function.

5.2.1. Fully Connected Layers

To implement a fully connected layer, we consider the following primitive function:

$$\Phi_k^{\text{fc}}(w_k, h^{k-1}, h^k) = (h^k)^\top \cdot w_k \cdot h^{k-1}. \tag{5.8}$$

In this expression, h^{k-1} and h^k are two consecutive layers of neurons, and w_k is a weight matrix of size $\dim(h^k) \times \dim(h^{k-1})$ connecting h^{k-1} to h^k. We note that Φ_k^{fc} is closely related to the Hopfield energy of Eq. 3.16.

By stacking several of these fully connected layers, we can form a network of multiple layers of the kind considered in the previous chapters (e.g. as depicted in Fig 5). The corresponding primitive function is obtained by summing together the primitive functions of individual pairs of layers. For example, consider $\Phi = \cdots + \Phi_k^{\text{fc}} + \Phi_{k+1}^{\text{fc}} + \cdots$, i.e.

$$\Phi = \cdots + (h^k)^\top \cdot w_k \cdot h^{k-1} + (h^{k+1})^\top \cdot w_{n+1} \cdot h^k + \cdots . \tag{5.9}$$

For this choice of primitive function, we have $\frac{\partial \Phi}{\partial h^k} = w_k \cdot h^{k-1} + w_{n+1}^\top \cdot h^{k+1}$. In practice, we find that it is necessary that the values of the state variable be bounded. For this reason, we apply an activation function σ and arrive at the following dynamics in the free phase, which is a discrete-time variant of the dynamics of the Hopfield network studied in Chapter 3:

$$h_{t+1}^k = \sigma\left(w_k \cdot h_t^{k-1} + w_{n+1}^\top \cdot h_t^{k+1}\right). \tag{5.10}$$

5.2.2. Convolutional Layers

Ernoult et al. [2019] propose to implement a convolutional layer with the following primitive function:

$$\Phi_k^{\text{conv}}(w_k, h^{k-1}, h^k) = h^k \bullet \mathcal{P}\left(w_k \star h^{k-1}\right). \tag{5.11}$$

In this expression, w_k is the kernel (convolutional weights) for that layer, $\star$ is the convolution operator, $\mathcal{P}$ is a pooling operation, and $\bullet$ is the canonical scalar product for pairs of tensors with same dimension. In particular, in this expression, h^k and $\mathcal{P}\left(w_k \star h^{k-1}\right)$ are tensors with same size. This implementation is similar to the one proposed by Lee et al. [2009] in the context of restricted Boltzmann machines.

By stacking several of these convolutional layers we can form a deep ConvNet (specifically a recurrent convolutional network with static input and symmetric weights). Consider a primitive function of the form $\Phi = \cdots + \Phi_k^{\text{conv}} + \Phi_{k+1}^{\text{conv}} + \cdots$, i.e.

$$\Phi = \cdots + h^k \bullet \mathcal{P}\left(w_k \star h^{k-1}\right) + h^{k+1} \bullet \mathcal{P}\left(w_{n+1} \star h^k\right) + \cdots \tag{5.12}$$

We have $\frac{\partial \Phi}{\partial h^k} = \mathcal{P}\left(w_k \star h^{k-1}\right) + \tilde{w}_{k+1} \star \mathcal{P}^{-1}\left(h^{k+1}\right)$, where $\mathcal{P}^{-1}$ is an 'inverse pooling' operation, and $\tilde{w}_k$ is the flipped kernel, which forms the transpose convolution. We refer to Ernoult et al. [2019] where these operations are defined in details. After restricting the space of the state variables by using the hardsigmoid activation function σ to clip the states, we obtain the following dynamics for layer h^k:

$$h_{t+1}^k = \sigma\left(\mathcal{P}\left(w_k \star h_t^{k-1}\right) + \tilde{w}_{n+1} \star \mathcal{P}^{-1}\left(h_t^{k+1}\right)\right). \tag{5.13}$$

We can also combine convolutional layers, followed by fully connected layers, to form a more practical deep ConvNet. Denoting N^{conv} and N^{fc} the number of convolutional layers and fully connected layers, the total number of layers is $N^{\text{tot}} = N^{\text{conv}} + N^{\text{fc}}$ and the primitive function is

$$\Phi(\theta, x, s) = \sum_{k=1}^{N^{\text{conv}}} \Phi_k^{\text{conv}}(w_k, h^{k-1}, h^k) + \sum_{k=N^{\text{conv}}+1}^{N^{\text{tot}}} \Phi_k^{\text{fc}}(w_k, h^{k-1}, h^k), \tag{5.14}$$

where the set of parameters is $\theta = \{w_k\}_{1 \leq k \leq N^{\text{tot}}}$, the input is $x = h^0$, and the state variable is $s = \{h^k\}_{1 \leq k \leq N^{\text{tot}}}$.

5.2.3. Squared Error

We have already studied in Chapters 3 and 4 the case where we optimize the loss associated to the squared error cost function. In this setting, the state variable of the network is of the form $s = (h, o)$, where h represents the *hidden neurons* and o the *output neurons*, and the cost function is

$$C(o, y) = \frac{1}{2} \|o - y\|^2 . \tag{5.15}$$

The nudged phase dynamics of the hidden neurons and output neurons read, in this context:

$$h_{t+1}^\beta = \frac{\partial \Phi}{\partial h}(\theta, x, h_t^\beta, o_t^\beta), \qquad o_{t+1}^\beta = \frac{\partial \Phi}{\partial o}(\theta, x, h_t^\beta, o_t^\beta) + \beta \, (y - o_t^\beta). \tag{5.16}$$

5.2.4. Cross-Entropy

Laborieux et al. [2021] present a method to implement the output layer of the neural network as a softmax output, which can be used in conjunction with the cross-entropy loss. In this setting, the state of output neurons (o) are not a part of the state variable (s), but are instead viewed as a readout, which is a function of s and of a weight matrix w_{out} of size $\dim(y) \times \dim(s)$. Specifically, the state of output neurons at time step t is defined by the formula:

$$o_t = \text{softmax}(w_{\text{out}} \cdot s_t). \tag{5.17}$$

Denoting $M = \dim(y)$ the number of categories in the classification task of interest, the cross-entropy cost function associated with the softmax output is then:

$$C(s, y, w_{\text{out}}) = -\sum_{i=1}^{M} y_i \log(\text{softmax}_i(w_{\text{out}} \cdot s)). \tag{5.18}$$

Using the fact that $\frac{\partial C}{\partial s}(s, y, w_{\text{out}}) = w_{\text{out}}^\top \cdot (\text{softmax}(w_{\text{out}} \cdot s) - y)$, the nudged phase dynamics corresponding to the cross-entropy cost function read

$$s_{t+1}^\beta = \frac{\partial \Phi}{\partial s}(\theta, x, s_t^\beta) + \beta \, w_{\text{out}}^\top \cdot \left(y - o_t^\beta\right), \tag{5.19}$$

where $o_t^\beta = \text{softmax}(w_{\text{out}} \cdot s_t^\beta)$. Note that in this context the loss $\mathcal{L} = C(s_\star, y, w_{\text{out}})$ also depends on the parameter w_{out}. The loss gradient with respect to w_{out} is given by

$$\frac{\partial \mathcal{L}}{\partial w_{\text{out}}} = s_\star^\top \cdot (y - o_\star)\,, \tag{5.20}$$

where $o_\star = \text{softmax}(w_{\text{out}} \cdot s_\star)$.

In practice, the state variable is of the form $s = (h^1, h^2, \ldots, h^N)$, where $h^1, h^2, \ldots, h^N$ are the hidden layers of the network, and w_{out} connects only the last hidden layer h^N (not all hidden layers) to the output layer o. The weight matrix w_{out} has size $\dim(y) \times \dim(h^N)$ in this case.

5.3. Experiments on MNIST and CIFAR-10

In this section, we present the experimental results of Ernoult et al. [2019] and Laborieux et al. [2021], on the MNIST (Table 2) and the CIFAR-10 (Table 3) classification tasks, respectively. The CIFAR-10 dataset [Krizhevsky et al., 2009] consists of $60,000$ colour images of 32×32 pixels. These images are split in 10 classes (each corresponding to an object or animal), with $6,000$ images per class. The training set consists of $50,000$ images and the test set of $10,000$ images.

Experiments are performed on different network architectures (composed of multiple fully-connected and/or convolutional layers), using different cost functions (either the squared error or the cross-entropy loss) and different loss gradient estimators. Using the notations of this chapter, the *one-sided* gradient estimator ($\widehat{\nabla}_\theta(\beta)$) and the *symmetric* gradient estimator ($\widehat{\nabla}_\theta^{\text{sym}}(\beta)$) presented in Chapter 2 take the form

$$\widehat{\nabla}_\theta(\beta) = \frac{1}{\beta}\left(\frac{\partial \Phi}{\partial \theta}(\theta, x, s_\star^\beta) - \frac{\partial \Phi}{\partial \theta}(\theta, x, s_\star)\right), \tag{5.21}$$

$$\widehat{\nabla}_\theta^{\text{sym}}(\beta) = \frac{1}{2\beta}\left(\frac{\partial \Phi}{\partial \theta}\left(\theta, x, s_\star^\beta\right) - \frac{\partial \Phi}{\partial \theta}\left(\theta, x, s_\star^{-\beta}\right)\right). \tag{5.22}$$

We refer to Ernoult et al. [2019] and Laborieux et al. [2021] for the implementation and simulation details. Finally, since the models considered here are RNNs, we can also train them with the more conventional *backpropagation through time* (BPTT) algorithm[1], and use BPTT as a benchmark for EqProp.

Table 2 compares the performance on MNIST of the discrete-time models presented in this chapter (FC-#h and ConvNet) with the continuous-time Hopfield networks of Chapter 3 (DHN-#h). No degradation of accuracy is observed when using discrete-time rather than

[1]In this case, BPTT is used on RNNs of a very specific kind. The RNN models considered here have a transition function of the form $F = \frac{\partial \Phi}{\partial s}$, a static input x at each time step, and a single target y at the final time step.

Model	Loss	EqProp Error (%) Estimator	Test	Train	T	K	Epochs	BPTT Error (%) Test	Train
DHN-1h	Squared Error	One-sided	2.06	0.13	100	12	30	2.11	0.46
DHN-2h			2.01	0.11	500	40	50	2.02	0.29
FC-1h	Squared Error	One-sided	2.00	0.20	30	10	30	2.00	0.55
FC-2h			1.95	0.14	100	20	50	2.09	0.37
FC-3h			2.01	0.10	180	20	100	2.30	0.32
ConvNet			1.02	0.54	200	10	40	0.88	0.12

Table 2. Experimental results of Ernoult et al. [2019] on MNIST. EqProp is benchmarked against BPTT. 'DHN' stands for the 'deep Hopfield networks' of Chapter 3. 'FC' means 'fully connected', and '-#h' stands for the number of hidden layers. The test error rates and training error rates (in %) are averaged over five trials. T is the number of iterations in the first phase. K is the number of iterations in the second phase. All these results are obtained with the squared error and the one-sided gradient estimator.

Model	Loss	EqProp Error (%) Estimator	Test	Train	T	K	Epochs	BPTT Error (%) Test	Train
ConvNet	Squared Error	One-sided	86.64	84.90	250	30	120		
		Random Sign	12.61⋆	8.64⋆	250	30	120	11.10	3.69
		Symmetric	12.45	7.83	250	30	120		
ConvNet	Cross-Ent.	Symmetric	11.68	4.98	250	25	120	11.12	2.19

Table 3. Experimental results of Laborieux et al. [2021] on CIFAR-10. EqProp is benchmarked against BPTT. The test error rates and training error rates (in %) are averaged over five trials. T is the number of iterations in the first phase. K is the number of iterations in the second phase. The 'one-sided' and 'symmetric' gradient estimators refer to Eq. 5.21 and Eq. 5.22, respectively. 'random sign' refers to the one-sided estimator with β being positive or negative with even probability.
⋆In the simulations with random β, the training process collapsed in one trial out of five, leading to a performance similar to the one-sided estimator. The test error mean and train error mean reported here include only the four trials that worked fine.

continuous-time networks, although the former require many less time steps in the first phase of training (T). The lowest test error rate ($\sim 1\%$) is achieved with the ConvNet model.

Table 3 shows the performance of a ConvNet model on CIFAR-10, for different gradient estimators and different loss functions. Unlike in the MNIST experiments, the one-sided gradient estimator with a nudging factor (β) of constant sign works poorly on CIFAR-10: training is unstable and the network is unable to fit the training data (84.90% train error). The bias of the one-sided gradient estimator can be reduced on average by choosing the sign of β at random in the second phase: with this technique, Laborieux et al. [2021] report that training proceeded well in four runs out of five, yielding a mean test error of 12.61%, but training collapsed in the last run in a way similar to the one-sided gradient estimator with constant sign. The symmetric difference estimator allows to reduce not only the bias but

also the variance, and to stabilize the training process consistently across runs (12.45% test error). Finally, the best test error rate, 11.68%, is obtained with the cross-entropy loss, and approaches the performance of BPTT with less than 0.6 % degradation in accuracy.

5.3.1. Challenges with EqProp Training

The theoretical guarantee that EqProp can approximate with arbitrary precision the gradient of arbitrary loss functions for a very broad class of models (energy-based models) suggests that EqProp could eventually train large networks on challenging tasks, as was proved feasible in the last decade with other deep learning training methods (e.g. backpropagation) relying on stochastic gradient descent. Nevertheless, EqProp training on current processors (GPUs) presents several challenges.

One difficulty encountered with EqProp training is that, although the gradient formula (Eq. 5.6) requires that $\frac{\partial \Phi}{\partial \theta}$ be measured *exactly* at the fixed points, in many situations however, these fixed points are only approached up to certain precision. Empirically, we observe that for learning to work, the fixed point of the first phase of training must be approximated with very high accuracy ; otherwise the gradient estimate is of poor quality and does not enable to optimize the loss function. This implies that the equations of the first phase of training need to be iterated a large number of time steps, until convergence to the fixed point. Table 2 and Table 3 show that hundreds of iterations are required for the networks to converge, even though these networks consist of just a few layers.

Various methods have been investigated to accelerate convergence, none of which has proved really satisfying so far. Scellier and Bengio [2016] propose a method based on variational inference, in which the state variables are split in two groups (specifically the layers of odd indices and the layers of even indices): at each iteration, one group of state variables remains fixed, while the other group is updated by solving for the stationarity condition. Bengio et al. [2016] give a sufficient condition so that initialization of the network with a forward pass provides sensible initial states for inference ; the condition is that any two successive layers must form a 'good autoencoder'. O'Connor et al. [2018] use a side network to learn these initial states for inference (in the main network).

One promising avenue to solve the problem of long inference times is offered by the recent work of Ramsauer et al. [2020], which shows that for a certain class of *modern Hopfield networks*, equilibrium states are reached in exactly one step. This idea could considerably accelerate simulations in software and demonstrate EqProp training on more advanced architectures and harder tasks. We emphasize however that the difficulty of long inference times is specific to numerical simulations (i.e. simulations on digital computers), and may not be a problem for neuromorphic hardware, where energy minimization is performed by the physics of system (Chapter 4).

A second difficulty with EqProp training is due to the saturation of neurons. All experiments so far have found that for EqProp training to be effective, the neurons' states need to be clipped to a closed interval, typically $[0, 1]$. This is achieved in most experiments by applying the hard-sigmoid activation function $\sigma(s) = \min(\max(0, s), 1)$ after each iteration during inference. Due to this technique however, many neurons 'saturate', i.e. they have a value of exactly 0 or 1 at equilibrium. In the second phase of training, due to these saturated neurons, error signals have difficulty propagating from output neurons across the network, when the nudging factor β is small. To mitigate this problem, in most experiments β is chosen large enough so as to amplify and better propagate error signals along the layers, at the cost of degrading the quality of the gradient estimate. To counter this problem, O'Connor et al. [2018] suggest to use a modified activation function which includes a leak term, namely $\sigma^{\text{mod}}(s) = \sigma(s) + 0.01s$. Another avenue to further reduce the saturation effect is to search weight initialization schemes specifically meant for the kind of network models trained with EqProp. The weight initialisation schemes that dominate deep learning today have been designed to fit feedforward nets [He et al., 2015] and RNNs [Saxe et al., 2013] trained with automatic differentiation. Finding appropriate weight initialization schemes for the kind of bidirectional networks studied in our context is an area of research largely unexplored.

The third difficulty with EqProp training is hyperparameter tuning, due to the high sensitivity of the training process to some of the hyperparameters. Initial learning rates for example need to be tuned layer-wise. In addition to the usual hyperparameters (architecture, learning rates, ...), EqProp requires tuning some additional hyperparameters: the number of iterations in the free phase (T), the number of iterations in the nudged phase (K), the value of the nudging factor (β), ... In the next section, we present a theoretical result called the *GDD property* that can help accelerate hyperparameter search. As we will see, the GDD property provides a criterion to decide whether the fixed point of the first phase has been reached or not.

5.4. Gradient Descending Dynamics (GDD)

The gradient formula of Eq. 5.6 depends only on the fixed points $s_\star$ and $s_\star^\beta$, not on the specific trajectory that the network follows to reach them. But similarly to the real-time setting of Chapter 3, assuming the dynamics of Eq. 5.5 when the neurons gradually move from their free fixed point values ($s_\star$) towards their nudged fixed point values ($s_\star^\beta$), we can show that the transient states of the network (s_t^β for $t \geq 0$) perform step-by-step gradient computation.

5.4.1. Transient Dynamics

First, note that the gradient of EqProp (Eq. 5.6), which is equal to the gradient of the loss in the limit $\beta \to 0$, can be decomposed as a telescoping sum:

$$\frac{1}{\beta}\left(\frac{\partial \Phi}{\partial \theta}\left(\theta, x, s_\star^\beta\right) - \frac{\partial \Phi}{\partial \theta}\left(\theta, x, s_\star\right)\right) = \sum_{t=0}^{\infty} \frac{1}{\beta}\left(\frac{\partial \Phi}{\partial \theta}\left(\theta, x, s_{t+1}^\beta\right) - \frac{\partial \Phi}{\partial \theta}\left(\theta, x, s_t^\beta\right)\right). \tag{5.23}$$

Second, we rewrite the dynamics of the free phase (Eq 5.3) in the form

$$s_{t+1} = \frac{\partial \Phi}{\partial s}\left(\theta_{t+1} = \theta, x, s_t\right), \tag{5.24}$$

where θ_t denotes the parameter of the model at time step t, the value θ being shared across all time steps. We consider the loss after T time steps:

$$\mathcal{L}_T = C\left(s_T, y\right). \tag{5.25}$$

$\mathcal{L}_T$ is what we have called the *projected cost function* in the setting of real-time dynamics (Eq. 3.6). Rewriting the free phase dynamics this way allows us to define the partial derivative $\frac{\partial \mathcal{L}_T}{\partial \theta_t}$ as the sensitivity of the loss $\mathcal{L}_T$ with respect to θ_t, when $\theta_1, \dots \theta_{t-1}, \theta_{t+1}, \dots \theta_T$ remain fixed (set to the value θ). With these notations, the full gradient of the loss can be decomposed as

$$\frac{\partial \mathcal{L}_T}{\partial \theta} = \frac{\partial \mathcal{L}_T}{\partial \theta_1} + \frac{\partial \mathcal{L}_T}{\partial \theta_2} + \dots + \frac{\partial \mathcal{L}_T}{\partial \theta_T}. \tag{5.26}$$

The following result links the right-hand sides of Eq. 5.23 and Eq. 5.26 term by term.

Theorem 5.1 (Ernoult et al. [2019])**.** *Let $s_0, s_1, \dots, s_T$ be the sequence of states in the free phase. Suppose that the sequence has converged to the fixed point $s_\star$ after $T - K$ time steps for some $K \geq 0$, i.e. that $s_\star = s_T = s_{T-1} = \dots s_{T-K}$. Then, the following identities hold at any time $t = 0, 1, \dots, K-1$ in the nudged phase:*

$$\lim_{\beta \to 0} \frac{1}{\beta}\left(\frac{\partial \Phi}{\partial \theta}\left(\theta, x, s_{t+1}^\beta\right) - \frac{\partial \Phi}{\partial \theta}\left(\theta, x, s_t^\beta\right)\right) = -\frac{\partial \mathcal{L}_T}{\partial \theta_{T-t}}, \tag{5.27}$$

$$\lim_{\beta \to 0} \frac{1}{\beta}\left(s_{t+1}^\beta - s_t^\beta\right) = -\frac{\partial \mathcal{L}_T}{\partial s_{T-t}}. \tag{5.28}$$

We refer to Ernoult et al. [2019] for a proof. Theorem 5.1 relates neural computation to gradient computation, and is as such a discrete-time variant of Theorem 3.1. In essence, Theorem 5.1 shows that in the nudged phase of EqProp, the temporal variations in neural activity and incremental weight updates represent loss gradients. Since the sequence of states in the nudged phase satisfies $s_{t+1}^\beta = s_t^\beta - \beta \frac{\partial \mathcal{L}_T}{\partial s_{T-t}} + o(\beta)$ as $\beta \to 0$, descending the gradients of the loss $\mathcal{L}_T$, we call this property the *gradient descending dynamics* (GDD) property.

As mentioned in section 5.3.1, one of the challenges with EqProp training comes from the empirical observation that learning is successful only if we are *exactly* at the fixed point at the end of the first phase, although in practice we use numerical methods to *approximate*

this fixed point. In particular, we need a criterion to 'decide' when the fixed point has been reached with high enough accuracy. Theorem 5.1 provides such a criterion: a necessary condition for the fixed point of the first phase to be reached is that the identities of Eqs. 5.27-5.28 hold.

5.4.2. Backpropagation Through Time

On the one hand we can define the neural and weight increments of EqProp as follows, which we can compute in the second phase of EqProp:

$$\Delta_\theta^{\mathrm{EP}}(\beta, t) = \frac{1}{\beta}\left(\frac{\partial \Phi}{\partial \theta}\left(\theta, x, s_{t+1}^\beta\right) - \frac{\partial \Phi}{\partial \theta}\left(\theta, x, s_t^\beta\right)\right), \tag{5.29}$$

$$\Delta_s^{\mathrm{EP}}(\beta, t) = \frac{1}{\beta}\left(s_{t+1}^\beta - s_t^\beta\right). \tag{5.30}$$

On the other hand, the loss gradients $\frac{\partial \mathcal{L}_T}{\partial s_{T-t}}$ and $\frac{\partial \mathcal{L}_T}{\partial \theta_{T-t}}$ appearing on the right-hand sides of Eqs. 5.27-5.28 can be computed by automatic differentiation. Specifically, these loss gradients are the 'partial derivatives' computed by the *backpropagation through time* (BPTT) algorithm. Here BPTT is applied in the very specific setting of an RNN with transition function $F = \frac{\partial \Phi}{\partial s}$, with static input x at each time step, and with target y at the final time step. In particular, there is no time-dependence in the data. We denote these partial derivatives computed by BPTT:

$$\nabla_\theta^{\mathrm{BPTT}}(t) = \frac{\partial \mathcal{L}_T}{\partial \theta_{T-t}}, \tag{5.31}$$

$$\nabla_s^{\mathrm{BPTT}}(t) = \frac{\partial \mathcal{L}_T}{\partial s_{T-t}}. \tag{5.32}$$

Using these notations, the GDD property (Theorem 5.1) states that under the condition that $s_\star = s_T = s_{T-1} = \dots s_{T-K}$, we have for every $t = 0, 1, \dots K-1$ that $\lim_{\beta \to 0} \Delta_s^{\mathrm{EP}}(\beta, t) = -\nabla_s^{\mathrm{BPTT}}(t)$ and $\lim_{\beta \to 0} \Delta_\theta^{\mathrm{EP}}(\beta, t) = -\nabla_\theta^{\mathrm{BPTT}}(t)$. The GDD property is illustrated in Figure 8.

We note that the GDD property also implies that the gradient computed by 'truncated EqProp' (i.e. EqProp where the second phase is halted before convergence to the second fixed point) corresponds to truncated BPTT.

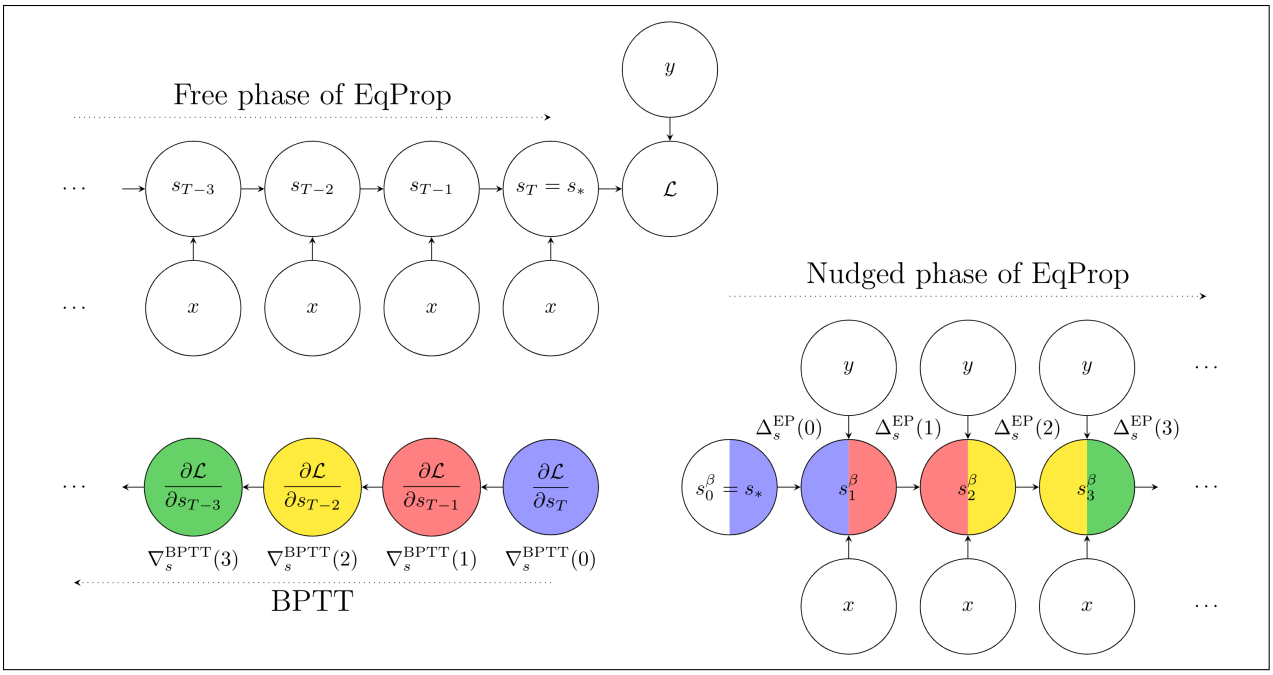

Fig. 8. Illustration of the gradient-descending dynamics (GDD) property (Theorem 5.1). Top left. Free phase. The final state s_T is the fixed point $s_\star$. **Bottom left.** Backpropagation through time (BPTT), in the very specific setting of an RNN with static input x. **Bottom right.** Nudged phase of EqProp. The starting state in the nudged phase is the final state of the free phase, i.e. the fixed point $s_\star$. **Theorem 5.1.** Step by step correspondence between the neural increments $\Delta_s^{\mathrm{EP}}(t)$ in the nudged phase of EqProp and the gradients $\nabla_s^{\mathrm{BPTT}}(t)$ of BPTT. Corresponding computations in EqProp and BPTT at timestep $t = 0$ (resp. $t = 1, 2, 3$) are colored in blue (resp. red, yellow, green). Forward-time computation in EqProp corresponds to backward-time computation in BPTT.

Chapter 6

Extensions of Equilibrium Propagation

In this chapter, we present research directions for the development of the equilibrium propagation framework. In section 6.1, we present a general framework for training dynamical systems with time-varying inputs, which exploits the principle of least action. In section 6.2, we adapt the EqProp framework to the setting of stochastic systems. In section 6.3, we briefly present the *contrastive meta-learning* framework of Zucchet et al. [2021], where they use the EqProp method to train the meta-parameters of a meta-learning model.

6.1. Equilibrium Propagation in Dynamical Systems with Time-Varying Inputs

In Chapter 2, we have derived the EqProp training procedure for a class of models called energy-based models (EBMs). A key element of the theory is the fact that, in EBMs, equilibrium states are characterized by variational equations. In this section, we show that the EqProp training strategy can be applied to other situations where variational equations appear. The equations of motion of many physical systems can also be characterized by variational equations – their trajectory can be derived through a *principle of stationary action* (e.g. a principle of least action). In such systems, the quantity that is stationary is not the energy function (as in an EBM), but the *action functional*, which is by definition the time integral of the Lagrangian function. Such systems, which we call *Lagrangian-based models* (LBMs), can play the role of machine learning models with time-varying inputs. This idea was also proposed by Baldi and Pineda [1991] in the context of the *contrastive learning framework*, and independently proposed by Kendall [2021] in the context of time-varying electrical networks.

6.1.1. Lagrangian-Based Models

A Lagrangian-based model (LBM) is specified by a set of adjustable parameters, denoted θ, a time-varying input, and a state variable. We write $\mathbf{x}_t$ the input value at time t, and $\mathbf{s}_t$ the state of the model at time t. We study the evolution of the system over a time interval $[0, T]$, and we write $\mathbf{x}$ and $\mathbf{s}$ the entire input and state trajectories over this time interval. The model is further described in terms of a *functional* $\mathcal{S}$ which, given a parameter value θ and an input trajectory $\mathbf{x}$, associates to each trajectory $\mathbf{s}$ the real number

$$\mathcal{S}(\theta, \mathbf{x}, \mathbf{s}) = \int_0^T L(\theta, \mathbf{x}_t, \mathbf{s}_t, \dot{\mathbf{s}}_t) dt, \tag{6.1}$$

where $L(\theta, \mathbf{x}_t, \mathbf{s}_t, \dot{\mathbf{s}}_t)$ is a scalar function of the parameters (θ), the external input ($\mathbf{x}_t$), the state of the system ($\mathbf{s}_t$) as well as its time derivative ($\dot{\mathbf{s}}_t$). The function L is called the *Lagrangian function* of the system, and $\mathcal{S}$ is called the *action functional*. The action functional is defined for any *conceivable* trajectory $\mathbf{s}$, but, among all conceivable trajectories, the *effective* trajectory of the system (subject to θ and $\mathbf{x}$), denoted $\mathbf{s}(\theta, \mathbf{x})$, satisfies by definition $\frac{d}{d\epsilon}\Big|_{\epsilon=0} \mathcal{S}(\theta, \mathbf{x}, \mathbf{s}(\theta, \mathbf{x}) + \epsilon \mathbf{s}) = 0$ for any trajectory $\mathbf{s}$ that satisfies the boundary conditions $\mathbf{s}_0 = 0$ and $\mathbf{s}_T = 0$. We write for short

$$\frac{\delta \mathcal{S}}{\delta \mathbf{s}}(\theta, \mathbf{x}, \mathbf{s}(\theta, \mathbf{x})) = 0. \tag{6.2}$$

Intuitively, $\delta\mathcal{S}$ can be thought of as the variation of $\mathcal{S}$ associated to a small variation $\delta\mathbf{s}$ around the trajectory $\mathbf{s}(\theta, \mathbf{x})$. Mathematically, $\frac{\delta \mathcal{S}}{\delta \mathbf{s}}(\theta, \mathbf{x}, \mathbf{s}(\theta, \mathbf{x}))$ represents the differential of the function $\mathcal{S}(\theta, \mathbf{x}, \cdot)$ at the point $\mathbf{s}(\theta, \mathbf{x})$. We say that the effective trajectory is stationary with respect to the action functional, and that the dynamics of the system derives from a *principle of stationary action*. Since the action functional ($\mathcal{S}$) is defined in terms of the Lagrangian of the system (L), we call such a time-varying system a *Lagrangian-based model*.

The loss to be minimized is an integral of cost values of the states along the effective trajectory:

$$\mathcal{L}_0^T(\theta, \mathbf{x}, \mathbf{y}) = \int_0^T c_t(\mathbf{s}_t(\theta, \mathbf{x}), \mathbf{y}_t) dt. \tag{6.3}$$

In this expression, $\mathbf{y}_t$ is the desired target at time t, $\mathbf{y}$ is the corresponding target trajectory, $\mathbf{s}_t(\theta, \mathbf{x})$ is the state at time t along the effective trajectory $\mathbf{s}(\theta, \mathbf{x})$, and $c_t(\mathbf{s}_t, \mathbf{y}_t)$ is a scalar function (the *cost function* at time t).

Similarly to the setting of EBMs, the concept of *sum-separability* is useful in LBMs. Let $\theta = (\theta_1, \ldots, \theta_N)$ be the adjustable parameters of the system. Let $\{\mathbf{x}_t, \mathbf{s}_t, \dot{\mathbf{s}}_t\}_k$ denote the information about $(\mathbf{x}_t, \mathbf{s}_t, \dot{\mathbf{s}}_t)$ at time t which is locally available to parameter θ_k. We say that the Lagrangian L is *sum-separable* if it is of the form

$$L(\theta, \mathbf{x}_t, \mathbf{s}_t, \dot{\mathbf{s}}_t) = L_0(\mathbf{x}_t, \mathbf{s}_t, \dot{\mathbf{s}}_t) + \sum_{k=1}^{N} L_k(\theta_k, \{\mathbf{x}_t, \mathbf{s}_t, \dot{\mathbf{s}}_t\}_k), \tag{6.4}$$

where $L_0(\mathbf{x}_t, \mathbf{s}_t, \dot{\mathbf{s}}_t)$ is a term that is independent of the parameters to be adjusted, and L_k is a scalar function of θ_k and $\{\mathbf{x}_t, \mathbf{s}_t, \dot{\mathbf{s}}_t\}_k$ for each $k \in \{1, \dots, N\}$.

6.1.2. Gradient Formula

Similarly to the static case (Section 2.3), we introduce the *total action functional*

$$\mathcal{S}^\beta(\mathbf{s}) = \int_0^T \left(L(\theta, \mathbf{x}_t, \mathbf{s}_t, \dot{\mathbf{s}}_t) + \beta\, c_t(\mathbf{s}_t, \mathbf{y}_t)\right) dt, \tag{6.5}$$

defined for any value of the nudging factor β (for fixed θ, $\mathbf{x}$ and $\mathbf{y}$). Intuitively, by varying β, the action functional $\mathcal{S}^\beta$ is modified, and so is the stationary solution of the action, i.e. the effective trajectory of the system. Specifically, let us denote $\mathbf{s}^\beta$ the trajectory characterized by the stationarity condition $\frac{\delta \mathcal{S}^\beta}{\delta \mathbf{s}}(\mathbf{s}^\beta) = 0$. Note in particular that for $\beta = 0$ we have $\mathbf{s}^0 = \mathbf{s}(\theta, \mathbf{x})$.

Theorem 6.1 (Gradient formula for Lagrangian-based models)**.** *The gradient of the loss can be computed using the following formula:*

$$\frac{\partial \mathcal{L}_0^T}{\partial \theta}(\theta, \mathbf{x}, \mathbf{y}) = \left.\frac{d}{d\beta}\right|_{\beta=0} \int_0^T \frac{\partial L}{\partial \theta}\left(\theta, \mathbf{x}_t, \mathbf{s}_t^\beta, \dot{\mathbf{s}}_t^\beta\right) dt \tag{6.6}$$

Furthermore, if the Lagrangian function L is sum-separable, then the gradient for each parameter θ_k depends only on information that is locally available to θ_k:

$$\frac{\partial \mathcal{L}_0^T}{\partial \theta_k}(\theta, \mathbf{x}, \mathbf{y}) = \left.\frac{d}{d\beta}\right|_{\beta=0} \int_0^T \frac{\partial L_k}{\partial \theta_k}(\theta_k, \{\mathbf{x_t}, \mathbf{s}_t^\beta, \dot{\mathbf{s}}_t^\beta\}_k) dt. \tag{6.7}$$

Proof of Theorem 6.1. We derive Theorem 6.1 as a corollary of Theorem 2.1. Recall that the action functional is by definition $\mathcal{S}(\theta, \mathbf{x}, \mathbf{s}) = \int_0^T L(\theta, \mathbf{x}_t, \mathbf{s}_t, \dot{\mathbf{s}}_t)$ and that the effective trajectory $\mathbf{s}(\theta, \mathbf{x})$ satisfies the stationary condition $\frac{\delta \mathcal{S}}{\delta \mathbf{s}}(\theta, \mathbf{x}, \mathbf{s}(\theta, \mathbf{x})) = 0$. We can define a cost functional $\mathcal{C}$ on any conceivable trajectory $\mathbf{s}$ by the formula $\mathcal{C}(\mathbf{s}, \mathbf{y}) = \int_0^T c_t(\mathbf{s}_t, \mathbf{y}_t) dt$. The loss $\mathcal{L}_0^T$ then rewrites $\mathcal{L}_0^T(\theta, \mathbf{x}, \mathbf{y}) = \mathcal{C}(\mathbf{s}(\theta, \mathbf{x}), \mathbf{y})$, the total action functional rewrites $\mathcal{S}^\beta(\mathbf{s}) = \mathcal{S}(\theta, \mathbf{x}, \mathbf{s}) + \beta\, \mathcal{C}(\mathbf{s}, \mathbf{y})$, and the nudged trajectory $\mathbf{s}^\beta$ satisfies the stationarity condition $\frac{\delta \mathcal{S}}{\delta \mathbf{s}}(\theta, \mathbf{x}, \mathbf{s}^\beta) + \beta\, \frac{\delta \mathcal{C}}{\delta \mathbf{s}}(\mathbf{s}^\beta, \mathbf{y}) = 0$.

Using these notations, the first formula to be proved (Eq. 6.6) rewrites

$$\frac{\partial \mathcal{L}_0^T}{\partial \theta}(\theta, \mathbf{x}, \mathbf{y}) = \left.\frac{d}{d\beta}\right|_{\beta=0} \frac{\partial \mathcal{S}}{\partial \theta}\left(\theta, \mathbf{x}, \mathbf{s}^\beta\right), \tag{6.8}$$

which is exactly the first formula of Theorem 2.1. Finally, the second formula to be proved (Eq. 6.7) is a direct consequence of Eq. 6.6 and the definition of sum-separability (Eq. 6.4). □

6.1.3. Training Sum-Separable Lagrangian-Based Models

Theorem 6.1 suggests the following EqProp-like training procedure for Lagrangian-based models, to update the parameters in proportion to their loss gradients. Let us assume that the Lagrangian function has the sum-separability property.

Free phase (inference). Set the system in some initial state $(\mathbf{s}_0, \dot{\mathbf{s}}_0)$ at time $t = 0$, and set the nudging factor β to zero. Play the input trajectory $\mathbf{x}$ over the time interval $[0, T]$, and let the system follow the trajectory $\mathbf{s}^0$ (i.e. the effective trajectory characterized by Eq. 6.2). We call $\mathbf{s}^0$ the *free trajectory*. For each parameter θ_k, the quantity $\frac{\partial L_k}{\partial \theta_k}(\theta_k, \{\mathbf{x_t}, \mathbf{s}_t^0, \dot{\mathbf{s}}_t^0\}_k)$ is measured and integrated from $t = 0$ to $t = T$, and the result is stored locally.

Nudged phase. Set the system in the same initial state $(\mathbf{s}_0, \dot{\mathbf{s}}_0)$ as in the free phase, and set now the nudging factor β to some positive or negative (nonzero) value. Play again the input trajectory $\mathbf{x}$ over the time interval $[0, T]$, as well as the target trajectory $\mathbf{y}$, and let the system follow the trajectory $\mathbf{s}^\beta$ (i.e. the effective trajectory that is stationary with respect to $\mathcal{S}^\beta$). For each parameter θ_k, the quantity $\frac{\partial L_k}{\partial \theta_k}(\theta_k, \{\mathbf{x_t}, \mathbf{s}_t^\beta, \dot{\mathbf{s}}_t^\beta\}_k)$ is measured and integrated from $t = 0$ to $t = T$.

Update rule. Finally, each parameter θ_k is updated locally in proportion to its gradient, i.e. $\Delta\theta_k = -\eta \widehat{\nabla}_{\theta_k}(\beta)$, where η is a learning rate, and

$$\widehat{\nabla}_{\theta_k}(\beta) = \frac{1}{\beta}\left(\int_0^T \frac{\partial L_k}{\partial \theta_k}(\theta_k, \{\mathbf{x_t}, \mathbf{s}_t^\beta, \dot{\mathbf{s}}_t^\beta\}_k)dt - \int_0^T \frac{\partial L_k}{\partial \theta_k}(\theta_k, \{\mathbf{x_t}, \mathbf{s}_t^0, \dot{\mathbf{s}}_t^0\}_k)dt\right). \tag{6.9}$$

As in the static setting (Chapter 2), it is possible to reduce the bias and the variance of the gradient estimator by using the symmetrized version

$$\widehat{\nabla}^{\text{sym}}_{\theta_k}(\beta) = \frac{1}{2\beta}\left(\int_0^T \frac{\partial L_k}{\partial \theta_k}(\theta_k, \{\mathbf{x_t}, \mathbf{s}_t^\beta, \dot{\mathbf{s}}_t^\beta\}_k)dt - \int_0^T \frac{\partial L_k}{\partial \theta_k}(\theta_k, \{\mathbf{x_t}, \mathbf{s}_t^{-\beta}, \dot{\mathbf{s}}_t^{-\beta}\}_k)dt\right). \tag{6.10}$$

This requires two nudged phases: one with a positive nudging $(+\beta)$ and one with a negative nudging $(-\beta)$.

Although the EqProp training method for Lagrangian-based models requires running the input trajectory twice (in the free phase and in the nudged phase), we stress that we do not require to store the past states of the system, unlike the backpropagation through time (BPTT) algorithm used to train conventional recurrent neural networks.

6.1.4. From Energy-Based to Lagrangian-Based Models

Conceptually, we have the following correspondence between the static setting (energy-based models) and the time-varying setting (Lagrangian-based models).

- The concept of *configuration* (s) is replaced by that of *trajectory* ($\mathbf{s}$). A trajectory $\mathbf{s}$ is a function from the time interval $[0, T]$ to the space of configurations, which assigns to each time $t \in [0, T]$ a configuration $\mathbf{s}_t$.
- The concept of *energy function* (E) is replaced by that of *action functional* ($\mathcal{S}$). Whereas an energy function E assigns a real number $E(s)$ to each configuration s, an action functional $\mathcal{S}$ assigns a real number $\mathcal{S}(\mathbf{s})$ to each trajectory $\mathbf{s}$.
- The concept of *equilibrium state* (denoted $s(\theta, x)$ or $s_\star$) is replaced by that of *effective trajectory* (denoted $\mathbf{s}(\theta, \mathbf{x})$). Whereas an equilibrium state is characterized by the stationarity of the energy ($\frac{\partial E}{\partial s} = 0$), an effective trajectory is characterized by the stationarity of the action ($\frac{\delta \mathcal{S}}{\delta \mathbf{s}} = 0$).

6.1.5. Lagrangian-Based Models Include Energy-Based Models

Consider a Lagrangian-based model whose Lagrangian function does not depend on $\dot{\mathbf{s}}_t$, i.e. L is of the form

$$L(\theta, \mathbf{x}_t, \mathbf{s}_t, \dot{\mathbf{s}}_t) = E(\theta, \mathbf{x}_t, \mathbf{s}_t). \tag{6.11}$$

Further suppose that the input signal $\mathbf{x}$ is static, i.e. $\mathbf{x}_t = x$ for any t. Denote $s_\star$ the equilibrium state characterized by $\frac{\partial E}{\partial s}(\theta, x, s_\star) = 0$. Then the trajectory $\mathbf{s}$ constantly equal to $s_\star$ (i.e. such that $\mathbf{s}_t = s_\star$ for all t) is a stationary solution of the action functional

$$\mathcal{S}(\theta, x, \mathbf{s}) = \int_0^T E(\theta, x, \mathbf{s}_t) dt. \tag{6.12}$$

Indeed, for any variationa $\delta \mathbf{s}$ around $\mathbf{s}$, we have $\delta \mathcal{S} = \int_0^T \delta E(\theta, x, \mathbf{s}_t) dt = \int_0^T \frac{\partial E}{\partial s}(\theta, x, s_\star) \cdot \delta \mathbf{s_t} dt = 0$. In this sense, energy-based models are special instances of Lagrangian-based models. Furthermore, assuming that the target signal $\mathbf{y}$ and the cost function c_t are also static (i.e. $\mathbf{y}_t = y$ and $c_t = c$ at any time t), then the loss is equal to $\mathcal{L}_0^T = \int_0^T c(s_\star, y) dt$, which is the loss in the static setting (up to a constant T). In this case, the EqProp learning algorithm for Lagrangian-based models boils down to the EqProp learning algorithm for energy-based models (up to a constant T).

6.2. Equilibrium Propagation in Stochastic Systems

Unlike neural networks trained on digital computers which can reliably process information in a deterministic way, physical systems (including analog circuits and biological networks) are subject to noise. In this section we present an extension of the equilibrium propagation framework to stochastic systems, which allows us to take such forms of noise into account, and may therefore be useful both from the neuromorphic and neuroscience points of view.

We note that the question whether the brain is stochastic or deterministic is controversial. However, even if the brain were deterministic, the precise trajectory of the neural activity is likely to be fundamentally unpredictable (i.e. chaotic) and thus easier to study statistically. In this case, the brain can still be usefully modelled with probability distributions (using probability theory or ergodic theory).

6.2.1. From Deterministic to Stochastic Systems

In the stochastic setting, when presented with an input x, instead of an equilibrium state $s_\star$, the model defines a probability distribution $p_\star(s)$ over the space of possible configurations s. Thus, rather than a stationary condition of the form $\frac{\partial E}{\partial s}(\theta, x, s_\star) = 0$, we now have an equilibrium distribution $p_\star(s)$ such that

$$p_\star(s) = \frac{e^{-E(\theta,x,s)}}{Z_\star}, \qquad \text{with} \qquad Z_\star = \int e^{-E(\theta,x,s)} ds. \tag{6.13}$$

The probability distribution defined by $p_\star(s)$ is called the Boltzmann distribution (or Gibbs distribution), and the normalizing constant $Z_\star$ is called the *partition function*. In this setting, the loss that we want to minimize is the expected cost over the equilibrium distribution

$$\mathcal{L}_{\text{sto}}(\theta, x, y) = \mathbb{E}_{s\sim p_\star(s)} \left[C\left(s, y\right) \right]. \tag{6.14}$$

We note that $\mathcal{L}_{\text{sto}}$ depends on θ and x through the equilibrium distribution $p_\star(s)$.

6.2.2. Gradient Formula

As in the deterministic framework, the stochastic version of equilibrium propagation makes use of the total energy function $E(\theta, x, s) + \beta\, C(s, y)$. The notion of nudged equilibrium state $(s_\star^\beta)$ is replaced accordingly by a nudged equilibrium distribution $p_\star^\beta(s)$, which is the Boltzmann distribution associated to the total energy function, i.e.

$$p_\star^\beta(s) = \frac{e^{-E(\theta,x,s) - \beta\, C(s,y)}}{Z_\star^\beta}, \qquad \text{with} \qquad Z_\star^\beta = \int e^{-E(\theta,x,s) - \beta\, C(s,y)} ds. \tag{6.15}$$

The following theorem extends Theorem 2.1 to stochastic systems.

Theorem 6.2 (Scellier and Bengio [2017])**.** *The gradient of the objective function with respect to θ is equal to*

$$\frac{\partial \mathcal{L}_{\text{sto}}}{\partial \theta}(\theta, x, y) = \frac{d}{d\beta}\bigg|_{\beta=0} \mathbb{E}_{s\sim p_\star^\beta(s)} \left[\frac{\partial E}{\partial \theta}\left(\theta, x, s\right) \right]. \tag{6.16}$$

Furthermore, if the energy function is sum-separable (in the sense of Eq. 2.5), then

$$\frac{\partial \mathcal{L}_{\text{sto}}}{\partial \theta_k}(\theta, x, y) = \frac{d}{d\beta}\bigg|_{\beta=0} \mathbb{E}_{s\sim p_\star^\beta(s)} \left[\frac{\partial E_k}{\partial \theta_k}\left(\theta_k, \{x, s\}_k\right) \right]. \tag{6.17}$$

PROOF OF THEOREM 6.2. Recall that the total energy function is by definition $F(\theta, \beta, s) = E(\theta, x, s) + \beta\ C(s, y)$, where the notations x and y are dropped for simplicity (since they do not play any role in the proof). We also (re)define $Z_\theta^\beta = \int e^{-F(\theta,\beta,s)} ds$, the partition function, as well as $p_\theta^\beta(s) = \frac{e^{-F(\theta,\beta,s)}}{Z_\theta^\beta}$, the corresponding Boltzmann distribution. Recalling the definition of the loss $\mathcal{L}_{\text{sto}}$ (Eq. 6.14), and using the fact that $\frac{\partial F}{\partial \beta} = C$ and that $\frac{\partial F}{\partial \theta} = \frac{\partial E}{\partial \theta}$, the formula to show (Eq. 6.16) is a particular case of the following formula, evaluated at the point $\beta = 0$:

$$\frac{d}{d\theta}\mathbb{E}_{s\sim p_\theta^\beta(s)}\left[\frac{\partial F}{\partial \beta}(\theta, \beta, s)\right] = \frac{d}{d\beta}\mathbb{E}_{s\sim p_\theta^\beta(s)}\left[\frac{\partial F}{\partial \theta}(\theta, \beta, s)\right]. \tag{6.18}$$

Therefore, in order to prove Theorem 6.2, it is sufficient to prove Eq. 6.18. We do this in two steps. First, the cross-derivatives of the log-partition function $\ln\left(Z_\theta^\beta\right)$ are equal:

$$\frac{d}{d\theta}\frac{d}{d\beta}\ln\left(Z_\theta^\beta\right) = \frac{d}{d\beta}\frac{d}{d\theta}\ln\left(Z_\theta^\beta\right). \tag{6.19}$$

Second, we have

$$\frac{d}{d\beta}\ln\left(Z_\theta^\beta\right) = \mathbb{E}_{s\sim p_\theta^\beta(s)}\left[\frac{\partial F}{\partial \beta}(\theta, \beta, s)\right], \tag{6.20}$$

and similarly

$$\frac{d}{d\theta}\ln\left(Z_\theta^\beta\right) = \mathbb{E}_{s\sim p_\theta^\beta(s)}\left[\frac{\partial F}{\partial \theta}(\theta, \beta, s)\right]. \tag{6.21}$$

Plugging Eq. 6.20 and Eq. 6.21 in Eq. 6.19, we get Eq. 6.18. Hence the result. □

We note that Eq. 6.18 is a stochastic variant of the fundamental lemma of EqProp (Lemma 2.2). The quantity $-\ln\left(Z_\theta^\beta\right)$ is called the *free energy* of the system.

6.2.3. Langevin Dynamics

The prototypical dynamical system to sample from the equilibrium distribution $p_\star(s)$ (Eq. 6.13) is the *Langevin dynamics*, which we describe here. Recall from Chapter 3 the gradient dynamics $\frac{ds_t}{dt} = -\frac{\partial E}{\partial s}(\theta, x, s_t)$. To go from this (deterministic) gradient dynamics to the (stochastic) Langevin dynamics, we add a new term (a Brownian term) which models a form of noise:

$$dS_t = -\frac{\partial E}{\partial s}(\theta, x, S_t)\, dt + \sqrt{2}\, dB_t. \tag{6.22}$$

In this expression, B_t is a mathematical object called a *Brownian motion*. Instead of defining B_t formally, we give here an intuitive definition. Intuitively, each increment dB_t (between time t and time $t + dt$) can be thought of as a normal random variable with mean 0 and variance dt, which is "independent of past increments". By following this noisy form of gradient descent with respect to the energy function E, the state of the system (S_t) settles to the Boltzmann distribution. This can be proved using the Kolmogorov forward equation (a.k.a. Fokker-Planck equation) for diffusion processes.

Here we have chosen the constant $\sqrt{2}$ in the Langevin dynamics, so that the 'temperature' of the system is 1. More generally, if the Brownian motion is scaled by a factor $\sigma(\theta, x)$, i.e. if the dynamics is of the form $dS_t = -\frac{\partial E_\theta}{\partial s}(\theta, x, S_t)\, dt + \sigma(\theta, x) \cdot dB_t$, then the exponent in the Boltzmann distribution needs to be rescaled by a factor $\frac{1}{2}\sigma^2(\theta, x)$. We call this modified equilibrium distribution the Boltzmann distribution with temperature $T = \frac{1}{2}\sigma^2(\theta, x)$. We note that if $\sigma(\theta, x) = \sqrt{2}$ then $T = 1$.

6.2.4. Equilibrium Propagation in Langevin Dynamics

In the setting of Langevin dynamics, EqProp takes the following form.

Free phase (inference). In the free phase, the network is shown an input x and the state of the system follows the Langevin dynamics of Eq. 6.22. 'Free samples' are drawn from the equilibrium distribution $p_\star(s) \propto e^{-E(\theta,x,s)}$.

Nudged phase. In the nudged phase, a term $-\beta \frac{\partial C}{\partial s}$ is added to the dynamics of Eq. 6.22, where β is a scalar hyperparameter (the nudging factor). Denoting S_t^β the state of the network at time t in the nudged phase, the dynamics reads:

$$dS_t^\beta = \left[-\frac{\partial E}{\partial s}\left(\theta, x, S_t^\beta\right) - \beta \frac{\partial C}{\partial s}\left(S_t^\beta, y\right)\right] dt + \sqrt{2}\, dB_t. \tag{6.23}$$

Here for readability we use the same notation B_t for the Brownian motion of the nudged phase, but it should be understood that this is a new Brownian motion, independent of the one used in the free phase. 'Nudged samples' are drawn from the nudged distribution $p_\star^\beta(s) \propto e^{-E(\theta,x,s)-\beta\, C(s,y)}$.

Gradient estimate. Finally, the gradient of the loss $\mathcal{L}_{\text{sto}}$ of Eq. 6.14 can be approximated using the samples from the free and nudged distributions to estimate:

$$\widehat{\nabla}_\theta(\beta) = \frac{1}{\beta}\left(\mathbb{E}_{s\sim p_\star^\beta(s)}\left[\frac{\partial E}{\partial \theta}(\theta, x, s)\right] - \mathbb{E}_{s\sim p_\star(s)}\left[\frac{\partial E}{\partial \theta}(\theta, x, s)\right]\right). \tag{6.24}$$

6.3. Contrastive Meta-Learning

Recently, Zucchet et al. [2021] introduced the *contrastive meta-learning* framework, where they propose to train the meta-parameters of a meta-learning model using the EqProp method. In this section, we briefly present the setting of meta-learning and show how Zucchet et al. [2021] derive the contrastive meta-learning method.

6.3.1. Meta-Learning and Few-Shot Learning

Meta learning, or *learning to learn*, is a broad field that encompasses *hyperparameter optimization*, *few-shot learning*, and many other use cases. Here, for concreteness, we present the setting of few-shot learning.

In the setting of few-shot learning, the aim is to build a system that is able to learn (or 'adapt' to) a given task $\mathcal{T}$ when only very limited data is available for that task. The system should be able to do so for a variety of tasks coming from a distribution of tasks $p(\mathcal{T})$. In this setting, the system has two types of parameters: a *meta-parameter* θ which is shared across all tasks, and a *task-specific parameter* ϕ which can be adapted to a given task. In the *adaptation phase*, the task-specific parameter ϕ adapts to some task $\mathcal{T}$ using a training set $\mathcal{D}_{\text{train}}$ corresponding to that task. The resulting value of the task-specific parameter after this adaptation phase is denoted $\phi(\theta, \mathcal{D}_{\text{train}})$, which depends on both θ and $\mathcal{D}_{\text{train}}$. The performance of the resulting $\phi(\theta, \mathcal{D}_{\text{train}})$ is then evaluated on a test set $\mathcal{D}_{\text{test}}$ from the same task $\mathcal{T}$. This performance is denoted $L(\phi(\theta, \mathcal{D}_{\text{train}}), \mathcal{D}_{\text{test}})$, where L is a loss function. The goal of meta-learning is then to find the value of the meta-parameter θ that minimizes the expected loss $\mathcal{R}(\theta) = \mathbb{E}_{(\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{test}})}\left[L(\phi(\theta, \mathcal{D}_{\text{train}}), \mathcal{D}_{\text{test}})\right]$ over pairs of training/test sets $(\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{test}})$ coming from the distribution of tasks $p(\mathcal{T})$. In other words, the goal is to find θ that generalizes well across tasks from the distribution $p(\mathcal{T})$.

6.3.2. Contrastive Meta-Learning

The idea of the *contrastive meta-learning* framework of Zucchet et al. [2021] is the following. In the adaptation phase, the task-specific parameter ϕ minimizes an *inner loss* L^{in}, so that

$$\phi(\theta, \mathcal{D}_{\text{train}}) = \arg\min_{\phi} L^{\text{in}}(\theta, \mathcal{D}_{\text{train}}, \phi). \tag{6.25}$$

In the setting of *regularization learning* for example, the inner loss is of the form $L^{\text{in}}(\theta, \mathcal{D}_{\text{train}}, \phi) = L(\phi, \mathcal{D}_{\text{train}}) + R(\theta, \phi)$, where L is the same loss as the one used on the test set, and $R(\theta, \phi)$ is a regularization term. Exploiting the fact that, at the end of the adaptation phase, the task-specific parameter $\phi(\theta, \mathcal{D}_{\text{train}})$ satisfies the 'equilibrium condition' $\frac{\partial L^{\text{in}}}{\partial \phi}(\theta, \mathcal{D}_{\text{train}}, \phi(\theta, \mathcal{D}_{\text{train}})) = 0$, Zucchet et al. [2021] then propose to use the EqProp method to compute the gradients (with respect to θ) of the meta loss

$$\mathcal{L}_{\text{meta}} = L^{\text{out}}(\phi(\theta, \mathcal{D}_{\text{train}}), \mathcal{D}_{\text{test}}), \tag{6.26}$$

where L^{out} is a so-called *outer loss*, e.g. $L^{\text{out}} = L$ in regularization learning.

More generally, the contrastive meta-learning method applies to any functions L^{in} and L^{out}, and any bilevel optimization problem where the aim is to optimize $\mathcal{L}_{\text{meta}}(\theta) = L^{\text{out}}(\theta, \phi(\theta))$ with respect to θ, under the constraint that $\frac{\partial L^{\text{in}}}{\partial \phi}(\theta, \phi(\theta)) = 0$.

Chapter 7

Conclusion

In this thesis, we have presented a mathematical framework that applies to systems that are described by variational equations, while maintaining the benefits of backpropagation. This framework may have implications both for neuromorphic computing and for neuroscience.

7.1. Implications for Neuromorphic Computing

Current deep learning research is grounded on a very general and powerful mathematical principle: automatic differentiation for backpropagating error gradients in differentiable neural networks. This mathematical principle is at the heart of all deep learning libraries (TensorFlow, PyTorch, Theano, etc.). The emergence of such software libraries has greatly eased deep learning research and fostered the large scale development of parallel processors for deep learning (e.g. GPUs and TPUs). However, these processors are power inefficient by orders of magnitude, if we take the brain as a benchmark. The rapid increase in energy consumption raises concerns as the use of deep learning systems in society keeps growing [Strubell et al., 2019].

At a more abstract level of description, the backpropagation algorithm of conventional deep learning allows to train neural networks by stochastic gradient descent (SGD). In this thesis, we have presented a mathematical framework which allows to preserve the key benefits of SGD, but opens a path for implementation on neuromorphic processors which directly exploit physics and the in-memory computing concept to perform the desired computations. Building neuromorphic systems that can match the performance of current deep learning systems is still in the future, but the potential speedup and power reduction is extremely appealing. This would also allow us to scale neural networks to sizes beyond the reach of current GPU-based deep learning models. Besides, by mimicking the working mechanisms of the brain more closely, such neuromorphic systems could also inform neuroscience.

7.2. Implications for Neuroscience

How do the biophysical mechanisms of neural computation give rise to intelligence? Ultimately, if we want to explain how our thoughts, memories and behaviours emerge from neural activities, we need a mathematical theory. Here, we explain how the mathematical framework presented in this thesis may help for this purpose.

7.2.1. Variational Formulations of Neural Computation ?

A number of ideas at the core of today's deep learning systems draw inspiration from the brain. However, these deep neural networks are not biologically realistic in details. In particular, the neuron models may look overly simplistic from a neurophysiological point of view. In these models, the state of a neuron is described by a single number, which can be thought of as its firing rate. A real neuron on the other hand, like any other biological cell, is an extraordinarily complex machinery, composed of a very large quantity of proteins interacting in complex ways. Because of this complexity, the hope to ever come up with a mathematical theory of the brain may seem vain.

This complexity should not discourage us, however. One key point is that not all details of neurobiology may be relevant to explain the fundamental working mechanisms of the brain that give rise to emerging properties such as memory and learning. Hertz [1991] puts it in these words: "Just as most of the details of the separate parts of a large ship are unimportant in understanding the behaviour of the ship (e.g. that it floats or transport cargo), so many details of single nerve cells may be unimportant in understanding the collective behaviour of a network of cells". Which biophysical characteristics of neural computation are essential to explain how information is processed in brains, and which can be abstracted away? While current deep learning systems use *rate models* (i.e. neuron models relying on the neuron's firing rate), a simple but more realistic neuron model is the leaky-integrate and fire (LIF) model, which accounts for the *spikes* (a.k.a. *action potentials*) and the electrical activity of neurons at each point in time. A more elaborated model is the Hodgkin-Huxley model of action potentials, which takes into account ion channels to describe how spikes are initiated. At a more detailed level, real neurons have a spatial layout, and each part of the neuron has its own voltage value and ion concentration values. In recent years, more realistic neuron models that include spikes [Zenke and Ganguli, 2017, Payeur et al., 2020] and multiple compartments [Bengio et al., 2016, Guerguiev et al., 2017, Sacramento et al., 2018, Richards and Lillicrap, 2019, Payeur et al., 2020] have been proposed for deep learning. Can we figure out which elements of neurobiology are essential to explain the mechanisms underlying intelligence, abstracting out those that are not necessary to understand these mechanisms?

In this thesis, we have presented a mathematical theory which applies to a broad class of systems whose state or dynamics is the solution of a variational equation. Given the predominance of variational principles in physics, a question arises: can neural dynamics in the brain be derived from variational principles too? We note that various variational principles for neuroscience modelling have been proposed [Friston, 2010, Betti et al., 2019, Dold et al., 2019, Kendall, 2021].

7.2.2. SGD Hypothesis of Learning

Today, the neural networks of conventional deep learning are trained by stochastic gradients descent (SGD), using the backpropagation algorithm to compute the loss gradients. The backpropagation algorithm is not biologically realistic as it requires that neurons emit two quite different types of signals: an activation signal in the forward pass, and a signed gradient signal in the backward pass. Real neurons on the other hand communicate with only one sort of signals – the *spikes*. Worse, the backpropagation through time (BPTT) algorithm used in recurrent networks requires storing past hidden states of the neurons.

Although these deep neural networks are not biologically realistic in details, they have proved to be valuable not just for AI applications, but also as models for neuroscience. In recent years, deep learning models have been used for neuroscience modelling of the visual and auditory cortex. Deep neural networks have been found to outperform other biologically plausible models at matching neural representations in the visual cortex [Mante et al., 2013, Cadieu et al., 2014, Kriegeskorte, 2015, Sussillo et al., 2015, Yamins and DiCarlo, 2016, Pandarinath et al., 2018] and at predicting auditory neuron responses [Kell et al., 2018]. Because SGD-optimized neural networks are state-of-the-art at solving a variety of tasks in AI, and also state-of-the-art models at predicting neocortical representations, a hypothesis emerges which is that the cortex may possess general purpose learning algorithms that implement SGD. More generally, a view emerges, which is that the fundamental principles of current deep learning systems may provide a useful theoretical framework for gaining insight into the principles of neural computation [Richards et al., 2019].

While the backpropagation algorithm is not biologically realistic, a more reasonable hypothesis is that the brain uses a different mechanism to compute the loss gradients required to perform SGD. A long standing idea is that the loss gradients may be encoded in the difference of neural activities to drive synaptic changes [Hinton and McClelland, 1988, Lillicrap et al., 2020]. If variational principles for neural dynamics exist, and if their corresponding energy function or Lagrangian function have the sum-separability property, then EqProp would suggest a learning mechanism involving local learning rules and suitable with optimization by SGD. Whereas in the setting of energy-based models, EqProp suggests that gradients are encoded in the difference of *neural activities* (as hypothesized by Lillicrap et al. [2020]), in

the Lagrangian-based setting, EqProp suggests that gradients are encoded in the difference of *neural trajectories*.

We note that the SGD hypothesis of learning also raises several questions. First, what is the loss function that is optimized? Unlike in conventional machine learning, there are likely a variety of such loss functions, which may vary across brain areas and time [Marblestone et al., 2016]. Second, SGD dynamics depend on the metric that we choose for the space of synaptic weights [Surace et al., 2020]. Also, while the SGD hypothesis is reasonable for the function of the cortex, other components of the brain such as the hippocampus may use different learning algorithms.

7.2.3. The Role of Evolution

In this manuscript we have emphasized the importance of learning. The ability for individuals to learn within their lifetime is indeed an essential component of intelligence. But learning alone is not the only key to human and animal intelligence. Far from being a blank slate, at birth, the brain is pre-wired and structured. This structure provides us straight from birth with innate intuitions, abilities, and mechanisms which make us predisposed to learn much more quickly [Dehaene, 2020, Chapters 3 and 4]. These innate structures and mechanisms have arisen through evolution. Machine learning models account for these innate aspects of intelligence using *inductive biases* (or *priors*). Traditionally, these inductive biases are manually crafted. However, given the complexity of the brain, one may wonder whether one will ever manage to reverse-engineer the inductive biases of the brain 'by hand'.

Evolution by natural selection can be regarded as another optimization process where, loosely speaking, the 'adjustable parameters' are the *genes*, and the 'objective' that is maximized is the *fitness* of the individual. The human genome has around 3×10^9 'parameters' (base pairs). Just like moving from manually crafted computations (in classical AI) to learned computations (in machine learning) proved extremely fruitful both for AI and neuroscience modelling, one may benefit from 'evolving' inductive biases, by mimicking the process of evolution in some way. One branch of machine learning which is relevant to address questions related to the optimization process carried out by evolution is *meta-learning* (Section 6.3). A related path, proposed by Zador [2019], is to reverse-engineer the program encoded in the genome which wires up the brain during embryonic development.

We note that the learning rules and loss functions of the brain have also arisen through evolution and are possibly much more complex than in the traditional view of machine learning (as we have formulated it in this manuscript).

7.3. Synergy Between Neuroscience and AI

Is a mathematical theory of the brain all we need to understand the brain? Or do we need to build brain-like machines to claim that we understand it? This question depends of course on what we mean by 'understanding' ; it is one of the fundamental questions of philosophy of science. In many fields of science, we have mathematical models of objects that we cannot build (for example, we have physics models of the Sun, but we cannot build one). Although a theory is all we need in principle to explain the measurements of experimentally accessible variables, it seems also clear that, if we can build a brain, or simulate one, our 'understanding' of the brain will further improve, and the underlying theory will become more plausible.

Can we simulate a brain in software? In the introductory chapter, we have argued that with current digital hardware this strategy would at best be extremely slow and power hungry, and more likely just unfeasible. Just like it is impossible for statistical physicists to simulate in software the internal dynamics of a fluid composed of 10^{23} particles, simulating a brain composed of 10^{11} neurons and 10^{15} synapses (and many many more proteins) seems unfeasible. In these respects, the development of appropriate neuromorphic systems will eventually be necessary to emulate a brain.

More likely, by making it possible to run and train neural networks with more elaborated neural dynamics that more closely mimic those of real neurons, the development of neuromorphic hardware will help us come up with new hypotheses about the working mechanisms of the brain. As we build more brain-like AI systems, and as the performance of these AI systems improves, we can formulate new mathematical theories of the brain. Just like the rise of deep learning as a leading approach to AI has eased the flow of information between different fields of AI (computer vision, speech recognition, natural language processing, etc.), we can expect that the development of neuromorphic systems together with mathematical frameworks to train them will ease the flow of information between AI and neuroscience too.

The problem of intelligence is thus both a problem for natural sciences and engineering. It counts to the greatest scientific problems, together with the problem of the origin of life, the problem of the origin of the universe, and many others. One specificity of the problem of intelligence is that, as we make progress towards solving this problem, we can use the knowledge that we acquire to build machines that can help us solve other scientific problems more easily. For example, most recently, a program called AlphaFold 2 promises to help us discover the 3D structure of proteins much more rapidly than prior methods, which is key to understanding most biological mechanisms in living organisms.

References

M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

D. H. Ackley, G. E. Hinton, and T. J. Sejnowski. A learning algorithm for boltzmann machines. *Cognitive science*, 9(1):147–169, 1985.

L. B. Almeida. A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. In *Proceedings, 1st First International Conference on Neural Networks*, volume 2, pages 609–618. IEEE, 1987.

S. Arora. Toward theoretical understanding of deep learning. *URL https://www. cs. princeton. edu/courses/archive/fall18/cos597G/lecnotes/lecture3. pdf*, 2018.

D. Attwell and S. B. Laughlin. An energy budget for signaling in the grey matter of the brain. *Journal of Cerebral Blood Flow & Metabolism*, 21(10):1133–1145, 2001.

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

P. Baldi and F. Pineda. Contrastive learning and neural oscillations. *Neural Computation*, 3(4):526–545, 1991.

Y. Bengio. *Learning deep architectures for AI*. Now Publishers Inc, 2009.

Y. Bengio, B. Scellier, O. Bilaniuk, J. Sacramento, and W. Senn. Feedforward initialization for fast inference of deep generative networks is biologically plausible. *arXiv preprint arXiv:1606.01651*, 2016.

Y. Bengio, T. Mesnard, A. Fischer, S. Zhang, and Y. Wu. Stdp-compatible approximation of backpropagation in an energy-based model. *Neural computation*, 29(3):555–577, 2017.

J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, pages 1–7. Austin, TX, 2010.

J. Bernstein, Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar. signsgd: Compressed optimisation for non-convex problems. *arXiv preprint arXiv:1802.04434*, 2018.

A. Betti, M. Gori, and S. Melacci. Cognitive action laws: the case of visual features. *IEEE transactions on neural networks and learning systems*, 31(3):938–949, 2019.

T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

G. W. Burr, R. M. Shelby, A. Sebastian, S. Kim, S. Kim, S. Sidler, K. Virwani, M. Ishii, P. Narayanan, A. Fumarola, et al. Neuromorphic computing using non-volatile memory. *Advances in Physics: X*, 2(1):89–124, 2017.

Cadence Design Systems, Inc. Spectre circuit simulator reference, version 19.1, Jan 2020.

C. F. Cadieu, H. Hong, D. L. Yamins, N. Pinto, D. Ardila, E. A. Solomon, N. J. Majaj, and J. J. DiCarlo. Deep neural networks rival the representation of primate it cortex for core visual object recognition. *PLoS Comput Biol*, 10(12):e1003963, 2014.

M. Campbell, A. J. Hoane Jr, and F.-h. Hsu. Deep blue. *Artificial intelligence*, 134(1-2): 57–83, 2002.

W. Chan, N. Jaitly, Q. Le, and O. Vinyals. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4960–4964. IEEE, 2016.

C.-C. Chang, P.-C. Chen, T. Chou, I.-T. Wang, B. Hudec, C.-C. Chang, C.-M. Tsai, T.-S. Chang, and T.-H. Hou. Mitigating asymmetric nonlinear weight update effects in hardware neural network based on analog resistive synapse. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 8(1):116–124, 2017.

K. Christianson and L. Erickson. The dirichlet problem on directed networks. 2007. URL `https://sites.math.washington.edu/~reu/papers/2007/KariLindsay/dirichlet.pdf`. Accessed: 2020-05-31.

L. Chua. Memristor-the missing circuit element. *IEEE Transactions on circuit theory*, 18 (5):507–519, 1971.

M. A. Cohen and S. Grossberg. Absolute stability of global pattern formation and parallel memory storage by competitive neural networks. *IEEE transactions on systems, man, and cybernetics*, (5):815–826, 1983.

S. Dehaene. *How We Learn: Why Brains Learn Better Than Any Machine... for Now*. Penguin, 2020.

D. Dold, A. F. Kungl, J. Sacramento, M. A. Petrovici, K. Schindler, J. Binas, Y. Bengio, and W. Senn. Lagrangian dynamics of dendritic microcircuits enables real-time backpropagation of errors. *target*, 100(1):2, 2019.

M. Ernoult, J. Grollier, D. Querlioz, Y. Bengio, and B. Scellier. Updates of equilibrium prop match gradients of backprop through time in an rnn with static input. In *Advances in Neural Information Processing Systems*, pages 7079–7089, 2019.

M. Ernoult, J. Grollier, D. Querlioz, Y. Bengio, and B. Scellier. Equilibrium propagation with continual weight updates. *arXiv preprint arXiv:2005.04168*, 2020.

R. P. Feynman. The principle of least action in quantum mechanics. In *Feynman's Thesis—A New Approach To Quantum Theory*, pages 1–69. World Scientific, 1942.

A. N. Foroushani, H. Assaf, F. H. Noshahr, Y. Savaria, and M. Sawan. Analog circuits to accelerate the relaxation process in the equilibrium propagation algorithm. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2020.

K. Friston. The free-energy principle: a unified brain theory? *Nature reviews neuroscience*, 11(2):127–138, 2010.

K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

J. Gammell, S. W. Nam, and A. N. McCaughan. Layer-skipping connections facilitate training of layered networks using equilibrium propagation. In *International Conference on Neuromorphic Systems 2020*, pages 1–4, 2020.

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

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

J. Guerguiev, T. P. Lillicrap, and B. A. Richards. Towards deep learning with segregated dendrites. *ELife*, 6:e22901, 2017.

J. Harris, C. Koch, J. Luo, and J. Wyatt. Resistive fuses: Analog hardware for detecting discontinuities in early vision. In *Analog VLSI implementation of neural systems*, pages 27–55. Springer, 1989.

K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages

770–778, 2016.

J. A. Hertz. *Introduction to the theory of neural computation*. CRC Press, 1991.

G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

G. E. Hinton and J. L. McClelland. Learning representations by recirculation. In *Neural information processing systems*, pages 358–366, 1988.

G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

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

J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

J. J. Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the national academy of sciences*, 81(10):3088–3092, 1984.

M. Hu, J. P. Strachan, Z. Li, E. M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, J. J. Yang, and R. S. Williams. Dot-product engine for neuromorphic computing: Programming 1t1m crossbar to accelerate matrix-vector multiplication. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2016.

J. Hutchinson, C. Koch, J. Luo, and C. Mead. Computing motion using analog and binary resistive networks. *Computer*, 21(3):52–63, 1988.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Z. Ji and W. Gross. Towards efficient on-chip learning using equilibrium propagation. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2020.

W. Johnson. Nonlinear electrical networks, 2010. URL `https://sites.math.washington.edu/~reu/papers/2017/willjohnson/directed-networks.pdf`. Accessed: 2020-05-31.

T. Karras, S. Laine, and T. Aila. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4401–4410, 2019.

A. J. Kell, D. L. Yamins, E. N. Shook, S. V. Norman-Haignere, and J. H. McDermott. A task-optimized neural network replicates human auditory behavior, predicts brain responses, and reveals a cortical processing hierarchy. *Neuron*, 98(3):630–644, 2018.

C. P. Kempes, D. Wolpert, Z. Cohen, and J. Pérez-Mercader. The thermodynamic efficiency of computations made in cells across the range of life. *Philosophical Transactions of the*

Royal Society A: Mathematical, Physical and Engineering Sciences, 375(2109):20160343, 2017.

J. Kendall. A gradient estimator for time-varying electrical networks with non-linear dissipation. *arXiv preprint arXiv:2103.05636*, 2021.

J. Kendall, R. Pantone, K. Manickavasagam, Y. Bengio, and B. Scellier. Training end-to-end analog neural networks with equilibrium propagation. *arXiv preprint arXiv:2006.01981*, 2020.

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

N. Kriegeskorte. Deep neural networks: a new framework for modeling biological vision and brain information processing. *Annual review of vision science*, 1:417–446, 2015.

A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. 2009.

A. Laborieux, M. Ernoult, B. Scellier, Y. Bengio, J. Grollier, and D. Querlioz. Scaling equilibrium propagation to deep convnets by drastically reducing its gradient estimator bias. *Frontiers in neuroscience*, 15:129, 2021.

C. Lanczos. *The variational principles of mechanics*. Courier Corporation, 1949.

Y. LeCun, D. Touresky, G. Hinton, and T. Sejnowski. A theoretical framework for backpropagation. In *Proceedings of the 1988 connectionist models summer school*, pages 21–28. CMU, Pittsburgh, Pa: Morgan Kaufmann, 1988.

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Y. LeCun, S. Chopra, R. Hadsell, M. Ranzato, and F. Huang. A tutorial on energy-based learning. *Predicting structured data*, 1(0), 2006.

Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436, 2015.

E. A. Lee. *Plato and the Nerd: The Creative Partnership of Humans and Technology*. MIT Press, 2017.

H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th annual international conference on machine learning*, pages 609–616, 2009.

T. P. Lillicrap, A. Santoro, L. Marris, C. J. Akerman, and G. Hinton. Backpropagation and the brain. *Nature Reviews Neuroscience*, pages 1–12, 2020.

V. Mante, D. Sussillo, K. V. Shenoy, and W. T. Newsome. Context-dependent computation by recurrent dynamics in prefrontal cortex. *nature*, 503(7474):78–84, 2013.

A. H. Marblestone, G. Wayne, and K. P. Kording. Toward an integration of deep learning and neuroscience. *Frontiers in computational neuroscience*, 10:94, 2016.

E. Martin, M. Ernoult, J. Laydevant, S. Li, D. Querlioz, T. Petrisor, and J. Grollier. Eqspike: Spike-driven equilibrium propagation for neuromorphic implementations. *arXiv preprint arXiv:2010.07859*, 2020.

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

C. Mead. Analog vlsi and neutral systems. *NASA STI/Recon Technical Report A*, 90, 1989.

W. Millar. Cxvi. some general theorems for non-linear systems possessing resistance. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 42(333): 1150–1160, 1951.

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

J. R. Movellan. Contrastive hebbian learning in the continuous hopfield model. In *Connectionist Models*, pages 10–17. Elsevier, 1991.

B. Muthuswamy and S. Banerjee. *Introduction to Nonlinear Circuits and Networks*. Springer, 2018.

A. Ng. Machine learning. Coursera, 2014. URL `https://sites.math.washington.edu/~reu/papers/2017/willjohnson/directed-networks.pdf`.

P. O'Connor, E. Gavves, and M. Welling. Initialized equilibrium propagation for backprop-free training. In *International Conference on Machine Learning, Workshop on Credit Assignment in Deep Learning and Deep Reinforcement Learning*, 2018. URL `https://ivi.fnwi.uva.nl/isis/publications/2018/OConnorICML2018`.

A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

R. C. O'Reilly. Biologically plausible error-driven learning using local activation differences: The generalized recirculation algorithm. *Neural computation*, 8(5):895–938, 1996.

J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

P. O'Connor, E. Gavves, and M. Welling. Training a spiking neural network with equilibrium propagation. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1516–1523, 2019.

C. Pandarinath, D. J. O'Shea, J. Collins, R. Jozefowicz, S. D. Stavisky, J. C. Kao, E. M. Trautmann, M. T. Kaufman, S. I. Ryu, L. R. Hochberg, et al. Inferring single-trial neural population dynamics using sequential auto-encoders. *Nature methods*, 15(10):805–815, 2018.

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

A. Payeur, J. Guerguiev, F. Zenke, B. Richards, and R. Naud. Burst-dependent synaptic plasticity can coordinate learning in hierarchical circuits. *bioRxiv*, 2020.

F. J. Pineda. Generalization of back-propagation to recurrent neural networks. *Physical review letters*, 59(19):2229, 1987.

T. Poggio, K. Kawaguchi, Q. Liao, B. Miranda, L. Rosasco, X. Boix, J. Hidary, and H. Mhaskar. Theory of deep learning iii: explaining the non-overfitting puzzle. *arXiv preprint arXiv:1801.00173*, 2017.

H. Ramsauer, B. Schäfl, J. Lehner, P. Seidl, M. Widrich, L. Gruber, M. Holzleitner, M. Pavlović, G. K. Sandve, V. Greiff, et al. Hopfield networks is all you need. *arXiv preprint arXiv:2008.02217*, 2020.

J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

B. A. Richards and T. P. Lillicrap. Dendritic solutions to the credit assignment problem. *Current opinion in neurobiology*, 54:28–36, 2019.

B. A. Richards, T. P. Lillicrap, P. Beaudoin, Y. Bengio, R. Bogacz, A. Christensen, C. Clopath, R. P. Costa, A. de Berker, S. Ganguli, et al. A deep learning framework for neuroscience. *Nature neuroscience*, 22(11):1761–1770, 2019.

F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

D. E. Rumelhart, G. E. Hinton, R. J. Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

J. Sacramento, R. P. Costa, Y. Bengio, and W. Senn. Dendritic cortical microcircuits approximate the backpropagation algorithm. In *Advances in Neural Information Processing Systems*, pages 8721–8732, 2018.

A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

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

F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

B. Scellier and Y. Bengio. Towards a biologically plausible backprop. *arXiv preprint arXiv:1602.05179v2*, 914, 2016.

B. Scellier and Y. Bengio. Equilibrium propagation: Bridging the gap between energy-based models and backpropagation. *Frontiers in computational neuroscience*, 11:24, 2017.

B. Scellier and Y. Bengio. Equivalence of equilibrium propagation and recurrent backpropagation. *Neural computation*, 31(2):312–329, 2019.

B. Scellier, A. Goyal, J. Binas, T. Mesnard, and Y. Bengio. Generalization of equilibrium propagation to vector field dynamics. *arXiv preprint arXiv:1808.04873*, 2018.

P. Sharma, N. Ding, S. Goodman, and R. Soricut. Conceptual captions: A cleaned, hypernymed, image alt-text dataset for automatic image captioning. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2556–2565, 2018.

D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

B. Speelpenning. Compiling fast partial derivatives of functions given by algorithms. Technical report, Illinois Univ., Urbana (USA). Dept. of Computer Science, 1980.

M. Stern, D. Hexner, J. W. Rocks, and A. J. Liu. Supervised learning in physical networks: From machine learning to learning machines. *arXiv preprint arXiv:2011.03861*, 2020.

E. Strubell, A. Ganesh, and A. McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019.

S. C. Surace, J.-P. Pfister, W. Gerstner, and J. Brea. On the choice of metric in gradient-based theories of brain function. *PLOS Computational Biology*, 16(4):e1007640, 2020.

D. Sussillo, M. M. Churchland, M. T. Kaufman, and K. V. Shenoy. A neural network that finds a naturalistic solution for the production of muscle activity. *Nature neuroscience*, 18 (7):1025–1033, 2015.

I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

T. Tieleman. Training restricted boltzmann machines using approximations to the likelihood gradient. In *Proceedings of the 25th international conference on Machine learning*, pages 1064–1071. ACM, 2008.

M. Tristany, S. Pequito, P. A. Santos, and M. A. Figueiredo. Equilibrium propagation for complete directed neural networks. *arXiv preprint arXiv:2006.08798*, 2020.

A. M. Turing. Computing machinery and intelligence. In *Parsing the Turing Test*, pages 23–65. Springer, 1950.

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

O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. M. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, et al. Alphastar: Mastering the real-time strategy game starcraft ii. *DeepMind blog*, page 2, 2019.

H. Vogt, M. Hendrix, and P. Nenzi. Ngspice (version 31), 2020. URL `http://ngspice.sourceforge.net/docs/ngspice-manual.pdf`.

Z. Wang, C. Li, W. Song, M. Rao, D. Belkin, Y. Li, P. Yan, H. Jiang, P. Lin, M. Hu, et al. Reinforcement learning with analogue memristor arrays. *Nature Electronics*, 2(3):115–124, 2019.

B. Widrow and M. E. Hoff. Adaptive switching circuits. Technical report, Stanford Univ Ca Stanford Electronics Labs, 1960.

Q. Xia and J. J. Yang. Memristive crossbar arrays for brain-inspired computing. *Nature materials*, 18(4):309–323, 2019.

D. L. Yamins and J. J. DiCarlo. Using goal-driven deep learning models to understand sensory cortex. *Nature neuroscience*, 19(3):356–365, 2016.

A. M. Zador. A critique of pure learning and what artificial neural networks can learn from animal brains. *Nature communications*, 10(1):1–7, 2019.

F. Zenke and S. Ganguli. Superspike: Supervised learning in multi-layer spiking neural networks. *arXiv preprint arXiv:1705.11146*, 2017.

G. Zoppo, F. Marrone, and F. Corinto. Equilibrium propagation for memristor-based recurrent neural networks. *Frontiers in Neuroscience*, 14:240, 2020.

N. Zucchet, S. Schug, J. von Oswald, D. Zhao, and J. Sacramento. A contrastive rule for meta-learning. *arXiv preprint arXiv:2104.01677*, 2021.

Appendix A

Gradient Estimators

Recall from Theorem 2.1 that the loss gradient is equal to

$$\frac{\partial \mathcal{L}}{\partial \theta}(\theta, x, y) = \left.\frac{d}{d\beta}\right|_{\beta=0} \frac{\partial E}{\partial \theta}\left(\theta, x, s_\star^\beta\right). \tag{A.1}$$

The *one-sided gradient estimator* is defined as

$$\widehat{\nabla}_\theta(\beta) = \frac{1}{\beta}\left(\frac{\partial E}{\partial \theta}(\theta, x, s_\star^\beta) - \frac{\partial E}{\partial \theta}(\theta, x, s_\star^0)\right), \tag{A.2}$$

and the *symmetric gradient estimator* is defined as

$$\widehat{\nabla}_\theta^{\mathrm{sym}}(\beta) = \frac{1}{2\beta}\left(\frac{\partial E}{\partial \theta}(\theta, x, s_\star^\beta) - \frac{\partial E}{\partial \theta}(\theta, x, s_\star^{-\beta})\right). \tag{A.3}$$

Lemma A.1. *Let θ, x and y be fixed. Assuming that the function $\beta \mapsto \frac{\partial E}{\partial \theta}(\theta, x, s_\star^\beta)$ is three times differentiable, we have, as $\beta \to 0$:*

$$\widehat{\nabla}_\theta(\beta) = \frac{\partial \mathcal{L}}{\partial \theta}(\theta, x, y) + \frac{A}{2}\beta + O(\beta^2), \tag{A.4}$$

$$\widehat{\nabla}_\theta^{\mathrm{sym}}(\beta) = \frac{\partial \mathcal{L}}{\partial \theta}(\theta, x, y) + O(\beta^2), \tag{A.5}$$

where $A = \left.\frac{d^2}{d\beta^2}\right|_{\beta=0} \frac{\partial E}{\partial \theta}(\theta, x, s_\star^\beta)$ is a constant (independent of β, but dependent on θ, x and y).

Lemma A.1 shows that the one-sided estimator $\widehat{\nabla}_\theta(\beta)$ possesses a first-order error term in β, which the symmetric estimator $\widehat{\nabla}_\theta^{\mathrm{sym}}(\beta)$ eliminates.

Proof. Define $f(\beta) = \frac{\partial E}{\partial \theta}(\theta, x, s_\star^\beta)$ and note that $f'(0) = \frac{\partial \mathcal{L}}{\partial \theta}(\theta, x, y)$ by Theorem 2.1, and that $f''(0) = \left.\frac{d^2}{d\beta^2}\right|_{\beta=0} \frac{\partial E}{\partial \theta}(\theta, x, s_\star^\beta)$. As $\beta \to 0$, we have the Taylor expansion $f(\beta) = f(0) + \beta\ f'(0) + \frac{\beta^2}{2} f''(0) + O(\beta^3)$. With these notations, the one-sided estimator reads $\widehat{\nabla}_\theta(\beta) = \frac{1}{\beta}\left(f(\beta) - f(0)\right) = f'(0) + \frac{\beta}{2} f''(0) + O(\beta^2)$, and the symmetric estimator, which is the mean of $\widehat{\nabla}_\theta(\beta)$ and $\widehat{\nabla}_\theta(-\beta)$, reads $\widehat{\nabla}_\theta^{\mathrm{sym}}(\beta) = f'(0) + O(\beta^2)$. □