Supervised learning in binary dynamical physical systems through energy minimization

Jérémie Laydevant

▶ To cite this version:

Jérémie Laydevant. Supervised learning in binary dynamical physical systems through energy minimization. Artificial Intelligence [cs.AI]. Université Paris-Saclay, 2022. English. NNT : 2022UPASP112 . tel-03956750

HAL Id: tel-03956750
https://theses.hal.science/tel-03956750

Submitted on 25 Jan 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

universite
PARIS-SACLAY

Supervised learning in binary dynamical physical systems through energy minimization

Apprentissage supervisé dans les systèmes physiques dynamiques binaires grâce à la minimisation de l'énergie

Thèse de doctorat de l'université Paris-Saclay

École doctorale n°564 : physique en Île-de-France (PIF)
Spécialité de doctorat: Physique
Graduate School : Physique, Référent : Faculté des sciences d'Orsay

Thèse préparée dans l'unité de recherche **UMPhy (Université Paris-Saclay, CNRS, Thales)**, sous la direction de **Julie GROLLIER**, Directrice de recherche

Thèse soutenue à Paris-Saclay, le 20 octobre 2022, par

Jérémie LAYDEVANT

Composition du jury

Jean-Michel PORTAL Professeur des Universités, IM2NP, Aix-Marseille Université	Président
Daniel BRUNNER Chargé de recherche, HDR, Institut FEMTO-ST	Rapporteur & Examinateur
Louis HUTIN Expert senior, CEA Leti	Rapporteur & Examinateur
Julie GROLLIER Directrice de recherche, UMPhy Université Paris-Saclay, CNRS, Thales	Directrice de thèse

THESE DE DOCTORAT

NNT : 2022UPASP112

ÉCOLE DOCTORALE

Physique en Île-de-France (PIF)

université PARIS-SACLAY

Titre: Apprentissage supervisé dans les systèmes physiques dynamiques binaires grâce à la minimisation de l'énergie

Mots clés: Apprentissage profond, Calcul neuromorphique, Réseaux de neurones binaires, Machine d'Ising

Résumé: L'apprentissage profond permet d'atteindre des performances jusqu'alors inaccessibles, que ce soit pour de la classification d'images, de la compréhension de parole ou de la génération de texte. Ces performances ont été permises par l'utilisation conjointe de réseaux de neurones profonds et d'un algorithme d'optimisation qui calcule le gradient d'une fonction de coût globale calculée à la sortie du réseau (rétropropagation du gradient) qui permet de faire émerger une hierarchie de détection des caractéristiques de la donnée d'entrée. Cependant, l'entrainement des réseaux de neurones, dont le nombre de paramètres augmente de façon dramatique, avec du matériel digital standard basé sur l'architecture Von Neumann est extrêmement inefficace d'un point de vue de la consommation énergétique puisque l'on doit continuellement transférer les valeurs des paramètres de la mémoire au processeur. Le calcul neuromorphique se pose en alternative basse consommation et rapide à ce matériel standard en proposant de rapprocher physiquement mémoire et éléments de calcul. Cependant, les implémentations matérielles manquent encore aujourd'hui d'algorithmes qui leur permettent d'atteindre les performances offertes par une optimisation basée sur le calcul du gradient d'une fonction de coût globale tout en évitant le surcoût énergétique dû aux circuits complexes qui réalisent la rétropropagation du gradient. Equilibrium Propagation (EP) est un algorithme d'apprentissage alternatif à la rétropropagation du gradient qui calcule le gradient d'une fonction de coût globale. EP permet de réaliser les deux phases d'apprentissage : la phase d'inférence et la phase de rétro-propagation des erreurs grâce à la propriété des systèmes physiques qui évoluent vers leur état le plus probable qui est aussi celui d'énergie minimale. La règle d'apprentissage prescrite par EP est locale et fait d'EP un bon candidat pour entrainer des implémentations matérielles neuromorphiques sans gros surcoût énergétique. En pratique, aucune implémentation matérielles grande échelle entraînée par EP n'a encore été démontrée. En effet, les dispositifs émergents envisagés pour les implémentations sont encore expérimentaux et souffrent donc d'une forte variabilité qui empêche l'entrainement sur puce. Dans cette thèse nous démontrons que nous pouvons entrainer avec EP un réseau de neurones artificiels dont les poids synaptiques et la fonction d'activation des neurones sont binarisés. Ceci permet d'envisager d'utiliser les dispositifs émergents dans un régime binaire qui réduit de façon considérable leur variabilité et autorise un apprentissage sur puce. Ces résultats peuvent également permettre de concevoir une puce digitale standard dédiée pour l'entrainement de réseaux de neurones binaires sur des dispositifs portables. Nous démontrons ensuite que nous pouvons appliquer EP à un système physique dont la fonction d'énergie est hautement paramétrisable: une Machine d'Ising (MI). La MI de DWave, par le biais de l'algorithme de recuit quantique, minimise successivement l'énergie des deux phases d'EP. Nous avons réussi à entrainer une architecture entièrement connectée sur la MI. Nous tirons également parti de la connectivité de la puce pour réaliser des convolutions et montrons que l'on peut entrainer un réseau convolutionel sur la MI avec EP. Ces travaux ouvrent la voie à l'entrainement supervisé sur puce de systèmes physiques non-conventionnels en s'affranchissant et de la nature expérimentale de nano-dispositifs envisagés pour des implémentations basse-consommation et de l'implémentation physique réelle puisque EP est adapté à entrainer des architectures de réseaux de neurones qui s'adaptent au matériel ciblé.

Title: Supervised learning in binary dynamical physical systems through energy minimization

Keywords: Deep learning, Neuromorphic computing, Binary neural networks, Ising Machine

Abstract: Deep learning makes it possible to achieve performances that were previously unattainable, whether for image classification, speech processing or text generation. These performances have been made possible by the joint use of deep neural networks and an optimization algorithm that calculates the gradient of a global cost function computed at the output of the network (backpropagation) which allows the emergence of a detection hierarchy of the input data characteristics. However, the training of neural networks, whose number of parameters is dramatically increasing, on standard digital hardware based on the Von Neumann architecture, is extremely inefficient from an electrical energy point of view since one must continuously transfer the values of the parameters from the memory to the processor. Neuromorphic computing is a low-power and fast alternative to this standard hardware by proposing to physically bring together memory and computing elements. However, hardware implementations still lack algorithms that allow them to reach the performances offered by an optimization based on the computation of the gradient of a global cost function that avoids the energy overhead caused by the complex circuits realizing backpropagation. Equilibrium Propagation (EP) is a learning algorithm that computes the gradient of a global cost function as an alternative to backpropagation. EP allows to realize the two learning phases: the inference phase and the backpropagation phase thanks to the property of physical systems that evolve towards their most probable state which is also the one of minimum energy. The learning rule prescribed by EP is local and makes EP a good candidate to train neuromorphic hardware implementations without large energy overhead. In practice, no large-scale hardware implementation trained by EP has been demonstrated yet. Indeed, the emerging devices considered for implementation are still experimental and therefore suffer from a high variability that prevents on-chip training. In this thesis we demonstrate that we can train with EP an artificial neural network whose synaptic weights and neural activation functions are binarized. This allows us to consider the use of emergent devices in a binary regime that considerably reduces their variability, compatible with on-chip learning. These results may also allow the design of standard digital chips dedicated to the training of binary neural networks on edge terminals. We then show that we can apply EP to a physical system that is energy-based by nature: an Ising Machine (IM). The DWave IM, through the quantum annealing algorithm, successively minimizes the energy of the two EP phases. We successfully demonstrate a training of a fully-connected architecture on the IM. We also take advantage of the chip layout to perform convolutions and show that we can train a convolutional network on the IM with EP. This work paves the way to supervised on-chip training of non-conventional physical systems by freeing itself from both the experimental nature of nano devices considered for low-power implementations and from the actual physical implementation since EP is adapted to train neural network architectures that fit the targeted hardware.

Contents

Remerciements **i**

Résume de la thèse **v**

Thesis summary **xi**

1 Conventional digital hardware performing deep learning with over-parametrized neural networks is energy-greedy **1**

1.1 Deep learning is about learning a non-linear mapping between an input to an output 2

1.1.1 A computation is a mapping of an input to an output, similarly to deep learning 2

1.1.2 The non-linear mapping of deep learning is done by an artificial neural network 3

1.1.3 A neural network learns the mapping by observing a lot of labelled data . . . 8

1.2 Neural networks are usually trained on standard digital hardware that are power-inefficient and thus exhibit limitations with the current large models 15

1.3 Increasing the power-efficiency of standard digital approaches 18

1.3.1 Better machine learning practices and open-source can reduce the energy and carbon footprint 18

1.3.2 Lighter models and more local learning algorithms improve energy-efficiency with less memory requirements 20

1.3.3 Specialized digital hardware improve energy efficiency with reduced data movement and massive parallelization 22

1.4 Summary: sota accuracy with gradient-based global optimization + energy efficiency with less data movement 25

2 Unconventional physics is a promising low-power alternative to conventional digital hardware for deep learning **27**

2.1 Leveraging physics to perform unconventional computing 28

2.1.1 Arranging emerging devices as a neural network-like hardware 29

2.1.2 Using a physical system on its own to perform the non-linear mapping 34

2.2 Leveraging the natural tendency of physical systems to evolve toward the ground state to parametrize the computation via the energy function 36

2.2.1 A statistical spin physics intuition for using the energy as the parametrizable feature of a physical system 36

2.2.2 The Ising Model: coupled spins that evolve collectively on a lattice 38

2.2.3 Hopfield networks: encoding patterns to-be-retrieved in the minima of the energy function 39

2.2.4 Boltzmann machines: tuning the energy function to approximate the data distribution and generate similar inputs 42

2.2.5 Conclusion 45

2.3 Summary 47

3 Introduction to Ising Machines, Equilibrium Propagation and Binary Neural Networks 49
3.1 Ising Machines: hardware minimizers of an energy function . . . 51
3.1.1 Solve an Ising Problem, solve them all . . . 51
3.1.2 Panorama of existing Ising Machines . . . 54
3.1.3 Existing Ising Machines . . . 57
3.2 Equilibrium Propagation: supervised learning with energy-based models . . . 62
3.2.1 Physical intuition behind Equilibrium Propagation . . . 62
3.2.2 Machine Learning description of EP . . . 67
3.2.3 Example: training a simple layered architecture . . . 72
3.2.4 EP has triggered great attention: benchmark . . . 73
3.3 Binary Neural Networks . . . 76
3.3.1 A historical introduction to Binary Neural Networks and definitions . . . 76
3.3.2 Challenges with BNNs: saturation and optimization . . . 79
3.3.3 BNNs have triggered great attention for custom hardware . . . 83
3.4 Summary . . . 84

4 Training Dynamical Binary Neural Networks with Equilibrium Propagation 87
4.1 Introduction . . . 87
4.2 EP Learning of Recurrent Binary Weights with Full Precision Neural Activations . . . 88
4.2.1 Feeding EP weight updates into BOP . . . 89
4.2.2 Normalizing the Binary Weights with a fixed scaling factor . . . 90
4.2.3 Normalizing the Binary Weights with a learnt scaling factor . . . 93
4.3 EP Learning of Recurrent Binary Weights with Binary Neural Activations . . . 95
4.3.1 Convergent neural networks with binary activations. . . . 95
4.3.2 Augmenting the Error Signal to Nudge Neurons with Binary Activations . . . 97
4.3.3 Results . . . 98
4.4 Related works . . . 102
4.5 Impact of Binary Equilibrium Propagation on hardware implementations . . . 102
4.5.1 Implementing in hardware the binary synapses and the optimization stage . . . 103
4.5.2 Hardware implementation of fully binarized Equilibrium Propagation . . . 105
4.5.3 Stochastic binary neurons? . . . 109
4.6 Conclusion . . . 111

5 Supervised learning in an Ising Machine with Equilibrium Propagation 113
5.1 Introduction: an Ising Machine as a parametrizable energy-based model for supervised learning: the D-Wave QPU . . . 113
5.1.1 Spin-based hardware are parametrizable energy-based models . . . 114
5.1.2 A specific case: the D-Wave IM . . . 115
5.2 Equilibrium Propagation allows supervised learning in a Ising Machine . . . 121
5.2.1 D-Wave IM can perform both the inference and the error-backpropagation through its intrinsic dynamics alone . . . 121
5.2.2 EP training algorithm with IM in the loop . . . 124
5.3 Training a fully-connected architecture on the D-Wave Ising Machine . . . 124
5.3.1 Embedding the Fully-Connected neural network on the Ising Machine and updating the parameters . . . 125
5.3.2 Results for a fully-connected architecture on MNIST . . . 126

5.4 Training a convolutional architecture on the IM: toward layout-guided architectures . 134
5.4.1 Why layout-guided architectures? . . . 135
5.4.2 Convolutional neural networks . . . 135
5.4.3 Local clusters of the DW2000 IM as a primitive convolution . . . 136
5.4.4 Integrate the cluster in a primitive convolutional neural network that fits the D-Wave layout . . . 138
5.4.5 Results . . . 140
5.5 Perspectives . . . 140
5.6 Conclusion . . . 142

6 Conclusion and perspectives **145**
6.1 Conclusion . . . 145
6.2 Perspectives . . . 146

Bibliography **148**

A Training Fully Connected Layers Networks with Equilibrium Propagation **165**
A.1 Energy-Based Settings . . . 165
A.1.1 Energy Function . . . 165
A.2 Prototypical Settings . . . 166
A.2.1 Dynamics . . . 166
A.2.2 Primitive Function . . . 167
A.2.3 Learning Rule . . . 167

B Training Convolutional Networks with Equilibrium Propagation **169**
B.1 Operations involved in the convolutional system . . . 169
B.2 Prototypical Settings . . . 170
B.2.1 Equations of the dynamics . . . 170
B.2.2 Learning rules . . . 170
B.3 Energy-Based Settings . . . 171
B.3.1 Equations of the Dynamics . . . 171
B.3.2 Learning Rules . . . 171

C A Scaling Factor for Equilibrium Propagation **173**
C.1 Learning the Scaling Factor with EP . . . 173
C.1.1 Learning Rules in the Prototypical settings . . . 173
C.1.2 Learning Rules in the Energy-Based Settings . . . 174
C.2 Simulations Details - Hyperparameters and Training Curves . . . 174
C.2.1 Binary Synapses . . . 174
C.2.2 Binary Synapses and Activations . . . 180

Remerciements

Je voudrais commencer par remercier Julie. Tu m'as fais confiance en me prenant sur un sujet qui n'était pas forcément ma spécialité. Je te remercie pour avoir été une super encadrante de thèse. Merci pour ton enthousiasme permanent, tes francs commentaires, ta patience et ta gentillesse. Merci d'avoir été si disponible, notamment en cette fin de thèse, accélerée par ma faute. Bravo pour ton intuition géniale qui veut concilier matériel et algorithme, et qui, comme on le pense, sera clé pour réduire l'empreinte carbone de l'intelligence artificielle dans le futur. Merci aussi de m'avoir laissé le choix des sujets sur lesquels je voulais travailler. Cela a toujours été un réel plaisir de travailler en sachant que tu nous fais confiance et nous soutient. Tu as su me faire évoluer sur un nombre infini de points, mais merci notamment pour l'amélioration de ma communication avec les autres ! Le groupe a de la chance de t'avoir.

Je veux aussi remercier chaleureusement Damien Querlioz, qui en plus d'avoir été membre de mon jury de thèse a été d'une immense aide tout au long de cette thèse. Merci pour la collaboration pour mon premier article et tes retours constructifs. Merci également pour tes commentaires pendant la deuxième partie de thèse et lors de la soutenance, ils font germer en moi plein d'idées !

Je tiens également à remercier Daniel Brunner, Louis Hutin, Jean-Michel Portal et Damien Querlioz d'avoir accepté d'être membre de mon jury de thèse. Merci pour votre intérêt pour le sujet et pour vos questions qui m'ont permis d'appréhender mon sujet et le calcul neuromorphique sous de nouveaux angles. Merci particulièrement à Daniel Brunner et Louis Hutin pour vos commentaires précieux et méticuleux sur mon manuscript.

Cette thèse n'aurait probablement pas pu avoir lieu sans Geoffroy Lerosey. Tu m'as fais confiance en me prenant comme « stagiaire mi-temps» à Greenerwave début 2018. Cette année et demi passée avec vous m'as permis de reconsidérer un parcours dans la recherche et tes conseils précieux m'ont finalement décidés à postuler à une thèse dont voici le résultat. Merci mille fois pour ta confiance et bravo pour la super aventure qu'est Greenerwave !

Merci à toute l'équipe neuromorphique qui m'a accompagné tout au long de cette thèse. Vous m'avez permis de grandir et murir scientifiquement, de découvrir de nouveaux horizons de recherche et d'obtenir des résultats. Merci notammenet à Dédalo d'avoir été un bon « parrain » de thèse, ton arrivée dans l'équipe le même jour que moi était certainement un signe puisque j'ai

pu compter sur tes (plus que) précieux conseils et commentaires pendant ces trois pour améliorer mes travaux. Tout en étant fort, tu restes humble et simple, tu es un exemple de chercheur ! Merci à Alice et Danijela pour leur encadrement. Merci à Jack (et Heidi) d'avoir été un super co-bureau et de nous avoir fait découvrir la culture chinoise/ australienne (nous nous souviendrons longtemps de vos dumplings maison !). Merci à Erwann pour les discussions et nos mises en commun de nos galères respectives avec EqProp. Merci à Marie d'avoir été une super co-thésarde dans l'équipe, ça a toujours été un plaisir d'échanger avec toi ! Merci à Pankaj, bien que nos sujets soient très distants, ça a toujours été agréable de discuter de nos travaux respectifs. Merci à Nathan, Erwan, Tristan, Arnaud, Dongshu, Andrew, Baptiste.

Je réserve une place particulière dans ces remerciements à Maxence Ernoult. Tu m'as initié à EqProp et à la rigueur nécessaire pour obtenir et analyser les résultats que nous avons obtenus ensemble. Ça a été un réel plaisir d'apprendre et d'échanger avec toi tout au long de cette thèse. C'est en partie grâce à toi que j'ai pu me faire une vision personnelle du sujet et du domaine, alors merci ! Je remercie aussi les membres du labo du C2N : Axel et Tifenn.

Merci aux personnes qui font vivre l'UMPhy. Merci à Vincent Cros et à Paolo Bortolotti de diriger ce labo. Merci à Anne Dussart et Florence Hamet pour votre aide au quotidien qui a été très précieuse ! Merci pour votre patience quand il a fallu attendre les factures.

Merci à Enzo, Kevin, Vincent, Sarah, Simon, qui en plus d'être des collègues de travail sont devenus au cours de ces trois années des amis précieux et sans doute indispensables pour la réussite de cette thèse. Bon courage Kevin et Sarah, la fin est proche ! Merci à toutes les autres personnes que j'ai cotoyé au sein de l'UMPhy, ça a été un réel plaisir de travailler à vos côtés.

Je tiens aussi à remercier tous mes amis hors laboratoire. Merci à Joris, qui me donne des leçons de ski depuis maintenant trop longtemps, d'être une bonne excuse pour rentrer à Annecy. Merci à mes amis de prépa : Diane, Eloi, Elodie, Erwan, Guillaume, Hugo, Mathéo, Pauline, Thomas et tout ceux que j'oublie. Merci aussi à mes amis de l'ESPCI : Loïc, Samuel, Jessie, Julie Z, Julie B, Alice, Chloé, Svetlana, Lucile. Merci aussi à Tristan. La distance et nos activités respectives ne permettent pas toujours des rencontres régulières mais vous avez une place particulière dans mon coeur.

Merci également à tout ceux que j'ai pu rencontrer à A Bras Ouverts. Merci à tous les merveilleux accompagnateurs (Adélaïde, Anaïs, ...) et jeunes porteur de handicap que j'ai pu rencontrer au sein du groupe Saint-Gobert. Être responsable de ce groupe pendant cette thèse a été un formidable moteur de motivation et remède au découragement.

Merci à toute ma famille pour votre présence permanente malgré la distance. Merci aux moments de partage et de rigolades qui ont été de vrais moments de détente pendant ces trois années. Merci plus particulièrement à ma soeur Laura, qui, malgré notre vision du monde complètement différente, a su me soutenir et me permettre d'être connecté au monde.

Merci papa et maman pour avoir toujours été mes premiers supporters. Vous avez été derrière moi depuis tout petit en me guidant et en acceptant tous mes choix. Merci de m'avoir suivi bien qu'étant étrangers au monde de la recherche. J'espère que cette thèse matérialise un peu plus ce que j'ai pu faire ces dernières années.

Enfin, merci Alice. Je ne sais pas comment te remercier assez pour ton soutien indefectible tout au long de cette thèse. Merci Alice d'avoir été la femme rêvée tout au long de ces trois années, de m'avoir supporté malgré mes sautes d'humeur, de m'avoir poussé à donner le meilleur de moi-même, merci ! Je remercie également notre petite Faustine, qui avec son sens tu timing légendaire est arrivée au moment de commencer la rédaction de thèse. Malgré les nuits blanches et une attention distraite, tes sourires et ta joie de vivre m'ont permis de finir cette thèse merveilleusement. J'ai hâte de voir la suite ensemble.

Synthèse

Contexte. Dans un monde où les données dictent la plupart des décisions, l'enjeu est aujourd'hui d'arriver à traiter ces données de la façon la plus efficace et performante possible. Les algorithmes de d'apprentissage automatique apprennent à découvrir des patterns sous-jacents à ces données en en regardant un très grand nombre. La branche la plus populaire de l'apprentissage automatique est actuellement l'apprentissage profond puisqu'il atteint les meilleures performances en reconnaissance d'image, génération de texte, etc [1]. Ce champ de l'intelligence artificielle repose sur l'utilisation d'algorithmes dits de "réseaux de neurones" artificiels qui apprennent à transformer des entrées de façon non-linéaire vers un espace des caractéristiques, par la présentation d'un grand nombre de ces entrées. L'apprentissage profond tire sa puissance de la façon dont le réseau de neurones est ajusté au fur et à mesure de ses observations pour affiner la transformation qu'il effectue sur les entrées. Une manière d'entrainer les réseaux de neurones artificiels est l'apprentissage supervisé. Cette procédure d'apprentissage requiert un ensemble de données d'entrainement dont les données sont étiquetées, l'étiquette correspondant à la classe de la donnée dans le tâche classique de classification d'images. Ainsi on peut calculer un signal de rétro-action pour mettre à jour les paramètres du réseau de neurones dans le but d'améliorer sa performance. En apprentissage supervisé, les ajustements, qui concernent les connections synaptiques entre neurones, sont appliqués de façon à minimiser une fonction de coût global qui décrit l'écart entre la sortie du réseau de neurones, obtenue pour une certaine donnée d'entrée, avec la sortie théorique attendue, qui est l'étiquette de cette même donnée. Combinée avec une architecture profonde, c'est à dire un réseau où plusieurs couches de neurones artificiels sont empilées, l'apprentissage supervisé, appliqué à la tâche de classification d'images par exemple, permet de faire émerger une hierarchie de détection des caractéristiques où chaque couche, en partant de la couche d'entrée, va détecter des caractéristiques de plus en plus complexes de l'entrée et permettre in-fine une séparation simple des entrées.

Récemment, l'explosion de la puissance de calcul disponible a rendu possible l'entrainement de réseaux de neurones artificiels ayant un nombre de paramètres gigantesque : Dall-E 2: 3B [2], GPT-3: 175B [3], Chinchilla [4], etc. Ceci a permis d'atteindre des performances qui surpassent largement toutes les méthodes existantes. Cependant, l'entrainement de ces très gros modèles s'accompagne d'une augmentation exponentielle de la consomma-

tion d'énergie requise pour faire tourner le matériel numérique utilisé pour entrainer les réseaux de neurones. Ces matériels sont basés sur l'architecture "Von Neumann" [5] où la mémoire et l'unité de calcul sont séparés physiquement. C'est une limitation majeure pour entrainer des réseaux de neurones, car la phase d'entrainement requiert de constamment bouger les données de la mémoire au processeur et inversement [6]. De plus, comme l'énergie nécessaire pour déplacer les données dépasse largement l'énergie nécessaire pour effectuer un calcul avec ces mêmes données [7], cela se traduit par une facture énergétique et des émissions de carbone pour entrainer des réseaux de neurones qui deviennent insoutenables. Ainsi, le besoin d'alternatives aux processeurs actuels est pressant.

Motivation de la thèse. Les réseaux de neurones artificiels peuvent être interprétés comme la transformation non-linéaire d'une donnée d'entrée vers un état de sortie. Alors on peut envisager d'utiliser des systèmes physiques, qui réalisent de façon intrinsèque cette transformation non-linéaire entre deux quantités physiques mesurables, pour effectuer cette transformation en lieu et place des processeurs numériques actuels. Tirer partie de la dynamique intrinsèque d'un système physique pour effectuer implicitement des opérations au lieu de calculer explicitement les opérations impliquées dans les réseaux neuronaux dans des simulations logicielles pourrait en effet conduire à des gains d'efficacité énergétique de plusieurs ordres de grandeur par rapport aux approches CMOS digitales traditionnelles [8]. Ces gains sont obtenus grâce à l'absence de déplacement de données et au calcul implicite effectué par la dynamique du système. Cependant, il est nécessaire de pouvoir paramétrer ces systèmes, c'est-à-dire de pouvoir appliquer des paramètres que l'on peut corriger et qui modifient la dynamique du système, afin de guider la dynamique vers la transformation souhaitée pour résoudre une tâche donnée.

L'une des possibilités pour paramétrer un système composé de nombreuses unités qui intéragissent entre elles, comme un réseaux neuronaux artificiels où les neurones sont couplés les uns aux autres, est d'ajuster ces interactions qui définissent la fonction d'énergie du système physique. En effet, Boltzmann nous a appris que de tels systèmes physiques systèmes physiques évoluent de façon la plus probable vers l'état de moindre énergie : $p(s,\theta) \propto e^{-\beta E(s,\theta)}$. Donc, si nous sommes capables de façonner le paysage énergétique à l'aide des paramètres ajustables, en particulier capable d'encoder dans des minima de l'énergie une sortie spécifique, nous pouvons guider le système pour qu'il effectue une tâche d'intérêt. Cette approche a largement inspiré les travaux pionniers en IA qui ont utilisé cette paramétrisation du système via la fonction d'énergie: réseau de Hopfield [9], Mémoire Associative Bi-directionnelle [10], Machine de Boltzmann [11], etc. Mais ces algorithmes peinent aujourd'hui à atteindre les performances des modèles de

deep learning car leur procédure d'entrainement n'optimise pas une fonction de coût globale mais plutôt des objectifs intermédiaires [11] voir aucun objectif [12], [13]. Ceci ne permet pas l'émergence d'une hiérachie de détection des caractéristiques de l'entrée et donc une moins bonne performance au final.

Mais aujourd'hui, l'intérêt pour ces modèles regagne en intensité pour la conception de puces basse consommation basées sur la physique de systèmes émergents. Contrairement à backpropagation, qui calcule le gradient de la fonction de coût par rapport aux paramètres du réseau grâce à la règle des dérivées successives, qui est hautement délocalisée et complexe et donc nécessite un circuit périphérique lourd pour être calculée, les algorithmes qui entraînent les modèles basés sur l'énergie se basent sur des règles d'apprentissage qui sont spatialement locales. Il n'y a donc aucun déplacement de données et la complexité du calcul est réduite, ce qui, en fin de compte, réduit les besoins en mémoire et en capacité de calcul requis pour l'apprentissage de ces modèles basés sur l'énergie. Ainsi, les règles d'apprentissage locales utilisées pour entraîner les systèmes physiques, qui sont naturellement basés sur l'énergie, pourraient conduire à des implémentations de réseaux neuronaux matériels à très faible consommation.

Problématique. Equilibrium Propagation (EP) [14], est une alternative à la backpropagation pour calculer le gradient d'une fonction de coût globale par rapport aux paramètres d'un réseau de neurones basé sur une fonction d'énergie. EP est donc un algorithme apprentissage prometteur pour la réalisation de systèmes physiques matériels, car il permet d'effectuer un apprentissage supervisé avec des règles d'apprentissage locales.

Cependant, une implémentation matérielle d'EP se heurte aujourd'hui à la nature encore expérimentale des composants basse consommation envisagés pour des implémentations matérielles (memristors, nano-oscillateurs, MRAM, ...) qui engendre une grande variabilité inter-composants [15] ainsi qu'un bruit intrinsèque [16] qui altèrent l'entrainement de tels systèmes.

Contributions de la thèse.

- Ainsi, dans cette thèse nous allons tout d'abord nous intéresser à réduire la précision requise pour les paramètres et les neurones afin que de tels systèmes basés sur une fonction d'énergie et entrainés par EP parviennent toujours à résoudre des tâches de classification d'image. Plus précisément, nous allons voir que l'on peut binariser les synapses et l'activation des neurones dans de tels réseaux et réussir à faire de la classification avec une précision à l'état de l'art. Ce travail permet d'envisager un entrainement sur puce par EP d'un système matériel fait

de composants émergents dans leur régime présentant le moins de variabilité et de bruit possible: le régime binaire, tout en conservant leur intérêt basse-consommation. Ce travail ouvre également l'opportunité d'entrainer des réseaux de neurones binaires (BNNs) sur du matériel numérique standard qui possède à la fois peu de mémoire et peu de capacité de calcul sans entrainement préalable sur un gros matériel (GPU), ce qui était impossible jusqu'à aujourd'hui.

Publication : Ce travail a fait l'object d'une publication *"Training Dynamical Binary Neural Networks with Equilibrium Propagation, Laydevant et al., 2021"* [17] dans le cadre du workshop Binary Vision au sein de la conférence CVPR 2021.

- Ensuite nous verrons comment nous avons réussi une première implémentation matérielle grande échelle entrainée par EP. Nous avons choisi comme système matériel la machine d'Ising proposée par D'Wave car c'est un système physique qui, par nature, implémente une fontion d'énergie, et qui, de plus, est facilement et rapidement reconfigurable, propriété indispensable pour les réseaux de neurones artificiels dont les paramètres sont continuellement ajustés lors de la phase d'entrainement. Nous allons voir que nous avons réussi à entrainer une petite architecture entièrement connectée sur la base de donnée de chiffres manuscrits MNIST en utilisant la procédure dite "d'embedding", qui assigne à un neurone plusieurs spins sur la puce afin de s'affranchir du problème de connectivité locale seulement offert par la puce. Ensuite nous verrons qu'on peut utiliser la connectivité inhérente au circuit de D'Wave pour faire des opérations plus complexes adaptées au matériel comme des convolutions et ainsi réussir à entrainer un petit réseau convolutionnel sur la puce. Ce travail ouvre la voie à d'autres implémentations matérielles entrainées par EP mais aussi à la possibilité d'utiliser les machines pour d'autres applications que la résolution de problèmes d'optimisation combinatorielle. Nous discuterons enfin du contexte plus global dans lequel s'inscrit cette thématique de recherche focalisée sur la réduction de l'énergie liée à l'entrainement et à l'utilisation des réseaux de neurones en tant que modèles d'inférence ainsi que des perspectives pour ces puces non-conventionnelles.

Publication : Ce travail va être l'objet d'une publication *"Supervised learning in an Ising Machine with Equilibrium Propagation, Laydevant et al."* dans un journal à comité de lecture et est actuellement en cours d'écriture.

Pour résumer, cette thèse s'intéresse à démontrer qu'un système physique non-conventionnel peut être une alternative pertinente aux processeurs clas-

siques pour faire de l'apprentissage profond. Tout au long de ce manuscript nous allons montrer au lecteur qu'utiliser le comportement physique de systèmes ext une voie prometteuse afin de combler les déficiences des approches tradionnelles digitales CMOS pour réaliser les calculs du deep learning.

Thesis summary

Context. In a data-driven world, the challenge today is to process this data in the most efficient and effective way possible. Machine learning algorithms learn to discover the underlying patterns in data by looking at a very large amount of it. The most popular branch of machine learning is currently deep learning because it achieves the best performance on image recognition, text generation, etc [1]. This field of artificial intelligence is based on the use of algorithms called "artificial neural networks" that learn on a large number of examples how to transform inputs in a non-linear way to a space of features. Deep learning gets its power from the way the neural network is adjusted as it observes more and more data to refine the transformation it performs on them. A successful way to train artificial neural networks is the supervised learning way. This learning procedure require a training dataset which inputs are labeled, the label corresponding to the input data category in the standard case of classification, such as we can compute a feedback to update the network parameters in order to improve its performance. In supervised learning, the adjustments, which concern the synaptic connections between neurons, are applied in order to decrease a global cost function that depicts the discrepancy between the output of the neural network, obtained for a specific input and the expected theoretical output, the label of this same input. Combined with a deep architecture, *i.e.* a network where several layers of artificial neurons are stacked, supervised learning, applied to image classification for instance, allows the emergence of a hierarchy of feature detection where each layer, starting from the input layer, will detect more and more complex features of the input and allows in-fine a simple separation of the inputs.

Recently, the explosion of the available computing power has made possible to train artificial neural network models that have a gigantic number of parameters: Dall-E 2 [2]: 3B, GPT-3 [3]: 175B, Chinchilla [4]: 70B, etc. This has made it possible to achieve performances that far exceed all existing methods. However, the training of these very large models is accompanied by an exponential increase in the power consumption required to run the digital hardware that is used to train the neural networks. These hardware are based on the "Von Neumann" architecture [5] where the memory and the processing unit are physically separated. This is a major limitation to train neural networks as we constantly need to move the parameters back and forth between the memory and the processing unit [6]. Furthermore, as the energy to move data largely excels the energy to do an actual computation

with the data [7], this results in a energy bill and carbon emissions to train neural networks that become unsustainable. Thus, the need for alternatives to the current processors is pressing.

Motivation of the thesis. Artificial neural network algorithms can be interpreted as a non-linear mapping from an input to an output in a non-linear way. Then we can consider using physical systems which intrinsically carry out this non-linear transformation between two measurable physical quantities instead of the current digital processors. Leveraging the intrinsic dynamics of a physical system to perform the computation instead of computing the operations involved in neural networks in software simulations could lead to orders of magnitude gains in energy-efficiency compared to traditional digital CMOS approaches [8]. These gains are obtained because of no data movement and the implicit computation performed through the dynamics of the system. However, it is necessary to be able to parameterize these systems, *i.e*. apply tunable parameter that modify the dynamics of the system, in order to guide the dynamics toward the desired mapping solving a given task.

One possibility for parametrizing a system made of many interacting units, such as an artificial neural networks where the neurons are coupled to each other, is to tune the interactions that define the energy function of the physical system. Indeed, Boltzmann taught us that such physical systems most likely evolve towards the state of least energy: $p(s, \theta) \propto e^{-\beta E(s,\theta)}$. So if we are able to shape the energy landscape with the help of the tunable parameters, especially encode in minima of the energy a specific output, we can guide the system to perform a task of interest. This approach has largely inspired the pioneering works in AI that have used this parametrization of the system via the energy function: Hopfield network [9], Bi-directionnal Associative Memories [10], Boltzmann machine [11], etc. But these algorithms struggle today to reach the performances of deep learning models because their training procedure does not optimize a global cost function but rather intermediate objectives [11] or no objectives at all [12], [13]. This does not allow the emergence of a hierarchy of input features detection and therefore leads to a lesser performance in the end.

But today, the interest for these models is regaining intensity for the design of low power chips based on the physics of emerging systems. Contrarily to backpropagation, which computes the gradient of the cost function with respect to the parameters with the chain-rule derivation which is highly delocalized and complex hence necessitates a heavy peripherical circuit to be computed, algorithms that train energy-based models build on learning rules that are spatially local. This results in no data movement and a reduced computational complexity which, ultimately, reduce both the memory and

computation requirements for training these energy-based models. Hence, local learning rules used to train physical systems, that are naturally energy-based, could lead to ultra low-power hardware neural networks.

Problematic. Equilibrium Propagation (EP) [14], is an alternative to back-propagation to compute the gradient of a global cost function with respect to the parameters of an energy-based neural network. EP is thus a promising learning framework for designing hardware physical systems as it does supervised learning with local learning rules.

However, a hardware implementation of EP is currently hampered by the still experimental nature of the low-power nanodevices considered for hardware implementations (memristors, nano-oscillators, magnetic random access memories, ...) which results in a large device-to-device variability [15] as well as an intrinsic noise that impairs the training of such systems [16].

Contributions of the thesis.

- Thus, in this thesis we will first focus on reducing the required precision of parameters and neurons so that such energy-based systems and trained by EP can still solve image classification tasks with high accuracy. More precisely, we will see that one can binarize synapses and neuron activations in such networks and succeed in classification at state-of-the-art levels. This work allows us to consider an on-chip training by EP of a hardware made of emerging components in their regime presenting the least variability and noise possible: the binary regime, while keeping their low-consumption interest. This work also opens the opportunity to train binary neural networks (BNNs) [18], [19] on standard digital hardware that has little memory and computational budgets without prior training on large hardware, which was impossible until now.

  Publication: : This work has been published as *Training Dynamical Binary Neural Networks with Equilibrium Propagation, Laydevant et al., 2021"* [17] within the workshop Binary Vision at the CVPR 2021 conference.

- Then we will describe how we realized a first large-scale hardware implementation driven by EP. We chose the Ising machine proposed by DWave as our hardware system for this purpose because it is an physical system which by nature implements an energy function and, moreover, it is easily and quickly reconfigurable, an indispensable property for artificial neural networks whose parameters are continuously adjusted during the training phase. We will see that we have succeeded in

training a small fully-connected architecture on the handwritten digits dataset MNIST using the so-called "embedding" procedure, which assigns to a neuron several spins on the chip in order to overcome with the local-only connectivity offered by the chip. Then we will see that we can use the inherent connectivity of the D'Wave circuit to perform more complex operations adapted to the hardware like convolutions and thus train a small convolutional network on the chip. This work opens the way to other hardware implementations driven by EP but also to the possibility of using Ising machines for other applications than solving combinatorial optimization problems. Finally, we will discuss the global context in which this research theme is developed, with a focus on the reduction of energy related to training and the use of neural networks as inference models as well as the perspectives for these unconventional chips.

Publication: : This work will be the subject of a publication "Supervised Learning in an Ising Machine with Equilibrium Propagation, Laydevant et al." in a peer-reviewed journal and is currently being written.

To summarize, this thesis is interested in demonstrating that an unconventional physical system can be a relevant alternative to classical processors for deep learning. Throughout this manuscript we will show the reader that using the physical behavior of systems to perform deep learning computations offers new routes to overcome the deficiencies of traditional digital CMOS approaches for deep learning computations.

CHAPTER 1

Conventional digital hardware performing deep learning with over-parametrized neural networks is energy-greedy

The automation of tasks such as image classification, text translation or motion control has recently become widespread. This adoption has been enabled by deep learning [1] which is a branch of artificial intelligence based on the intensive use of artificial neural networks that learn to solve tasks from a large amount of data. However, the methods and hardware currently used to make these networks learn are not optimized, which questions the widespread adoption of deep learning without considering the related energy and ecological costs.

As an introduction to the subject of this thesis, we will first describe what deep learning is and how it works today. We will then see why its backbone (the backpropagation algorithm [20]) makes deep learning, as currently trained, difficult to reconcile with economical and ecological sobriety objectives. Then we will see that a possible alternative to the standard processors and architectures currently used to train deep neural networks can be found in the physics of other richer systems. Finally, we will discuss the challenges of designing such new processors based on the physics of emerging unconventional components for computing to reach the performance achieved on current CMOS processors.

We will start by going back to the definition of computation. This concept refers to how two states of a given physical systems are related to each other [21]. Starting from this principle, we will focus on the computations on which deep learning is based to highlight the type of operation that a deep learning algorithm performs: the transformation of an input data into a vector of interest that constitutes the output. We will also detail the algorithms that allow us to obtain the amazing results of deep learning. This will highlight the limits of current deep learning, whose trend is to increase the number of parameters and therefore the need for computing power, entailed on the current conventional hardware (CMOS) especially its high energy and economic costs. Based on this observation, we will explore two possible approaches that attempt to overcome these limitations.

The first method will be developed in this first chapter. It is based on standard deep learning methods but strives to improve practices, whether at the global system level, at the software level or at the hardware level. Thus, we will see how the advent of cloud computing and open source could be one of the solutions to the carbon emission problems of deep learning models. We will also describe the efforts made to reduce the size and computational requirements of trained models. Finally we will see the research done to improve the standard hardware specialized for deep learning computations, they will be optimized to perform the vector matrix product efficiently. We will dissociate the axes of improvement for hardware dedicated to server use and that dedicated to edge computing.

The second approach will be the one this thesis focuses on and will be introduced in the second chapter.

1.1 . Deep learning is about learning a non-linear mapping between

an input to an output

Answering the question "why using physics to compute?" requires first defining what computing means. Before diving into the computations that interest us in this thesis - those performed in the context of deep learning - we will go back to the concept of computation. The definition that we will give : "the mapping from an input to an output to perform tasks of interest", will then serve as a guideline throughout this introductory chapter

We will then see that this mapping from an input to an output is exactly what an artificial neural network, the basic algorithm of deep learning, does. We will then describe what an artificial neuron is and how it is integrated into an artificial neural network by being connected to other neurons through synaptic connections that modulate the propagation of the signal in the network.

Finally, we will see how these synaptic connections are adjusted so that the network performs the task we want it to do. We will introduce the concept of supervised learning by taking the example of the backpropagation algorithm, the most widespread learning algorithm today, which allows the minimization of a global cost function by stochastic gradient descent.

1.1.1 . A computation is a mapping of an input to an output, similarly to deep learning

A mathematical computation speaks to all of us in an implicit way. We know what an addition, a multiplication, etc. are. But in general, it is more difficult to define what a computation is. So we can then refer to the philosophers of science to help us to give this definition.

Gualtiero Piccinini's summary of this account states that a physical system can be said to perform a specific computation when there is a mapping between the state of that system and the computation such that the "microphysical states of the system mirror the state transitions between the computational states" [21].

Thus the concept of computation is intrinsically broad and encompasses many mathematical operations. But we also see that in his attempt to define the concept of computation, Piccinini brings into play the principle of relation between two states of a certain physical system. From then on, the term computation can be understood in different ways. The first one can be the fundamental transition between two states of a physical property of a system such as the on or off state of a transistor or the transition from a non-synchronized state to a synchronized state for an oscillator. The term physical system in the definition can also refer to a more complex physical system composed of different units interacting with each other. The relation between the two states is the "computation" and can be the temporal evolution of a dynamic system for example.

Figure 1.1: Computation can be interpreted as the transformation of an input to an output

In this thesis, we will focus on a very literal use of this definition by placing ourselves in a field of computer science that makes intensive use of this mapping between two states of a physical system. This field of computer science is called deep learning. This discipline is based on the so-called artificial neural network algorithms that perform a transformation (preferably non-linear) of input data in order to perform tasks of interest. These tasks are multiple and can be for example image classification, text generation, ...

In the case of standard deep learning, the physical system described in the definition of computation is in fact the neural network. In this thesis we will take this definition literally and apply it to the case of deep learning and see how a "real" physical system can, by its intrinsic dynamics, perform a computation on data by realizing this mapping between input data and a state of interest which is the output.

1.1.2 . The non-linear mapping of deep learning is done by an artificial neural network

Artificial neural networks are the basic algorithms on which deep learning is based. In this part, we detail the important steps that led to artificial neural networks as we know them today, and then introduce the formalism specific to these algorithms.

The concept of the artificial neuron was invented in 1943 by McCulloch and Pitts [22]. It was initially conceptualized to model biological neurons which, at that time, were seen as "all-or-nothing" binary units:

$$O_i = H(y) \tag{1.1}$$

where $H(y)$ is the Heaviside step function defined as:

$$H(y) = \begin{cases} 0 \; if \; y < 0 \\ 1 \; else \end{cases} \tag{1.2}$$

and y any input which is a stimulus for the neuron. The activation function of the formal neuron which is the Heaviside step function can be extended to continuous σ functions with a more progressive behavior between two extreme values like the sigmoid or hyperbolic tangent function.

McCulloch-Pitts were also the first to model the concept of artificial synapse by weighting the inputs of the neuron by a real coefficient W_{ij} (weight of the input j to the neuron i) whose sign and amplitude describe the importance of the input for the neuron. Thus, the formal McCulloch-Pitts neuron receives inputs $\{X_i\}$ which are weighted by synaptic weights W_{ij}, realizes the sum of them and then applies a nonlinear activation function which is the Heaviside function in this case:

$$O_i = H(\sum_j W_{ij} X_j) \tag{1.3}$$

The McCulloch-Pitts formal neuron solved logical tasks such as the logical OR[1] or the logical AND[2]. However, it has shown its limitations as an individual neuron since it does not solve the logical exclusive OR task[3] which requires several such neurons to solve. Furthermore, the synaptic weights used for the neuron are chosen and fixed to perform a specific task and do not exhibit plasticity - they have no possibility to change over time.

In 1957, Rosenblatt developed the perceptron [24] which is an algorithm for binary classification of input data that works on the principle of the formal neuron of McCulloch-Pitts, with the addition of a learning rule that allows the neuron to learn to classify the data presented to it. Thus, with the perceptron, the property of synaptic plasticity is introduced and will pave the way for artificial

[1]The logical OR is the logical function that given inputs X_1 and X_2, returns 1 as soon as one of the two inputs is 1, and this also in the case where both inputs are 1

[2]The logical AND is a logical function which, given inputs X_1 and X_2, returns 1 only when both inputs are 1.

[3]Logical exclusive OR is a logical function that given inputs X_1 and X_2, returns 1 only when only one of the two inputs is 1. Regardless of X_1 or X_2.

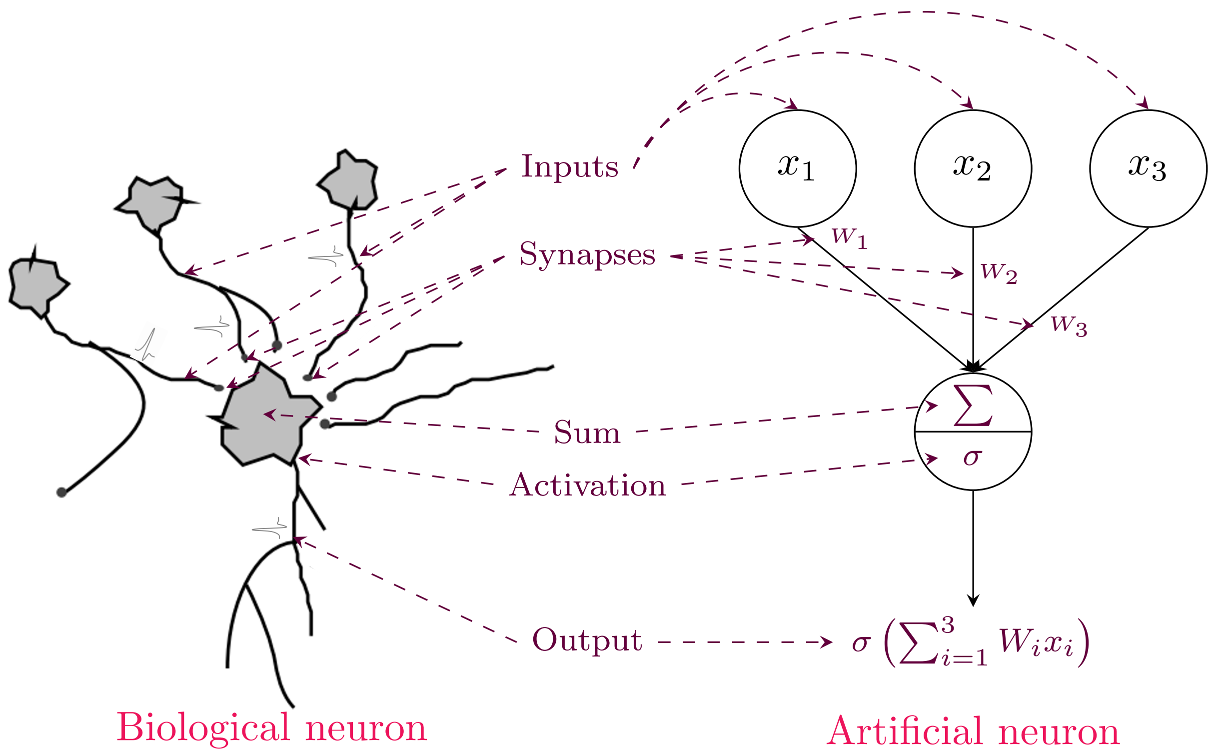


Figure 1.2: The building bricks of artificial neurons are inspired from biological neurons. We here show the more general case of the neuron of McCulloch-Pitts where σ can be a continuous activation function. Schematic of the biological neuron is from [23]

neural networks. To learn to classify the presented data, the perceptron requires a training database $\mathcal{D}$ which contains data pairs $\{(X^k, y^k)\}_{k\in\mathcal{D}}$ which correspond to data-label pairs. Thus we can know what is the expected theoretical output of the perceptron for a given input and compute a prediction error with the output obtained by the perceptron $\{O^k\}_{k\in\mathcal{D}}$. From this a learning rule is derived, which constitutes an indication of the direction and magnitude of the update of a parameter to reduce the prediction error or to improve the prediction success of the perceptron. This type of learning is called supervised learning and has since been used extensively to solve complex machine learning tasks.

The learning rule for the synaptic weights of a perceptron is as follows:

$$\Delta W_{ij} = X_j^k (O^k - y_i^k) \tag{1.4}$$

where the index i indexes the different inputs and the index k denotes the example k presented to the network as well as the state of the corresponding output neuron and the target associated with the input.

This learning rule inspired by Hebb's work [25] minimizes the Mean Squared Error cost function $C(O^k, y_i^k) = \frac{1}{2}\sum_i (O^k - y_i^k)^2$ which is computed at the output of the perceptron.

The perceptron was a founding step in learning in artificial neural networks. However, the perceptron is only applicable to binary classification tasks (the output of the neuron can only be 0 or 1). And like the formal neuron of McCulloch-Pitts, the perceptron can only solve linearly separable

tasks, which is almost never the case for most databases. Finally, in 1969 the book "Perceptron" [26] by Minsky and Papert, cast a shadow on the perceptron by pointing out its inability to solve the exclusive OR problem.

It took more than 10 years for multi-layer perceptrons to appear [27] and to finally solve the exclusive OR problem [27]. These networks are said to be "fully-connected" - that is, each neuron in one layer is connected to all neurons in the next layer via synaptic connections (Fig. 1.3). The synapses modulate the signal propagation of the previous layer in the same way as the inputs of a perceptron are modulated and in a unidirectional way and make the neurons sensitive to distinct features by combining the state of the previous layer in a different way. In fact, these different layers perform in several steps a non-linear transformation of the inputs to a space of features where the different classes are much more easily separable. The state of each layer is then calculated successively according to the state of the previous layer until the state of the output layer o is calculated, which allows to make a prediction on the class to which the presented input x belongs when the network is trained to perform classification.

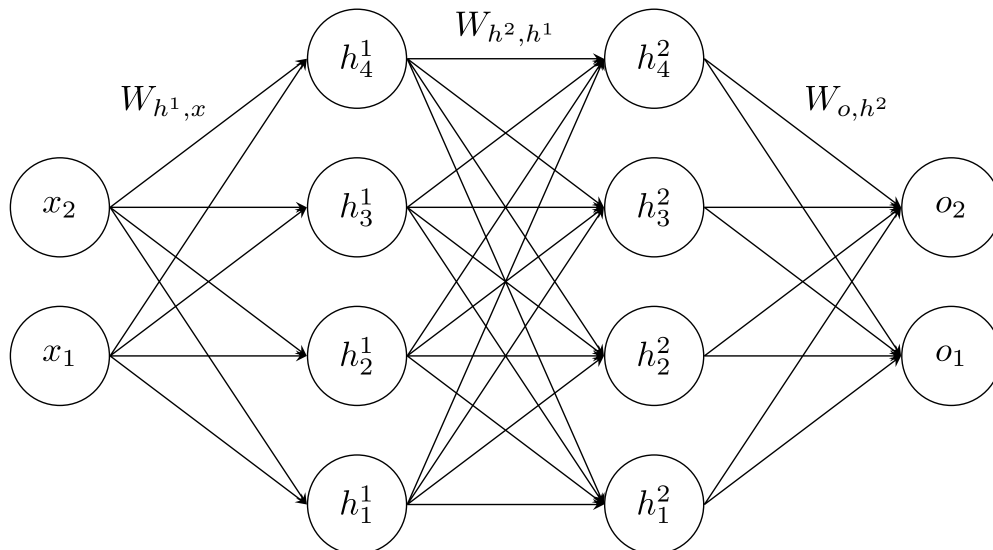
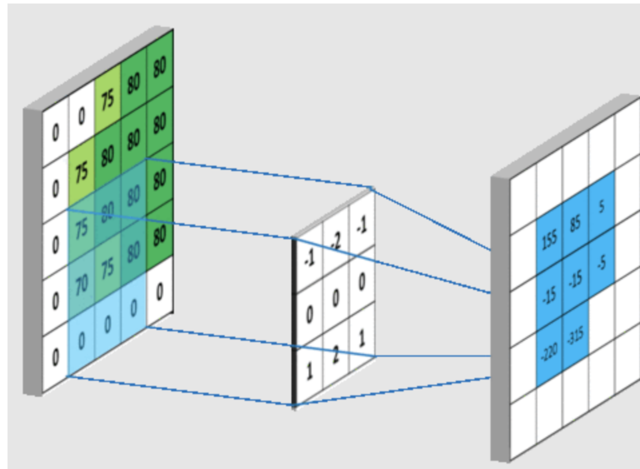


Figure 1.3: A multi-layered perceptron - or fully connected neural network. x units denote the input nodes, h^k units denotes the nodes of the $k-th$ hidden layer and o units denotes the output units.

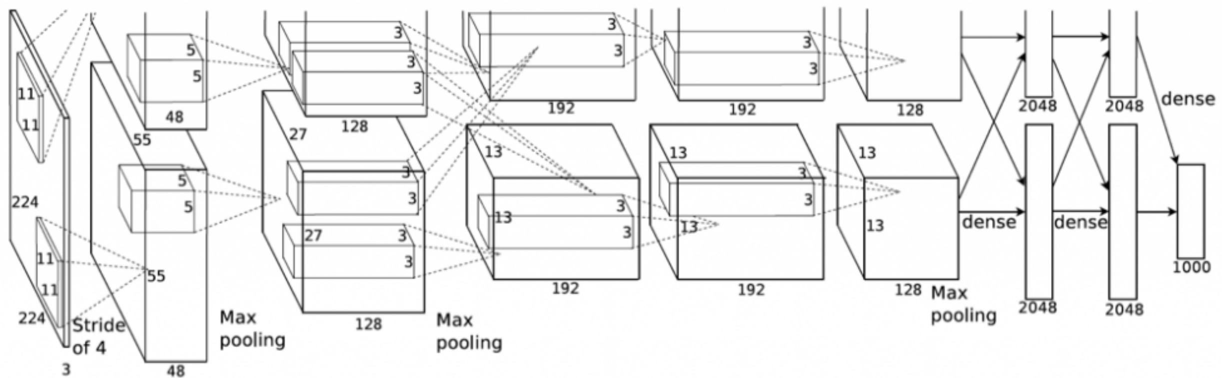
As already suggested by Minsky and Papert in "Perceptron" [26], Rumelhart, Hinton and Williams confirm in their founding paper [20] the importance of introducing neurons between the inputs and the output neurons to build complex representations of the input data that allow a progressive separation in the space of the representations and their classification at the output. This is how the concept of "deep network", with stacked layers of neurons, was born, and this is where deep learning gets its power from.

By inserting intermediate - hidden - neurons between the inputs and the output neurons, Rumelhart, Hinton and Williams [20] succeeded in solving the exclusive OR task by training the neural network in a supervised way using the backpropagation procedure. It is also mentioned in the article that the number of times the network must be presented with inputs to be trained decreases as the number of hidden neurons increases. This highlights the importance of hidden units to build a complex representation of the inputs that can be combined to the output layer to perform classification.

Finally, between 1986 and 2012, the main advance in the field is the use of convolutional operations instead of "fully-connected" connections in first layers in the so-called convolutional networks (CNNs) [28]. These convolutional operations allow to operate a detection by filtering the input images with filters which, during the process of optimization, will specialize to detect certain spatial features of an input image, or of its internal representations in the network. The first demonstration of the interest of these convolutional networks was to apply a CNN to the recognition of handwritten digits [28].



(a) Convolution operation between an input image and a (3x3) filter. The combination of weights in the filter allows to apply different filters to the input to activate according specific inputs. From [29]



(b) Architecture of the Alexnet CNN [30] that extensively used (11x11) filters and Max pooling operations that allow to downsample data.

Figure 1.4: Convolutional neural networks are based on a different operation between layers than a fully-connected operation. The hierarchy of features detected at the output emerges when convolutional layers are stacked.

Unfortunately, as we will see in the next section, the optimization algorithm to minimize the computed error at the output of the network, backpropagation, requires very large memory and computational resources that have been out of reach for a long time for very deep networks (with more than 2 convolutional layers such as the LeNet-5 network [28] that has been for a long time the

reference in the convolutional network field).

2012 is a pivotal year for the research field of artificial neural networks with the first backpropagation training of a deep network containing 5 convolutional layers followed by 2 fully-connected layers [30]. Instead of using standard processors rather specialized in sequential computation (Central Processing Unit (CPU)), Krizhevsky *et al*. had the idea to use graphic processors (GPU), very powerful in parallelized computation, to perform, in a fast and efficient way, the vector-matrix products repetitively realized in the neural networks. This unconventional use of the hardware available at the time allowed them to successfully train such a network with backpropagation. At the time, the authors had to write the operations involved in their network in low-level code. But today, the success of deep learning also comes from the fact that open-source high-level libraries allowing a great abstraction from low-level implications are available to any user: Pytorch, TensorFlow on Python are the best known. These libraries allow the user to train his neural network in an ultra simplified way with a command like $network.to(gpu)$. As we will see later, but they also do a lot for the dissemination of trained models, because they can be made available via these libraries as open-source models and loaded by the user in an extremely simple way.

By halving the classification error obtained previously obtained on Imagenet with other machine learning algorithms, Krizhevsky et al [30] demonstrated the importance of the depth of artificial neural networks to build powerful internal representations.

In fact, stacking successive layers in a globally optimized manner allows a hierarchy to emerge in the neural network's internal representation of the input data. The fact that successive layers of neurons are sensitive to increasingly complex features is called a detection hierarchy (Fig. 1.5). The first layer of a network applied to image recognition can for example recognize simple geometrical shapes: vertical, horizontal, diagonal lines, ... The next layer can be sensitive to combinations of geometric shapes recognized in the previous layer: square, circle, oval, ... The last layer before the classifier is sensitive to patterns that are close to the different classes to be classified in the database: heads, car, dog, cat, ... The classifier layer for a classification task like here corresponds to the last layer of the network which is fully-connected and combines the last features extracted from the image to assign membership scores for the input to the different classes in the database. Hierarchical feature detection allows for a more intelligent and sparse use of parameters where a feature can be recognized by a combination of several elementary filters.

The emergence of the detection hierarchy allows for better segregation and combination of useful features to make the right prediction at the network output. On the contrary, the learning that was mostly used before backpropagation and the optimization of a global objective was based on the optimization of intermediate objectives [11], [32] that were disconnected from each other, which did not allow a coherent hierarchy to emerge across the entire network - or in a much less powerful way [33], which drastically limited the potential of such approaches.

Since 2012, new approaches have made it possible to have even deeper networks and thus increase the accuracy obtained on ImageNet with the Residual neural networks (ResNets) which can have 18, 34, 50, 101 and even 152 hidden layers [34].

In this section, by quickly going through the history of artificial neural networks, we have discussed the two key concepts that make deep learning successful, which are the joint use of a stack of layers of neurons connected to each other by different operations, together with the minimization of a global cost function through gradient descent. These two methods have allowed deep learning to outperform previous methods by a large margin. We have also seen the interdependence of deep learning as an algorithm with the hardware on which the algorithms run by highlighting the turn taken in 2012 with the use of GPUs for fast highly parallelized computing.

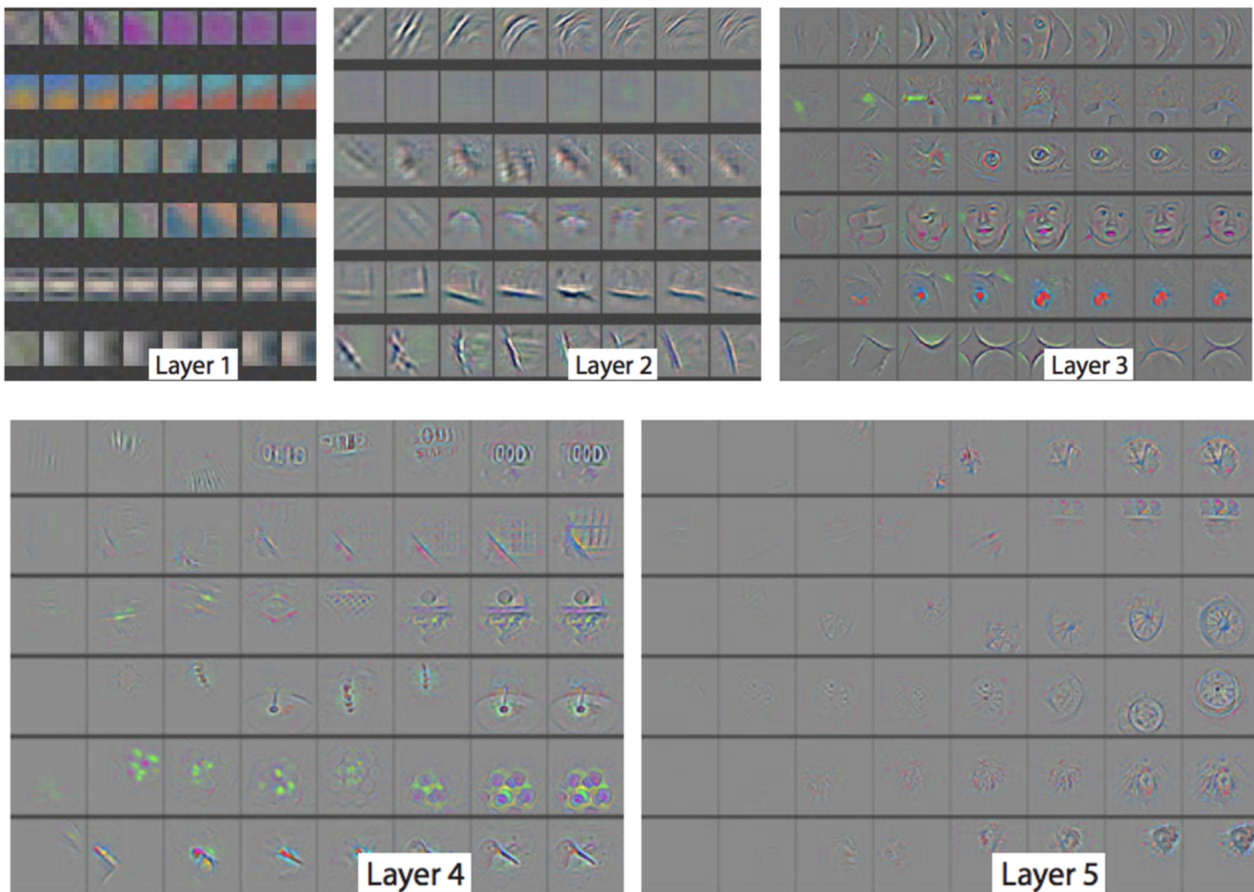


Figure 1.5: Visualization of the features detected on inputs images with varying depths: Layer 1 to 5 of Alexnet-like NN. We clearly see how the features detected are more and more fine-grained when we go deeper in the network. Inside each layer, from left to right is the evolution of the detected feature over training - filters become more and more sensitive to a specific feature. From [31]

In the next section we will dig in the concept of learning in artificial neural networks that we have just introduced by detailing the backpropagation algorithm.

1.1.3 . A neural network learns the mapping by observing a lot of labelled data

The physical system we have used to define the concept of computation can be, and must be, parameterizable. In this case, the relation between the output and the input also depends on these parameters. Thus by modifying the parameters, we modify the output while the input is kept constant.

In an artificial neural network, the synaptic weights act as parameters by influencing the propagation of the input signal to the output layer of the network. In the classical task of input classification, the network performs the transformation parameterized by θ from the input to the output layer:

$$\begin{aligned} output &= f(input, \theta) \\ &= network(input, \theta) \end{aligned} \tag{1.5}$$

Indeed, we use the state of the output layer to assign a class to the input, which is called inference. Most often we arbitrarily assign a neuron per class in the database to be classified, and

the class of an input corresponds to the index of the output neuron that has the most activity: $class = argmax(s_{output})$.

However, we cannot just choose the parameters of the neural network at random to make the right prediction. A complex task in the field of deep learning is then to find the right set of parameters that allows the neural network to approximate the theoretical function that relates inputs to their theoretical outputs (targets), or labels of the inputs. The reason why artificial neural networks are so powerful is that they "learn" - or update - their parameters by observing a lot of training data to approximate as well as possible $output = network(input_{training}, \theta)$, which amounts to approximating the statistical distribution of training data. By having the network learn on a fairly broad distribution, i.e., very many and varied examples, one hopes that the underlying distribution learned by the network will be broad enough to perform well on never-seen-before data - this is the generalization property, crucial for real-world use cases.

Before describing how to change parameters to improve the prediction that a neural network makes, we must first describe the goals of optimizing an artificial neural network.

Supervised, Unsupervised, Self-supervised learning. The optimization of an artificial neural network, also called the training step, obviously depends on the goal and the type of data on which the network will be trained. Thus, there are several types of optimization.

The first type of optimization, the one that interests us in this thesis, is the so-called "supervised learning". In this case, the artificial neural network has access to a database composed of pairs (input, label) where the label corresponds to the class of the input in the case of a classification task. Thus, we can know how much the neural network is wrong when it makes a prediction on an input by comparing the output obtained for a given input with the label of this same input. From this error calculated at the output, various algorithms will then prescribe updates which will have for effect to minimize this same error. By repeating these updates on many data, it is hoped that the network will eventually arrive at a parameter configuration that correctly predicts the class of all training data. Supervised learning allows the network to approximate the parameterized function $output = f(input, \theta)$. This will obviously depend on the size of the network and the ability of the optimization algorithm to prescribe updates that properly minimize the prediction error computed at the output of the network.

The second type of learning is unsupervised learning and corresponds to the case where the input data has no assigned class. This is the most frequent case when databases are found since assigning classes to inputs is time consuming and very expensive. It is also the simplest way to train a network on a lot of data. In this case, learning is not going to consist in minimizing an output error between the prediction and the label of the input, but rather in optimizing other objectives, such as the reconstruction of the input. In unsupervised learning, the network will not learn to approximate the function that relates the input to the output, but rather will learn to model the statistical distribution of the training data. The network will eventually be able to allow segregation/classification of the data based on their statistics relative to the distribution learned from the database. However, without a supervisory signal to guide the learning as done in supervised learning, unsupervised learning fails to reach the performance of supervised learning since it does not allow, or in a much less efficient way, the emergence of a detection hierarchy of the input features. Also, in many cases of unsupervised learning, we cannot formulate an objective to be optimized and therefore we cannot derive a learning rule that optimizes this objective. So we use heuristic learning rules such as the Hebbian learning [25] inherited from Hebb who wrote "neurons that fire together wire together" or the Spike Timing Dependent Plasticity rule where the synaptic weight connecting two neurons is increased if the

presynaptic neuron triggers an increase in activity for the postsynaptic neuron (causal relationship) and vice versa, if the post-synaptic neuron has its activity increased while the pre-synaptic neuron is not active, then the synaptic weight is decreased. Thus, although unsupervised learning can perform some tasks on large amounts of data, such as clustering, which groups data into clusters according to similarities found by the network, it does not reach the performance of supervised learning.

A third type of mixed learning that is taking more and more place in the current deep learning panorama is "self-supervised" learning where the network will learn to predict hidden patterns in the input: words removed from a sentence for an NLP model, a part of a hidden image for a vision model, ... This type of training is very powerful since we can train the models on huge databases without having to label each piece of data. This type of training is generally applied to very large models which will not be our object of study. However, it is a very attractive training method and a potential way to allow training on edge terminals without the need to label the data and thus without human intervention (see Section 1.3.3).

Although we will describe a supervised optimization procedure in the following, it also applies for the self-supervised learning case since we can write a cost function at the output.

Supervised learning with gradient-based methods. We will place ourselves here in the framework of supervised learning where, from the error calculated at the output of the network, we can calculate the updates of the parameters in order to minimize this error. The most used method for this minimization, called gradient descent, is based on the calculation of the gradient of the cost function. Indeed, to optimize (minimize or maximize) a mathematical function f which depends on a parameter λ, it suffices to update the parameter in the direction of the gradient [35], [36], or to perform the following update:

$$\lambda \leftarrow \lambda - \frac{\partial f}{\partial \lambda} \tag{1.6}$$

We add to the parameter the opposite of the gradient of the function computed at the point λ in the framework of the minimization of the function $f(\lambda)$.

This method can be extended to functions that depend on multiple parameters. We will next detail how we can leverage this optimization method in the context of training artificial neural networks.

Now it remains to apply this rule to an entire neural network, the cost function computed at the output being the function to be minimized, the parameters to be updated being the synaptic weights of the neural network. The state of the output layer, and thus the value of the cost function, depends on the state of all the previous layers, so changing the value of the network parameters influences the state of the output layer and thus the value of the cost function calculated from the state of this same output layer.

We could use a method of estimating the gradient of the cost function with respect to each parameter of the neural network to update the parameters of the network one after the other [38], for example by perturbing the parameters one by one and watching how the perturbation makes the cost function vary at the output of the network. However, since the output does not depend on a single parameter but on a multitude of parameters, the parameter-by-parameter optimization would probably not converge to an optimal minimum and would be extremely slow [39]. It is therefore necessary to optimize all the parameters in parallel each time an input is presented to the network.

Now it remains to know how we can compute $\frac{\partial f}{\partial \lambda}$ for all the parameters of the network where f is now the cost function of the network which expresses the difference between the state of the

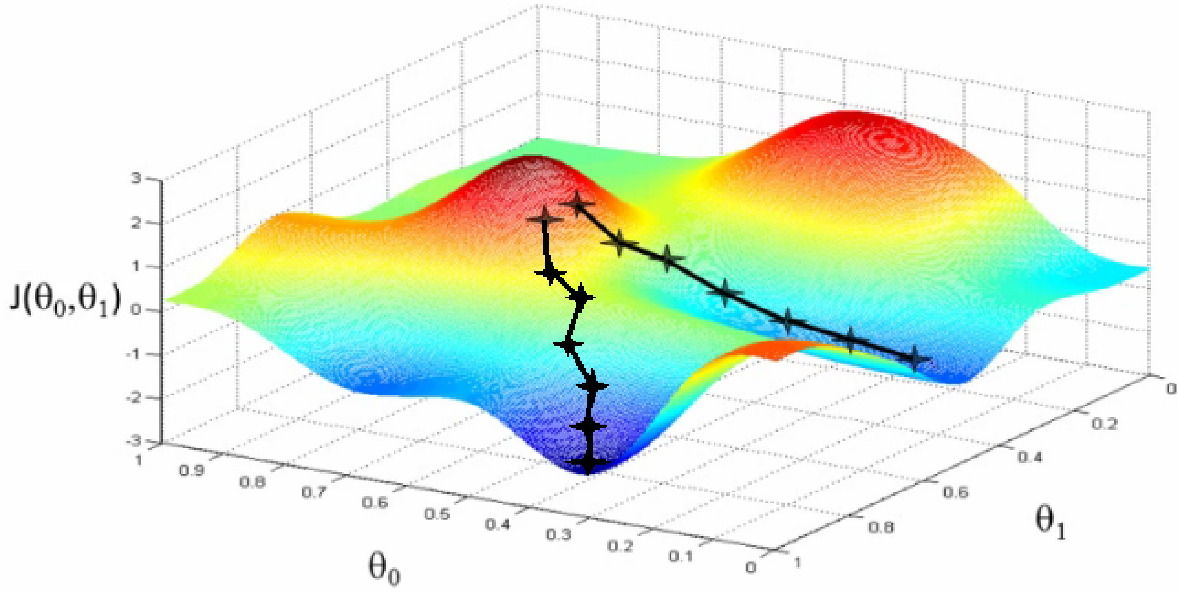


Figure 1.6: Stochastic gradient descent for a function that has 2 parameters. [37]

output layer of the network with the ideal state or target state corresponding to the input.

The procedure to compute the gradient of the cost function computed at the output of the network is based on the chain derivation rule and is called backpropagation [27]. Thus the calculation of the derivative of the cost function with respect to a given parameter in the network will depend on the derivatives of all the intermediate functions involved between this same parameter and the output layer.

Let's derive the operations involved in the forward pass (*i.e.* computing the state of the output layer given an input by computing the state of all intermediate hidden layers).

Each neuron in the network has a pre-activation part (x_i^l) and an activation (a_i^l) where l stands for the index of the layer. The state of a layer a^l depends on the activations of the previous layer weighted by the synaptic weights. Thus, for a fully-connected architecture, the neuron i in the $l-th$ hidden layer a_i^l receives x_i^l as input:

$$x_i^l = \sum_j W_{ij}^{l,l-1} a_j^{l-1} \tag{1.7}$$

where $W^{l,l-1}$ represents the matrix of the synaptic weights connecting layer $l-1$ to layer l.

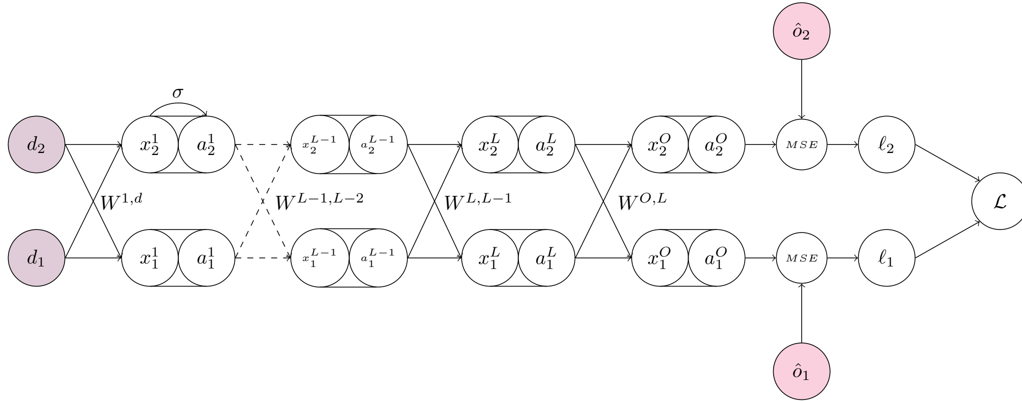
The neurons in the layer then apply a non-linear $\sigma(x)$ function called the activation function:

$$a^l = \sigma(x^l) \tag{1.8}$$

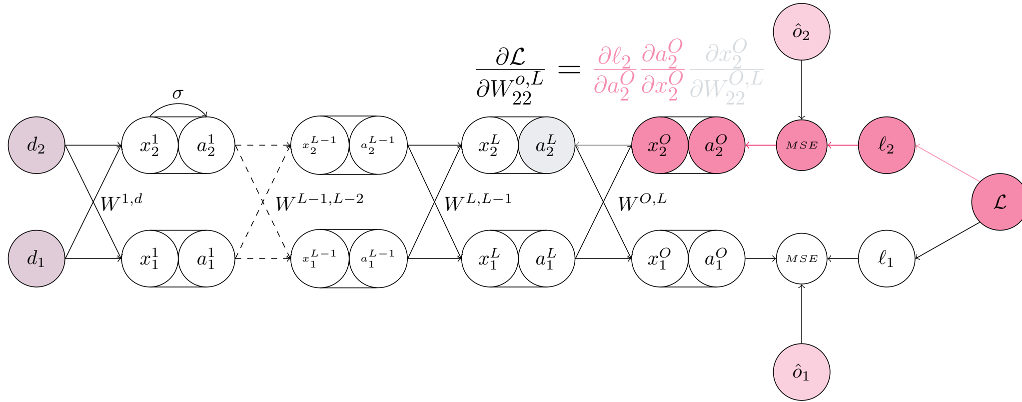
These two equations are repeated for each layer until we have calculated the state of the output layer o. We can then calculate the cost function:

$$\mathcal{L}(o, \hat{o}) \tag{1.9}$$

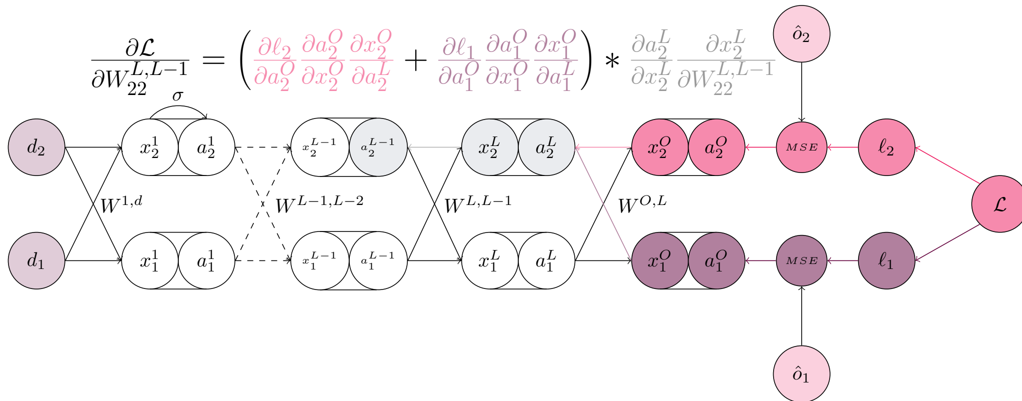
where $\hat{o}$ denotes the target state, which is the state in which the output layer should be given the input presented to the network.



(a) Forward pass of a neural network with L hidden layers - from the input to computing the global loss function $\mathcal{L}$. Each neuron has a pre-activation part (x_i^l) and an activation (a_i^l) where l stands for the index of the layer. Output neurons are denoted with a O superscript. ℓ_i denotes the contribution of output neuron i to the MSE loss in our case such as $\mathcal{L} = \ell_1 + \ell_2$. In the supervised learning settings, the input data consist in the couple (data, label/target): $(d_x, \hat{o}_x)$



(b) Error-backpropagation for the last weights: $W_{22}^{0,L}$ contributes to the cost function only via the output neuron a_1^O.



(c) Error-backpropagation for deeper weights: $W_2^{L,L-1}$ contributes to the cost function via both output neurons and the hidden neuron a_1^L

Figure 1.7: Computing the gradient of the global cost function with respect to the weights of a fully-connected neural network.

For more clarity, we can also rewrite:

$$\mathcal{L}(\sigma(x^o), \hat{o}) = \mathcal{L}(\sigma(\sum_j W_{ij}^{o,L} a_j^L), \hat{o}) \tag{1.10}$$

Now that we have written the operations involved in the forward-pass, we will derive those involved in the step of back-propagating the error computed at the output throughout the network, also called the backpropagation step.

The idea is to compute the derivative of all operations involved in the computation of the output layer o state between the use of the same parameter and the computation of the error at the output.

Thus, the gradient of the error function with respect to the parameter $W_{ij}^{o,L}$ connecting the neuron a_j of the last hidden layer and the neuron o_i of the output layer is written:

$$\frac{\partial \mathcal{L}}{\partial W_{ij}^{o,L}} = \frac{\partial \mathcal{L}}{\partial o_i} \frac{\partial o_i}{\partial x_i^o} \frac{\partial x_i^o}{\partial W_{ij}^{o,L}} \tag{1.11}$$

Each intermediate derivative is easily computable. If $\mathcal{L}$ is the Mean Squarred Error: $\mathcal{L} = \frac{1}{2}\sum_i (o_i - \hat{o}_i)^2$, then $\frac{\partial \mathcal{L}}{\partial o_i} = o_i - \hat{o}_i$. Moreover, if the activation function $\sigma(x)$ is the function ReLU such that $\sigma(x) = max(0, x)$, then $\frac{\partial o_i}{\partial x_i^o} = \frac{\partial \sigma(x_i^o)}{\partial x_i^o} = H(x)$ where $H(x)$ is the Heaviside function. Finally, the derivative $\frac{\partial x^{o_i}}{\partial W_{ij}^{o,L}}$ is easily computed from Eq. 1.7 and $\frac{\partial x_i^o}{\partial W_{ij}^{o,L}} = a_j^{l-1}$.

Thus, the learning rule for the weights connecting the last hidden layer to the output layer is written :

$$\frac{\partial \mathcal{L}}{\partial W_{ij}^{o,L}} = (o_i - \hat{o}_i) H(x_i^o) a_j^{L-1} \tag{1.12}$$

Now, we also need to be able to compute the gradient of the cost function with respect to the weights located deeper in the network. But we immediately see that the weights $W_{ij}^{L,L-1}$ affect all the output neurons because of the fully-connected architecture between the last hidden layer and the output layer. Thus, the gradient of the cost function with respect to this weight will depend on the error computed in the output at each output neuron.

Then, we sum the different branches that start from the hidden neuron h_i:

$$\frac{\partial \mathcal{L}}{\partial W_{ij}^{L,L-1}} = \sum_k \left(\frac{\partial \mathcal{L}}{\partial o_k} \frac{\partial o_k}{\partial x_k^o} \frac{\partial x_k^o}{\partial a_i^L} \right) * \frac{\partial a_i^L}{\partial x_i^L} \frac{\partial x_i^L}{\partial W_{ij}^{L,L-1}} \tag{1.13}$$

By replacing each derivative with its true value, we obtain the learning rule that computes exactly the gradient of the cost function computed at the output of the network with respect to a weight located deeper in the network:

$$\frac{\partial \mathcal{L}}{\partial W_{ij}^{L,L-1}} = \sum_k \left((o_k - \hat{o}_k) H(x_k^o) W_{ki}^{o,L} \right) * H(x_i^L) a_j^{L-1} \tag{1.14}$$

One can then continue to use the chain derivative rule of Eq. 1.13 to compute the updates for parameters even deeper than those weighting the inputs of the last hidden layer.

Once the gradient of the cost function is computed for each network parameter, we use the quantity obtained to perform the gradient descent step. However, we do not immediately apply the update of the parameters according to Eq. 1.6 to perform a standard gradient descent on the $\mathcal{L}$ cost function. Indeed, Eq. 1.6 applies to functions that depend on only one parameter. However,

artificial neural networks are parameterized by many parameters, and thus the cost function is also parameterized by many parameters. Thus, such a large step in the direction of the gradient given by Eq. 1.14 could not result in a good optimization of the cost function. Moreover, the gradient of the function must be computed for many different inputs sequentially, which never gives the same value. So, to keep up with the intricacies of the cost function, the parameters should only undergo small updates in the direction of the gradient. This gives rise to the stochastic gradient descent (SGD) algorithm (Fig. 1.6) [40] where each parameter is updated by an amount proportional to the computed gradient but modulated by what is called a learning rate η :

$$W_{ij} \leftarrow W_{ij} - \eta \frac{\partial \mathcal{L}}{\partial W_{ij}} \tag{1.15}$$

In Fig. 1.6 we illustrate how the stochastic gradient descent algorithm is applied to the optimization of an objective function J that depends on 2 parameters θ_1, θ_2. The black lines show two possible paths that go to a minimum of the function $J(\theta_1, \theta_2)$. The initialization of the network drives the final minimum reached after minimization of the function with stochastic gradient descent, hence the crucial importance of the initialization schemes of the parameters to reach good minima [41], [42].

The backward chain rule of derivation implies that the deeper one goes into the network from output to input, the more the computation of the gradient with respect to a parameter will require more memory and computational capacity. Indeed, for each forward operation, we have to calculate and store in memory the derivative of the operation. In the backpropagation phase, we have to fetch in memory all these intermediate results and calculate the sums of the product of derivatives to compute the updates of each parameter.

Methods exist to limit the memory requirement to perform backpropagation as with Pytorch, the function *checkpoint_sequential*[4]. This function allows to save in memory the state of only a few intermediate layers during the inference phase and to perform the backpropagation phase : recalculates the forward operations involved between two saved checkpoints. This function allows either to train larger models or to increase the batch size on a given material but at the same time increases the training time by 25% since it is necessary to recalculate inference steps during the backpropagation phase. But the memory requirement to use backpropagation to compute the gradient of the cost function with respect to each parameter of the network is still phenomenal and we will see why this necessity leads to colossal energy consumption and training time of artificial neural networks.

Why does it work? Artificial neural networks need a large number of parameters to be able to generalize well, *i.e.* to make the right inference on data that the network has never seen; we say that they are over-parameterized. However, this over-parameterization of the network makes the optimization task much more complex. Indeed, the landscape of the cost function is not guaranteed to be convex and gradient-based optimization methods such as backpropagation can cause the network to arrive in a local cost minimum. However, it has been shown that the over-parameterization regime of neural networks instead allows to get rid of the non-convexity of the cost landscape and that increasing the number of hidden units increases the generalization ability of the network [43]. Increasing the hidden units also allows to express the features of the training database more globally, avoiding the network to focus on a few features only, which leads to over-fitting on the training data

[4]Pytorch documentation for *checkpoint_sequential*

but poor generalization on data not yet seen. A more serious problem than the non-convexity of the function is the presence of saddle points [44], [45] where the gradient becomes almost zero and can then prevent the training from progressing. These problems related to the optimization itself are beyond the scope of this thesis but are subject of many current studies.

The optimization of neural networks can be improved by applying certain principles. Stochastic gradient descent, where the gradient of the function with respect to the network parameters is computed on a single example, can be replaced by a gradient descent computed and averaged over several examples. This is the mini-batch gradient descent method. The gradient is then much less noisy and converges much faster and better to a minimum of the cost function. We can also add a "momentum" term to the gradient [27], [46]: $\Delta W_{ij}^t = \alpha \cdot g_{ij}^t + (1-\alpha) \cdot g_{ij}^{t-1}$ which is in a way a memory term of the past updates and will allow the network which arrives in a global minimum to escape from it. Momentum is inspired by the physical concept of the same name, which means that an object in motion continues to move for a certain period of time even though the force that caused the motion is no longer applied to it. For synaptic weights, the motion corresponds to their evolution during the update phases and the force at the origin of their update corresponds to the gradient of the cost function, which cancels in a local minimum or in a saddle point.

1.2 . Neural networks are usually trained on standard digital hardware that are power-inefficient and thus exhibit limitations with the current large models

We have just described how supervised learning methods based on the gradient - backpropagation for example - can train a neural network to solve an input-output mapping task. We have also seen that neural networks draw their power from their over-parametrization which gives them the ability to approximate any function $output = f(input)$ after optimization. However, this over-parametrization is nowadays a challenge from the point of view of energy consumption, required computing power and training time. Indeed, artificial neural networks are currently trained on digital CMOS hardware based on the Von Neumann architecture. The accuracy of vision [47] and natural language processing [4], [48] models increases when the number of parameters increases. The current trend of deep learning is thus to increase the number of parameters of the models (Fig. 1.8), which means that a strong increase in the energy and time consumed to train such networks on standard hardware is to be expected.

Artificial neural networks are nowadays mainly trained on standard digital hardware, *i.e.* multi-purpose CMOS processors. These processors have this property of having a great abstraction with respect to the architecture of the network to be trained but also with respect to the mathematical operations involved in the network. These processors are said to be "universal" in the sense that they are highly re-configurable and that running in binary logic, they can implement any requested algorithm. This allows to train any kind of neural network (fully-connected [20], convolutional [28], transformer [49], ...) on the same processor and thus avoids having a machine/chip specific to the architecture of a given network. This is very useful especially during the hyperparameter search phase where the network architecture is optimized to obtain the best possible performance. Generally speaking, it is the accuracy that is maximized, but we may also want to minimize the number of FLOPs[5] to reduce computational complexity for example. or when the same processor is used for

[5] FLoating Point OPerationS - metric widely used in the deep learning community to denote the numbers of operations required for a single instance/inference of a deep learning model. Not to be confused with FLOPS

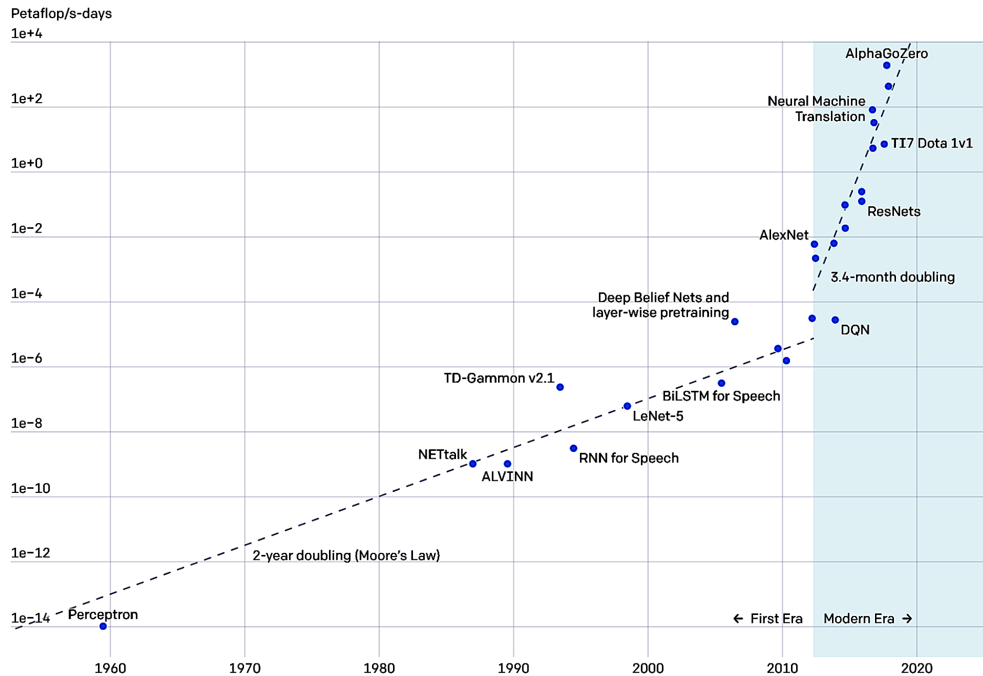


Figure 1.8: Evolution of the computing resources required to run the machine learning models over time. 2012, the year of Alexnet, clearly separates two regimes where the demand of computing power keeps increasing. Source: OpenAI

different purposes.

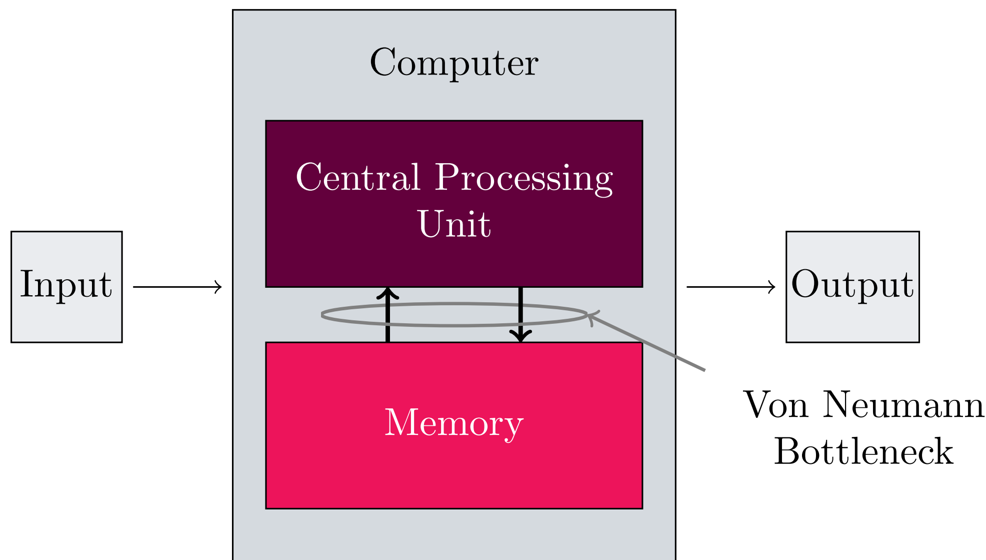


Figure 1.9: Sketch of the Von Neumann architecture

These conventional hardware, CPUs (Central Processing Units) in particular, are based on the so-called Von Neumann architecture (Fig. 1.9) [5]. In this architecture, the computing unit and the memory are dissociated and connected by a bus. Thus to carry out a certain calculation, the

= Floating Point Operations Per Second - which describes the computing power of a given hardware.

processor must go seek in memory the data implied in the calculation, put it in local memory, carry out the operation with the data at the level of the logical arithmetic unit then return the result in memory. This round trip of data between the physical memory and the calculation unit is named Von Neumann bottleneck [6]. This bottleneck has a speed of transfer of the data between the physical memory and the processor limited at the same time by the speed of computation of the arithmetic unit of the processor but also by the time of load of the bus that transfers the data between the memory and the processor. The training of neural networks, which have a very high number of parameters, is then slowed down since it is necessary to fetch each parameter in memory, to carry out an operation with it and to return it to memory. Moreover, training a neural network with a large number of parameters and neurons coupled with the use of a learning algorithm such as backpropagation, which makes intensive use of the intermediate values of the computations performed during the inference phase and stored in memory, is extremely slow and energy consuming:

Operation	Relative energy consumption
MAC	1x
Access data from cache	6x
Access data from DRAM	200x

Table 1.1: Energy to transfer data from the memory to the processor compared to the energy used to perform a MAC operation. From [7]

The use of GPUs to carry out the computations [30] related to deep learning solves part of the problem since this type of processor is highly parallelized and circumvents, in a limited way, the Von Neumann bottleneck. However this type of hardware is very energy consuming.

Therefore, the current trend in deep learning that consists in increasing the number of parameters of the different models and training them in Von Neumann hardware does not seem efficient for the future, from different point of views:

- Temporal: the models will take more time to be trained. Bloom [50], for example, is a NLP model that has 176 billions parameters which has been trained in 117 days on 384 Nvidia A100 GPUs.
- Economical : the computation time is expensive. Bloom has cost €3M to be trained.
- Ecological : this computation time also corresponds to CO_2 emissions which, estimated in a conservative way correspond to $32 * 10^3 kgCO_2^{eq}$. To get this estimation, we followed the following observations:
  1. BLOOM has been trained on 384 A100 NVIDIA GPUs that operate at 300W, so the whole setup operates at 115200W.
  2. BLOOM has been trained for 117 consecutive days that is 2808 hours (117x24).
  3. Then the whole energy consumed is $115200 * 2808 = 0.32GWh$.
  4. Finally, BLOOM has been trained on the Jean Zay supercomputer in France where the averaged carbon intensity of the electricity is $0.1kgCO_2^{eq}/kWh$ [51] (which is very low due to the large nuclear power plant park in France, in Europe the average carbon intensity is $0.45kgCO_2^{eq}/kWh$ [51] which impacts a lot the total emission of the training).

This estimation does not even take into account the power required to cool down the datacenter and the power required to synchronize and fetch from all the GPUs.

5. The resulting carbon emitted is then $3.2*10^5 kWh*0.1kgCO_2^{eq}/kWh = 32*10^3 kgCO_2^{eq}$.

It is the same amount of CO_2 that 4 french people would emit in a year [52].

We list two possible alternatives that are currently being developed:

- The first alternative relies on the same software approaches and hardware technology. However, there are dedicated efforts in order to build software (learning algorithms, size of the model, computational ressources required) and hardware that overall use less memory and computational ressources so that the power required to train the models is less as well as the CO_2 emitted. This is the most straightforward approach and it concentrates the majority of researchers working on both the software and the hardware of AI.

- The second alternative diverges radically from the first one. This approach relies on the natural tendency of some physical systems to perform non-linear operations (as those involved in neural networks) to realize computations. That approach is very promising as some physical systems operate with very low powers but also as it proposes to go beyond the Von Neumann architecture with the in-memory computing paradigm, where the memory and the processing elements are brought together to avoid data movement.

In the next sections, we will first review some work about what we call the standard approach. Then we will dive into the physics-driven computing world and argue why using physics to compute may lead to breakthrough in terms of low-power computing and low-memory requirements in the near future.

1.3 . Increasing the power-efficiency of standard digital approaches

In this section, we focus on attempts to reduce the computational complexity of deep learning models and learning algorithms, but also on attempts to improve the standard hardware used for artificial neural networks. These approaches focus primarily on large-scale tasks that often run in the cloud due to the size of the models. Before describing these two paths to energy-efficient deep learning, we will see how one can already improve the energy efficiency of deep learning by simply applying some rules and practices. This is a short-term solution to the energy consumption problem of deep learning. In the final section, we will see that there is an opportunity for low-power computing hardware, especially for applications that run on devices where compute and memory budgets are very limited. We will therefore enter the neuromorphic world.

1.3.1 . Better machine learning practices and open-source can reduce the energy and carbon footprint

Before describing attempts to improve standard software and hardware approaches, it is worth to start by watching if we can find an ideal combination of existing deep learning tools (software and hardware) that can already help reduce the carbon emissions and energy bill of deep learning. Interestingly, a recent paper [53] advocates that standard deep learning can achieve carbon neutrality while being trained and run on CMOS devices with backpropagation if we add simple rules during the training process.

In this paper the authors claim that previous papers [54] overestimated carbon emissions of ML models and that not only ML emissions are not that important but they will also decrease rapidly in the following years. For that purpose they propose 5 rules to follow when training a ML model:

1. Train the best model: they show that using Neural Architecture Search (NAS) [55] for getting the lighter version of a standard NN architecture allows to reach same accuracy as with dense models but with much lower number of parameters and thus reduces the energy for training the model. Interestingly, they show that even if NAS consumes some energy to select the best sparse architecture, NAS is only performed on a simple task so the overall gain in energy is positive :
$$Energy(NAS) + Energy(best\ sparse\ model) < Energy(dense\ model) \tag{1.16}$$
They also underline that models found via NAS are often made open-source so that many users and/ or companies can use them as trainable already FLOPs-optimized sparse neural networks. Also, most of the big models that are trained (after the NAS step) are accessible online (via Github, Pytorch, Hugging Face, ...) as open-access models, so many users can use the model as an inference-only engine thus saving the energy consumed for training. In computer vision it is a technique widely used to exploit the convolutional part of a large ConvNet as a feature extractor and then just fine-tune the classifier at the very end of the network to fit the use case dataset. That is why it is important to have initiatives that propose large open-source models to be used by the community after training. Bloom [50] or OPT [56] are great steps in that direction for natural language processing (NLP) where the number of parameters is astonishingly large.

2. Select a good standard hardware: as employees of Google, the authors extensively use Tensor Processing Units (TPUs). TPUs, that we will describe in the next section, are CMOS processors extremely optimized to do computation on tensors, the core data representation in deep learning. The authors show that every new generation of TPUs reduces a lot the energy consumption of ML training compared to previous versions mostly because of better logic and better mapping on the chip: TPUv4 (780MFLOPS/W) improve the energy efficiency by a factor 13.7 compared to unoptimized P100s GPUs (21 MFLOPs/W) where the TPUv2 (170MFLOPs/W) only improves by 5.7.

3. Train your model on the cloud: they show that datacenters optimized for doing ML better use the energy required for cooling, computing, etc versus standard datacenter. Also it is important to notice that training models on the cloud can reduce the number of hardware devices required overall as they are extensively used by different users - compared to the case where each user has locally its own hardware. It is in line with the recent trend of sobriety emerging in our society.

4. Choose the best location for your datacenter: the authors show that depending on the location of the datacenter you use (the datacenter is selectable via the cloud interface), the electricity provided has more or less carbon intensity. Some states in the US have almost 100% renewable energy while other states run 100% on oil to produce the same electricity. However, it does not tackle the problem of how we produce low carbon intensity energy, that is huge.

5. Finally the authors highly encourage authors to publish the energy consumed to train a model and that this criterion should be taken into account as well as the accuracy to classify best models (idea already developed in the paper « Green AI » [54] in 2020).

This cloud-based approach is interesting as it relies on the fact that even if a model is large and can consume a lot of energy to be trained, if its deployment as an inference model is distributed on many users, then the overall impact is little compared to the scenario where each user train his own model. This motivates the open source publication of the big models trained on large datacenters.

This approach sounds like a good near-term approach to maintain sota accuracy on complex and daily-life tasks while diminishing the carbon intensity of the trainings that the others unconventional approaches that we will detail next still fail to reach. However, the problem of green energy production and the scaling law of moderns models stating that "larger is better" make unconventional approaches appealing for the future. Indeed, these unconventional approaches aim at delivering hardware lowering by 3 to 6 order of magnitude the energy consumption [57] (thus in carbon emitted), justifying the effort to design new hardware.

1.3.2 . Lighter models and more local learning algorithms improve energy-efficiency with less memory requirements

We have just mentioned NAS [55] as a technique to get models that perform as well as the sota models but that are lighter (in terms of the number of parameters). This technique is in line with many others techniques that aim at reducing the size of deep learning models while maintaining reasonable accuracy on standard datasets. Neural Architecture Search is a method where multiple architectures are trained on the same task and the best one regarding to a specific criterion is selected. While the criterion is mainly the accuracy of the model, in order to get models that are lightweight, it can also include the size of the model as well, thus NAS finds the best compromise between accuracy and model size.

Pruning [58] is an alternative approach for reducing the size of the model. It removes some parameters in the network that do not contribute a lot to the performance of the model *i.e.* when pruned the accuracy does not drop. In fact, pruning relies on the property of neural networks to have under-parametrized sub-networks that perform as well as the entire networks. This has been highlighted with the Lottery Ticket Hypothesis paper of Frankle *et al*. [59]. Those approaches still rely on full precision variables (either parameters and neural pre-activations and activations). However, such FP32 variables are vey computationally and memory inefficient because they require a lot of bits to store and compute a single data.

There are works [60]–[62] about quantizing those parameters and activations in order to reduce the computational complexity and the size of the model. In these works, FP32 variables are reduced to either fixed point variables with less bits precision or even integer variables that have 16, 8, 4, 2 (ternary neural networks) or even 1 (binary neural networks) bit precision. This has been extremely efficient at the task of reducing the size of models. However this approach often needs more parameters as the information loss due to the quantization can be large. A last improvement that rose recently is the search for architectures relying on operations that are computationally less expensive than the traditional ones. One example is the MobileNet architecture [63] that extensively use dephtwise separable convolutions to reach near-sota accuracy with a model that can fit the computational and memory budget of a mobile phone for instance, hence the name of the architecture.

All those methods perform great at reducing the size and the computation complexity of the models. However they still rely on backpropagation (Section 1.1.3) that is highly non-local and thus requires a huge amount of memory in order to compute the parameter updates. Alternative learning frameworks with more local learning rules are now sought in order to reduce the requirement in terms of memory. In this context we can cite Feedback-Aligment [64], Target Prop [65], [66], three-factors

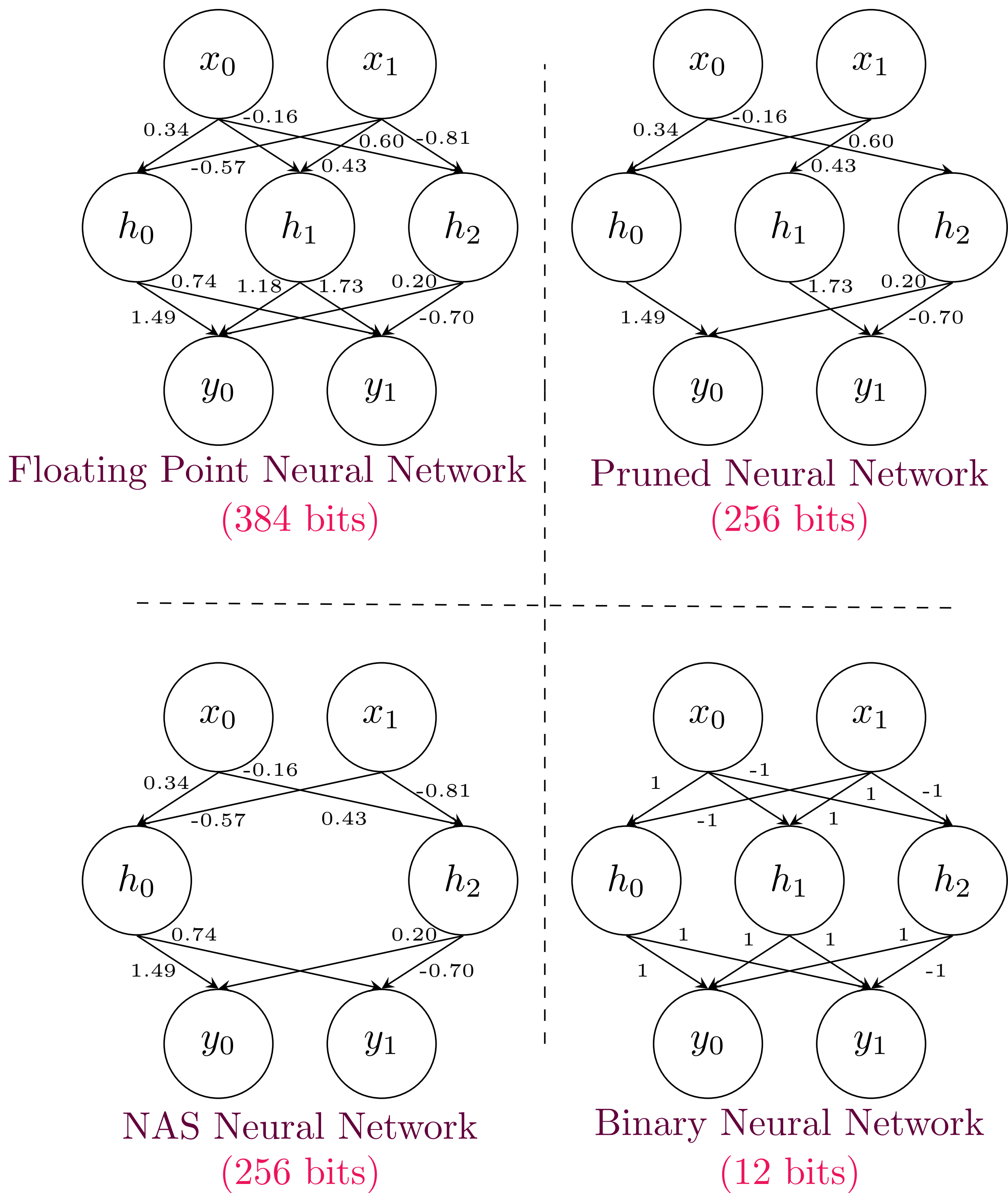


Figure 1.10: Three different ways to reduce the size of a initial model: Pruning [58], NAS [55], Binarization [61]. Rough estimation of the models size: $size = \#parameters * 32\ bits$ - FP32 is the mainstream precision for the parameters of neural networks trained on GPUs.

hebbian learning [67], ...

Those learning algorithms propose local learning rules for updating the parameters but the locality

is not yet leveraged at its best as the hardware on which the algorithms are run is still non-local due to the Von Neumann architecture. In the next section we will introduce attempts to design new hardware that do better that Von Neumann architectures.

1.3.3 . Specialized digital hardware improve energy efficiency with reduced data movement and massive parallelization

As mentioned before, two main types of workflow exist in deep learning. The first one, already mentioned, is the one based on decentralized computing done in servers or "cloud computing". The use of the cloud is done both for training models and for inference when the models are really large (recent NLP models for instance). The second workflow is a response to the problems posed by cloud-computing and tends to train deep learning models and perform inference on "edge" terminals where computing power and memory are limited.

"Server" computing. This paragraph groups together two types of computing at a remote server level - "cloud computing".

- The first type of use is when the training of a neural network and the inference are done at the server level. This is often the case when the model is very large and requires a lot of FLOPs to be trained but also for the inference step. We then use the important computing power of the server to carry out the training and the inference. This use of datacenters for the training allows to train models whose size is important as we can distribute the computation load on many processors inside the server.

  Once the model is trained and stored on the server it can be used as an inference model. In this case, the input data used for the inference is sent as a request to the server by the user. The response is the result of the inference. This type of infrastructure can pose several problems when doing inference:

  1. Latency: the time between the sending of the request and the receiving of the answer, and thus of the inference result, includes the inference time at the server level but also the travel time of the request and the answer on the Internet network. When the inference result is required in a short period of time, in autonomous driving for example, this latency introduced by the fact of moving the computing center can be critical and prevent the system from working (or cause serious problems).
  2. Data privatization: the second problem with sending data for inference on an Internet network is the security that relies on its data. In the case where the data contains confidential information (defense, nuclear, ...), this remote sending is not possible.

- The second type of use is when the training of a neural network is done on a server but the inference is done on edge terminals whose computing power and memory are much more restricted. Thus, the model once trained at the server level is sent to the edge terminals to be used as an inference model - sometimes after steps to lighten the size of the model with the methods described previously (Section 1.3.2). No training is done on this model on the edge terminal.

In both of these cases, the use of optimized CMOS hardware to perform the computations done in deep learning is extremely important since the speed and energy to train a model depends on the speed of the processors to do the common operations of a neural network. In fact this ability

to execute the computations done during the inference and backpropagation stages efficiently has been crucial to get deep learning off the ground as demonstrated by the training of AlexNet [30] in 2012 (Fig. 1.8).

More recently, Google designed a new kind of hardware to train neural networks that is specialized in performing operations on tensors[6]. This hardware is called "Tensor Processing Unit" (TPU) [68], [69]. In fact, [70] found that each kind of standard CMOS hardware (CPU, GPU, TPU) is specialized for a given task : TPU for convolutional neural networks, GPU for fully-connected neural network and CPU for recurrent neural networks (see Fig 3.7).

TPUs constitute a good transition to the next paragraph were we will discuss dedicated hardware implementations for edge computing. Indeed TPUs are a kind of ASIC[7] designed to accelerate neural network operations, however, they are still relatively universal in the sense that one can train multiple types of neural networks on the chip as it capable of some level of abstraction. Hardware implementations for edge computing often lose that capacity of abstraction in order to be the most efficient possible at training/ doing inference with a specific type of neural network.

Edge computing. Contrarily to the hardware that are used to train neural network in datacenters, hardware that run on edge devices for the same purpose have little computational and memory budgets which is a strong limitation for training and doing inference with large neural networks on those chips as a lot of memory and computational power are often required as we stated in the previous section. Training being the most memory and computationally intensive task, most hardware for edge applications are designed to be inference-only engines. FPGAs and ASICs are such dedicated hardware implementations that are very fast and low-power alternatives to general purpose processors such as CPUs, GPUs, TPUs, ... An example of such a chip developed for embedded applications is the Neural Engine for iPhones [71] - an ASIC dedicated to run neural networks on mobile phone designed by Apple. Even Tesla builds its own inference chip "FSD chip" [72] for "Full Self Driving" which features 2 embedded neural engines that are dedicated to perform the operations of convolutional neural networks (Fig. 1.11). The "FSD chip" supports most of the operations (inference only) involved in Self-driving features of Tesla cars despite the fact that it runs with only 72W [72], which is impressive.

For most applications, the standard workflow where the model is first trained on the cloud and deployed on edge devices without customization as inference-only engines is just fine.

But there exist applications, mainly when the two majors limitations of cloud-computing described above prevent the use of the standard workflow, where it is interesting to be able to perform training on the edge. There are also situations where the 72W consumed by the Tesla FSD chip is not even imaginable. Both cases require ultra-low power hardware and a clever usage of the memory to be able to train and run neural networks on the edge.

These scenarios fall under the scope of the use of neuromorphic chips. Neuromorphic computing has been pioneered by Carver Mead in 1990 [73] who coined the term which initially referred to the use of standard devices in an unconventional way such as transistor in their analog regime to compute [73]. Nowadays the neuromorphic computing field encompasses as well the unconventional hardware built with either standard digital hardware (transistors) or emerging devices and assembled in the intention to circumvent the Von Neumann bottleneck as the new computational schemes that

[6]Tensors are N dimensional matrices extensively used in neural networks. For instance the convolutional weights are stored as a 4 dimensional tensor where each dimension has the size: $fan_{in} * fan_{out} * kernel_{dim_1} * kernel_{dim_2}$.

[7]ASIC=Application Specific Integrated Circuit

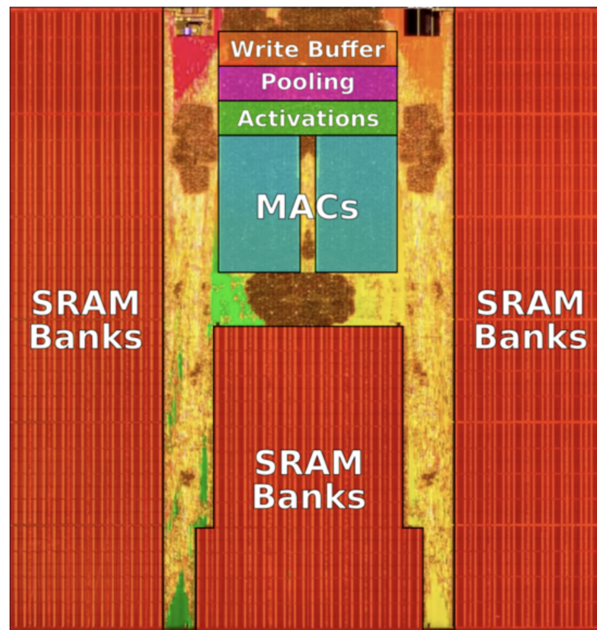


Figure 1.11: Neural engine part of the FSD chip from Tesla - taken in [72]. The Neural engine features custom parts that perform specific computation involved in convolutional neural networks: MAC (multiply and accumulate) operations, activation functions and pooling operations.

leverage at best those new hardware.

Behind the generic name of neuromorphic are two types of neuromorphic approach. One based on a non-conventional use of standard digital hardware (CMOS), the other oriented towards the use of emerging components the build either digital or analog systems. Both approaches seek to emulate behaviors borrowed from biological neurons to perform the calculations of neural networks and incorporate a local learning component that, although often unsupervised, is a first step towards the possibility of learning on the edge.

The first approach is pushed by the major foundries of the CMOS industry and is based on digital hardware but exploited in a way that goes beyond the Von Neumann Bottleneck. Two recent chips have shown to be the flagship of the digital neuromorphic approach. These chips are TrueNorth from IBM [74] and Loihi 1 [75] and Loihi 2 [76] from Intel.

They are based on the use of "spiking" neurons which allow a digital communication between neurons and are therefore more reliable. Moreover, this type of neurons does not emit spikes in a continuous way but more in a sparse way. This allows a huge gain in terms of energy: Loihi chips [77] exhibits an energy gain of a factor 300 compared to the Intel i7 processor for a synaptic event and of a factor 100 for a neuron update, which allows an energy gain of two orders of magnitude compared to standard digital processors while keeping the same technology. These chips confirm the interest of architectures where memory and computational elements are distributed and of course close together. Inference hardware based on spiking neurons is a very active field of research, in particular the possibility to do supervised deep learning [78], [79], driven by the low power promises of such networks. Loihi proposes in particular the implementation of local learning rules such as the Hebbian learning rule but also rules with a "third" factor [67] which modulates the local rule by a global error signal.

The second approach departs from the digital neuromorphic approach and leverages mostly

analog physical properties of some devices to perform unconventional computations. This approach is the main topic of the next section.

1.4 . Summary: sota accuracy with gradient-based global optimization + energy efficiency with less data movement

To begin the introduction to the topic of the thesis, we started by zooming in on the computation that interests us by defining computation as the evolution of a physical system from a state A to a state B. This transformation is used in machine learning to perform tasks such as image classification with algorithms such as artificial neural networks. We discuss the power of deep learning that comes from the gradient-based optimization of a global objective function where a hierarchical representation of the input emerges. This hierarchy arises when depth is combined with a global objective optimization. The power of deep neural networks has been demonstrated in 2012 with AlexNet which halved the error obtained with previous machine learning models on ImageNet by adding 3 additional layers.

However, as we have seen in Section 1.1.3, learning rules optimizing a global objective (backpropagation) require information that is not accessible locally and thus the computation of the update of a parameter involves the movement of distant data and thus an energy and time cost (see Section 1.2). Moreover, backpropagation-type learning rules call for different operations than those supporting the inference phase and thus require the implementation of hardware units dedicated to the computation of errors, thus increasing the energy, computational and temporal cost of the optimization phase. This type of learning implemented on CMOS hardware based on the Von Neumann architecture leads to very high energy consumption.

Although approaches based on cloud computing combined with open source models are emerging and are promising, there are situations where training and inferring with a cloud-based model is not possible.

Then it is promising to turn to other methods/hardware to make the calculations usually made on digital CMOS. Using the physics of devices and systems could allow orders of magnitude gains in terms of energy consumption for neural networks as we will see in the next section.

Chapter 2

Unconventional physics is a promising low-power alternative to conventional digital hardware for deep learning

In the first chapter we have seen that deep learning relies on neural networks that are over-parametrized and learn on data by optimizing a global cost function using the algorithm of back-propagation that computes the gradient of that function to reach the high performances we know.

By going back to the characteristics of conventional hardware, we will see that the limits of current deep learning we developed in the end of the previous chapter, are in fact inherent to this type of hardware and therefore motivate unconventional alternatives that perform the same calculations at a lower energy, economic and carbon cost. We will show that using the dynamic behavior of certain physical systems to perform computation is a promising approach in the search for new hardware. We will rely on recent results using physics to perform the inference step of artificial neural networks to see different ways to introduce parametrization into those systems in order to optimize them. We will see in particular that the energy function of some physical systems is interesting to exploit for computing and to parametrize since it relies on the natural tendency of physical systems to evolve towards states that minimize their energy (*i.e*. the most probable state). These energy-based approaches are highly sought after for hardware implementations since they propose local learning rules, *i.e*. they only depend on spatially-close measurable quantities, and thus only require a simple circuitry to compute parameter updates. However, these approaches have faced the difficulty of having a hardware implementation that rivals the accuracy obtained by backpropagation on the same tasks. Indeed, the parameter updates prescribed by early energy-based approaches only optimize local objectives, i.e. at the layer level, rather than a global objective such as a cost function to be minimized at the output of the network. Thus, no hierarchy of characteristics emerges, preventing the high accuracies reached with the backpropagation algorithm, which is by nature highly non-local and thus not very adapted to drive low power hardware implementations. Indeed, the circuitry and memory that would be required to compute the gradient with backpropagation for an energy-based physical system would be very complex and power hungry which could overshadow the energy gains allowed during inference by the physical system.

Large-scale hardware implementations, therefore scalable on tasks of interest, also suffer from the variability of the components used to emulate both synapses and neurons. Indeed, not only do synaptic components not react linearly to applied updates, as a full-precision variable does in simulation, but their behavior differs from one component to another. Neural components can also be noisy to the point of not having the same activation for the same example from one run to another. Without a complex procedure to limit the impact of this variability on the performance of the physical system to perform a deep learning task, and thus a procedure that is also energy intensive, the training of such hardware is compromised. The challenge today is to succeed in training an energy efficient hardware implementation of neural networks with an algorithm that calculates the gradient of a cost function computed at the output of the system like backpropagation does, but with a local learning rule that is simply computed, limiting the energy overhead and this despite the variability of the emerging low power hardware still mostly experimental.

2.1 . Leveraging physics to perform unconventional computing

The standard approaches to compute described above are all based on multi-purpose digital hardware. This allows them to emulate in a re-configurable way all the operations that make up an artificial neural network: multiplication by the synaptic weights of the inputs and summation, non-linear behavior of the units called "artificial neurons". These processors are able to make great abstraction of the architecture of the trained network and can emulate a very large network although the processor is limited in physical size, as long as the associated memory supports the number of associated parameters. But, as we have also seen previously, this approach has its limits, especially in terms of energy efficiency and computation speed.

These multi-purpose reconfigurable digital processors owe their ability to abstract the task they perform from the architecture of the processor itself to the theoretical work of Turing and Von Neumann in the mid-20th century. At this period Von-Neumann and Turing conceptualize a machine operating with a binary logic rather than a decimal logic as it was the case in the first half of the 20th century and show that this machine operating in binary logic is a universal machine capable of executing any program requested. Also, this logic combined with the advent of transistors in the 1970s allowed the design of processors based on TTL or CMOS logic as we know them today.

However, the use of digital processors to perform calculations in artificial intelligence has not always been obvious. The field of AI started long before the birth of the first integrated circuits [22], [24], [26]. The pioneers of AI imagined to realize calculations by taking advantage of physical phenomena governing the behavior of certain components. A famous example is the invention of the parametron by Goto and Von-Neumann [80]. They use the property of resonators to oscillate at half the frequency of an excitation frequency, with or without phase shift with respect to this signal, to calculate. This property makes the resonator a two-state component that can encode a bit in the phase of the component. This way of encoding bits was very successful before being supplanted by transistors which were first used as discrete components before being assembled in integrated circuits. The number of transistors that can be placed on these chips has evolved incredibly, a growth described by the famous Moore's law [81] until recently - offering with each new generation of chips an ever more powerful computing capacity. Unfortunately, in the last few years, we have seen a stagnation in the number of transistors that can be put on a chip due to the small size of the etched features required, making the behavior of transistors enter the probabilistic domain and no longer the deterministic domain for which they were known.

At the beginning of the 1990's Carver Mead was interested in using transistors in their analog regime [82] to try to reproduce certain sensory functions of living beings such as sight or hearing. He called this approach, very different from what was being done in terms of computing hardware, neuromorphic computing. In fact he based his work on the observation that transistors showed certain properties that were similar to those of biological neurons and therefore they were potentially good candidates to perform the same task on a chip.

Mead takes up the idea of using physical phenomena to encode information or to carry out calculations, which for a long time had been supplanted by the power and versatility of digital processors, but goes further since he makes his chips work in the low-power analog regime, which henceforth made these chips lose their general-purpose character but made them special-purpose. An interesting example is the development of an artificial retina in 1990 by Mahowald and Mead [83].

We propose to make a small review of the paths taken by the research of new hardware to emulate the computations used in artificial intelligence based on the physical behavior of components. Despite

the incredible number of works on this subject, we can see the emergence of two main paths:

- The first one consists in finding components that, at an individual scale, intrinsically perform an elementary operation of a neural network: weighting of the input by a parametrizable coefficient = synapse, summation of several signals, non-linear behavior = neuron. We will describe some recent ideas based on long known physical laws as well as on emerging components and therefore still at the experimental stage.

- The second way explored is based on the property of neural networks to perform a non-linear parametrized mapping between inputs and an output. Physical systems work in the same way. Given inputs, they evolve, transform, mix these inputs into a measurable quantity that depends on these inputs. We will see a recent work where the inputs are both the input data to be processed and the parameters of the system.

2.1.1 . Arranging emerging devices as a neural network-like hardware

This approach consists in designing an artificial neural network in hardware. That is to say, recreating hardware synaptic connections that weight a signal applied to them as well as recreating hardware artificial neurons that sum their inputs and apply a non-linearity.

Thus, each operation performed in an artificial neural network is associated with a well-characterized physical component or process that intrinsically performs the same arithmetic. As synapses and neurons are not performing the same operations, it is easy to imagine that we need different physical components/processes to implement synapses and neurons in hardware.

One obvious difference is the memory property that synapses have. They should indeed have adjustable but stable values in time in order to implement the parameters of the neural network for inference as well as during the optimization phase. On the contrary, the value of neuron activation functions varies according to the inputs presented to the network. Also, we have seen how the fact that the memory, storing the synaptic weights, is separated from the computational unit (Von Neumann architecture) dramatically impacts the power consumption and the time required to train artificial neural networks. We will therefore first discuss how the use of physical components with intrinsic and non-volatile memory in conjunction with physical laws has allowed the design of new computational architectures where the memory and the computational units are physically joined. These components are memristors whose conductance can be continuously modified between two states in a reversible but non-volatile way. Assembled as cross-bar arrays, Kirchhoff's laws allow to calculate the product of a vector of inputs - encoded as voltages applied to the memristors - with the matrix of weights encoded as conductance of the memristors. The example of the hardware translation of the vector-matrix product by a cross-bar array illustrates the philosophy of these hardware approaches, where efforts were made to group the memory and the processing elements - which were separated in Von Neumann architectures - in the same place.

- **Hardware synapses.** In this first part we detail the operation of memristors as well as their assembly in the form of cross-bar arrays which is the reference architecture for this type of device.

  The memristor, although theorized for a long time [84] was only recently highlighted as an excellent candidate to implement synapses experimentally [85]. Being based on different technologies, the main property of a memristor is that its conductance, that is non-volatile in the absence of inputs, can be changed according to a certain procedure - most often by sending programming pulses to the memristor terminals. This property makes it a component

of choice for emulating the behavior of a synaptic weight that can be updated during a learning procedure. The operations involving synapses in a neural network are the "Multiply and Accumulate" operations where "Multiply" refers to the operation of weighting by a synaptic weight the input at the synapse and "Accumulate" refers to the summation of weighted inputs at the input of a neuron. We will see how the memristor, at the level of the individual component, performs the first operation "Multiply" and that its integration in a cross-bar array allows to perform the "Accumulate" operation.

- *Multiply.* A memristor is first of all a resistor. Thus, if we apply a voltage V to the terminals of a memristor of resistance R, the current I that will flow through the device is given by the Ohm's law:

  $$I = \frac{1}{R}V \tag{2.1}$$

  or if we denote G the conductance of the memristor which is equal to $\frac{1}{R}$:

  $$I = GV \tag{2.2}$$

  This operation of multiplying an input value, the voltage U, by a scalar, here the conductance G, is the same operation performed at the level of the synapses of an artificial neural network where the activations of the neurons of a layer are weighted by the synaptic weights. Thus a memristor performs intrinsically and immediately the weighting of a signal by a scalar. Now, the question we can ask ourselves is: why use a memristor and not a standard resistor? The answer lies in the ability of the synaptic weights of an artificial neural network to evolve when an update signal is sent to them. A simple resistor has its conductance fixed by its geometry and chemical composition and therefore cannot serve as a support for a trainable artificial neural network. Simple resistors could on the other hand be used for a inference-only circuit whose values would be given by a model network trained on a classical computer before.

  However, this approach of transferring the weight from a network trained on a computer to a hardware without the possibility of on-chip fine-tuning of the parameters suffers from a decrease in performance [86], [87]. But above all, the size of the standard resistors is too imposing to be able to be integrated in sufficient numbers on a chip to approach the size of the most performing neural networks. The memristor fills all these gaps. It is a nano-metric component and therefore can potentially be integrated in large numbers on a chip: up to 2.4M [88] and 4M [89] on a single chip and is also low power. A memristor can also see its conductance change under the effect of a certain voltage pattern applied to its terminals, making this component able to support the step of updating the parameters of an artificial neural network and thus allowing the training of a hardware neural network whose synapses would be based on memristors. The "memory" of memristors relies on different physical behaviors depending on the nature of the materials composing the memristor. Some are based on the migration of oxygen atoms [85], [90], [91] to form a conductive filament in the initially very resistive material. We can also quote the Phase Change Memories [92]–[94] based on the same principle of conductive filament formation but with a different physical mechanism.

  Although based on different mechanisms, memristors can be described by a simple linear model that describes the change in conductance that occurs when a potential difference V is applied to their terminals. This model takes into account a particularity of memristors

that is essential to their use as a synaptic element of a hardware neural network. Indeed, the change in conductance of a memristor is highly non-linear in the sense that there is a change in conductance only if the applied voltage exceeds a certain threshold. This allows memristors to be used as synaptic elements during the inference phase under sub-threshold voltage and to be able to modify the conductances during the weight update phase by applying supra-threshold voltages.
So the model illustrating this threshold operation is the following:

$$\Delta G(V) \propto \begin{cases} 0 \; if \; V < V_{th} \\ V - V_{th} \; else \end{cases} \tag{2.3}$$

In practice, the voltage applied during the phase of updating the parameters - and thus the conductance - is done in the form of pulses. These pulses can be stereotyped, in which case the desired change in conductance ($\Delta G \propto -\frac{\partial \mathcal{L}}{\partial G}$) is obtained by applying N pulses - the simplest in hardware since it requires only one voltage source. They can also be of variable amplitude, the pulse voltage being adjusted to modify the conductance of the memristor by the desired amount ($\Delta G \propto -\frac{\partial \mathcal{L}}{\partial G}$). Unfortunately, the model of linear variation of the conductance with the number of applied pulses described above is an extreme simplification far from the experimental measurements. A real memristor sees its conductance change in a highly non-linear way [95] when voltage pulses are applied (Fig. 2.1). Querlioz *et al*. derived a model that fits well with experimental observations:

* To increase the conductance we apply a positive voltage pulse which induces the following change:

$$\Delta G^{+} = \alpha \Delta G(V) e^{-\beta \frac{G - G_{min}}{G_{max} - G_{min}}} \tag{2.4}$$

* To decrease the conductance we apply a negative voltage pulse which induces the following change:

$$\Delta G^{-} = \alpha \Delta G(V) e^{-\beta \frac{G_{max} - G}{G_{max} - G_{min}}} \tag{2.5}$$

where α, β are parameters, often experimentally determined, which depend on the nature of the memristor, G_{min} and G_{max} are the minimum and maximum conductances of the memristor and $\Delta G(V)$ is the same function as in Eq. 2.3.
This non-ideal behavior, where the conductance change curve saturates as the conductance approaches its high or low state, implies designing complex update schemes since if we want to change the conductance by a certain amount $\Delta G \propto -\frac{\partial \mathcal{L}}{\partial G}$, the number of pulses to be applied will depend on the value of the conductance before updating the conductance. Recent works propose sophisticated rules [87] allowing a good learning but which penalize both the energy consumption since the circuit required to compute the update is complex, and the training speed. The ability of a memristor's conductance to change under the effect of an applied voltage will also vary from one memristor to another - this is called *device-to-device* variability. This behavior is totally undesirable [15], [96], since it also implies to revise the conductance update scheme on an individual scale, which complicates the circuit that programs the pulses to be applied to the memristors according to the gradients calculated by a backpropagation type algorithm. Finally, memristors also exhibit *intra-device* variability [16], [97], [98]. This variability at

the device level is also expressed by the conductance change mechanism which in reality does not exactly follow the 2.4 model but is rather stochastic as we can see in Fig. 2.1. Thus, although on average the measured memristor conductance changes according to the model 2.4, locally we can see that some pulses have no effect on the memristor conductance (and others much larger effects than expected). Without a readout mechanism after the application of a pulse, one can think that the conductance of the memristor has been updated but in practice it remains constant. This can penalize the training of a neural network, since this variability affects all the memristors in parallel. Unfortunately, the stochastic gradient descent algorithm works well if all the parameters are updated at the same time so that the global cost function is reduced. If some memristors are updated, others not and others more than expected, the network as a whole will not evolve according to the exact gradient of the cost function computed at the output and therefore this strongly penalizes the training. This intra-variability also comes from the fact that the operation of a memristor is affected after several SET/ RESET cycles that is called *cycle-to-cycle* variability and makes the update scheme of its conductance even more complex thus energy costly. Nevertheless, memristors offer a new and very interesting computation scheme by allowing access to intermediate states between LRS and HRS, contrary to the components usually used for memory (SRAM, MRAM).

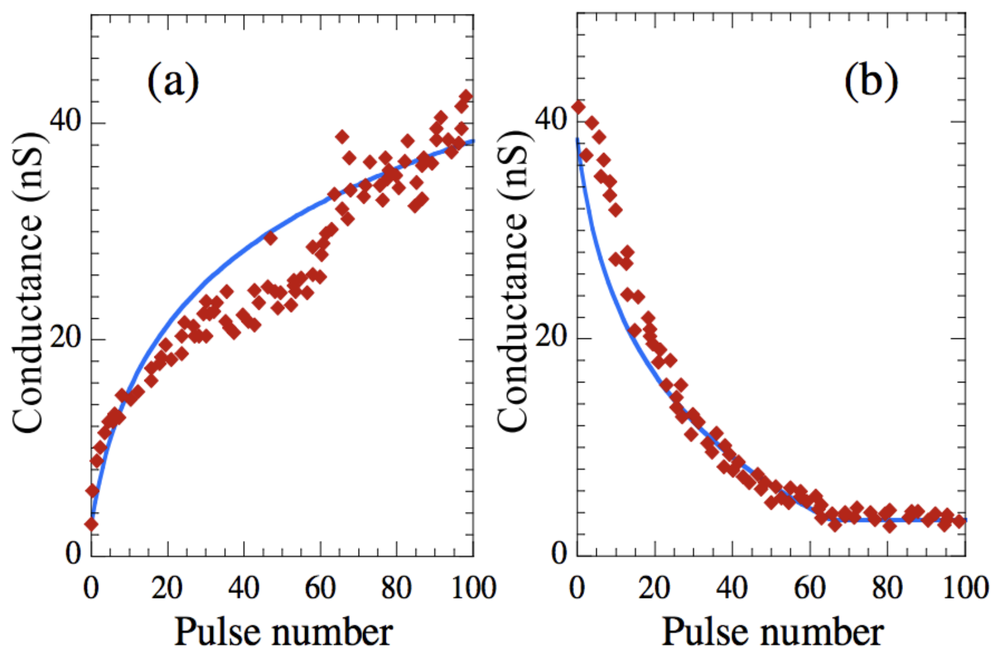


Figure 2.1: Experimental data from [99] fitted with a non-linear model in [15]. The conductance changes when stereotypical voltage pulses are applied to the terminals of a memristor. The relationship between the conductance change and the number of pulses applied is highly non-linear and affects the programming scheme during the update step of the optimization of memristor-based neural networks.

Memristors implement at the individual scale the operation of weighting an input by a scalar. It is difficult to have a non-linear component that emulate a neuron that also does the summation of its inputs. This is why this summation step can be carried over to

the synaptic stage of the hardware implementation. We can associate several memristors with each other, for example by connecting their terminal on the output side so that the current coming out of this terminal is the sum of the currents flowing through the different memristors, as a consequence of the law of nodes. In the same way, the input terminal of several memristors can be linked together so that an input signal crosses several memristors to feed different neurons. This architecture is called a cross-bar array and is widely used for hardware implementations based on resistive elements such as memristors. Eq. 2.6 gives the contribution to the current arriving at the post-synaptic neuron 1 as a function of the voltages V_i representing the activation function of the pre-synaptic neurons and of the synaptic weights encoded as conductance G_{ij}.

$$I_1 = G_{11} * V_1 + G_{12} * V_2 + G_{13} * V_3 \tag{2.6}$$

In Figure 2.2, the input voltages are applied to "word lines", and post-synaptic currents are collected through "bit lines".

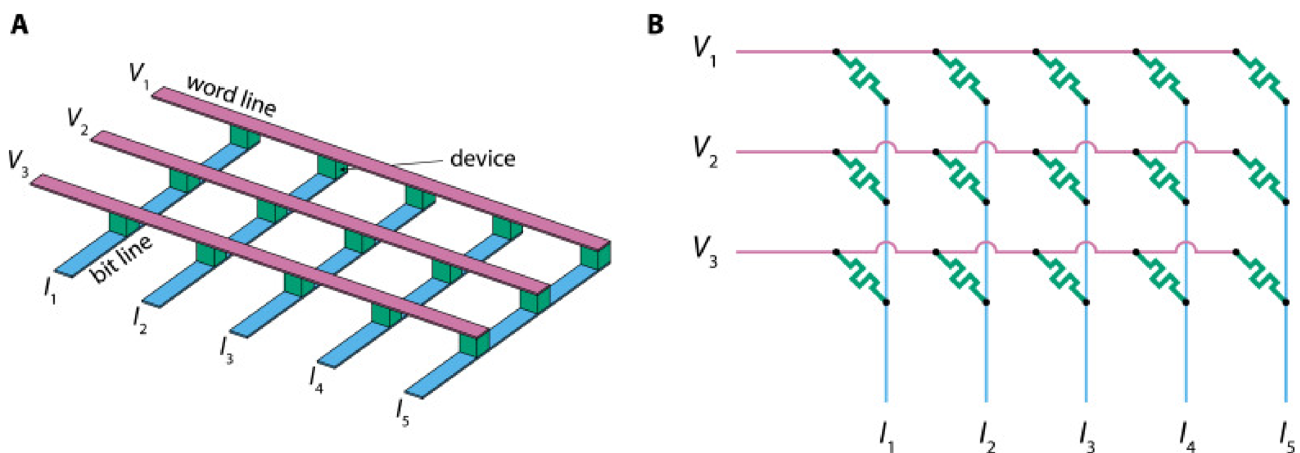


Figure 2.2: Figure of a crossbar array where memristors are located at the intersection of word and bit lines. Input voltages are applied at word lines. The current resulting of the addition of every current in each NVM device connected to a bit line is read a the end of each bit line. Figure from [100]

Other approaches rely on other hardware based on other physical mechanisms. One can cite the use of spintronic resonators [101], [102] where the synaptic weight is encoded in their resonance frequency. This work overcomes the limitation of dense connectivity as it relies on the transmission on RF signals from a layer to another, that could be done in principle without any hardware connections. The resonant frequency can be tuned in a non-volatile way thus making this component a possible challenger to memristors for hardware implementations.

- **Hardware neurons.**

  Now that we have been able to characterize the behavior of hardware synapses with the memristor example, we will see how to implement artificial neurons in hardware.

  We have previously described the functioning of an artificial neuron: it performs the summation of its inputs which have been weighted by the synaptic weights and then applies a non-linearity. We have also seen that the summation step of the weighted inputs can be transferred to the

synaptic stage using the simple Kirchhoff's Law of Nodes if one uses a cross-bar of memristors or the Kirchhoff's Voltage Law if one uses a chain of spin-diode resonators). An artificial hardware neuron must then only have a non-linear behavior according to the signal applied to it. Highly investigated artificial neurons compatible with memristors synapses are CMOS-based leaky integrate and fire neurons [103] or neuristors, which are miniaturized spiking neurons based on volatile of memristors [104]. A nano-scale spintronic component which is the spin transfer oscillator presents a non-linearity that makes it a good candidate to implement a hardware artificial neuron compatible with spintronic resonators.

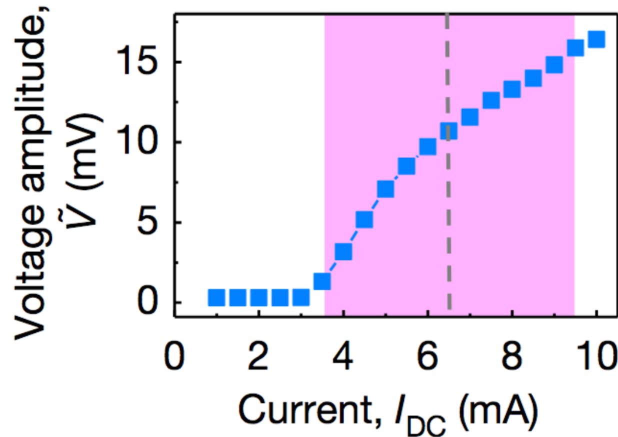


Figure 2.3: Voltage amplitude of the emitted oscillations of a spin transfer nano-oscillator from [105]. The non-linear behavior with respect to the input current makes spin transfer nano-oscillators good candidate to emulate neurons on-chip.

2.1.2 . Using a physical system on its own to perform the non-linear mapping

Another approach in using physical systems for deep learning has been to take the function of an artificial neural network - i.e. the non-linear mapping between inputs and outputs - literally. Indeed, as said before, many physical systems evolve and transform in a non-linear way the inputs that are presented to them. Thus, a physical system to which we can apply control values acting as inputs will be able to perform the non-linear mapping to a measurable quantity or state acting as the output of the system. In this case we do not necessarily need to have the layered architecture of standard neural networks but we also do not need to artificially integrate a dense connectivity between neurons which greatly facilitates the hardware implementation.

A first way to use the complex dynamics of a physical system is reservoir computing (RC) that only trains parameters external to the system [107], [108]. For this type of computation, inputs are sent to a complex physical system that transforms the input data in a non-linear way. This system is often modeled as recurrently, randomly and sparsely connected neurons, but can in practice consists in the coupled non-linear time-dependent variables (nodes) of a physical system. Then, some nodes in the system are read and feed a fully connected classifier that performs classification for example (Fig. 2.4). The weights of the classifier are the only parameters that are trained, through a simple matrix inversion: if the output Y is given by $Y = W * X$ where W are the matrix of weights and X are the values of the read-out nodes, then $W^* = \hat{Y} * X^{-1}$ where $\hat{Y}$ is the corresponding label or target vector for the input. RC has been shown to be effective for some tasks, especially when the data has a temporal dimension, but since it does not create any hierarchy in the data representation,

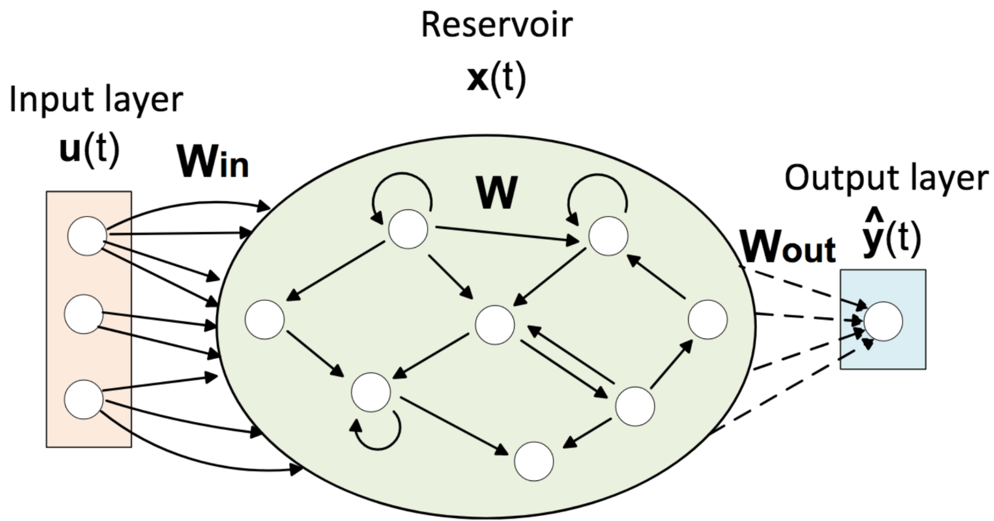


Figure 2.4: Schematic of a reservoir being used for data classification from [106]

its power is limited. Nevertheless, RC could be used as a pre-processing step before feeding the data to a neural network, as it projects the data onto a larger representation space that might be better for training a neural network.

A second and very recent approach [109] aims at building deep physical neural network where the layered architecture of standard neural network is present but where the standard operation performed at each layer (matrix multiplication, convolutional operation, ...) is replaced by the non-linear mapping of a physical system. Now the controls applied to the system are not only the input data but also some tunable parameters that act as the parameters of standard neural networks. The challenge with this kind of network is to compute the parameters updates. To perform backpropagation through the physical system is excluded since the system only achieves the parametrized mapping of the input to the measured output state. The idea of the paper [109] was to use a digital proxy of the physical system to backpropagate the error calculated at the output of the actual physical system (Fig. 2.5).

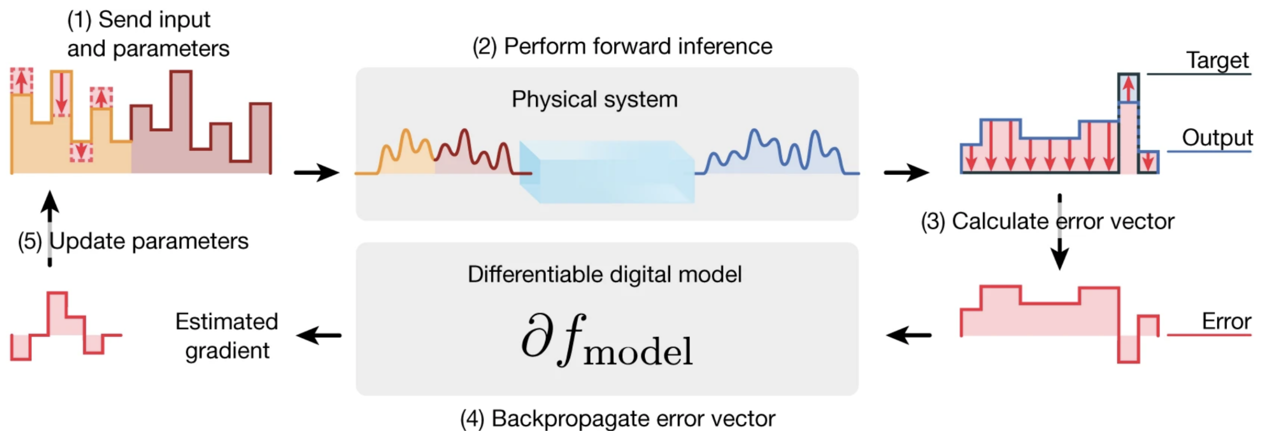


Figure 2.5: Physic-aware training from [109]

The proxy being either a physical model of the system of interest or an artificial neural network

trained to reproduce the behavior of the physical system, it can be used to compute the parameters updates in a supervised way with backpropagation. By using the output of the physical system as input for the same system, mixed with other input parameters, we can build a system emulating several "layers" and thus a "deep" physical system. By revisiting the operations on which a neural network builds on, this approach seems promising, but although there are indications that the stacking of such layers improves the performance of the overall system, however it remains to be demonstrated that this depth does indeed bring out a kind of hierarchy that is the power of deep learning. This approach also does not yet improve the energy efficiency that some physical systems can offer as the digital model is still digital and mostly trained to emulate the physical system with backpropagation.

2.2 . Leveraging the natural tendency of physical systems to evolve toward the ground state to parametrize the computation via the energy function

2.2.1 . A statistical spin physics intuition for using the energy as the parametrizable feature of a physical system

The last section highlighted the interest of using physical programmable systems to perform the calculations on which deep learning is based. We could see that depending on the angle of attack - realizing a hardware neural network or just having the non-linear input-output relation - the control on the system can be done in very different ways.

In this section, we focus on a particular type of control which is the control via the energy function of a physical system. This type of neural network is called energy-based. This type of machine learning model is the model on which the work of this thesis is based. A question immediately comes to mind: why introduce control in the energy function of a physical system? The answer is simple: we take advantage of the fact that a physical system evolves intrinsically towards a state of minimal energy to encode the output of this system in local minima of this energy function. An output can be defined as a transformation of the inputs: initial state, subset of parameters, ... to a state of the system which corresponds to a minimum of the energy function of this system.

Controlling a physical system by its energy function already implies that one can modify parameters that act explicitly on this function and thus directly on the physical system of interest. This point will be discussed in a first step with the review of the founding works for energy-based models. These works have linked coupled spin systems evolving according to the Ising energy function and binary neural networks connected by synapses. The energy function of such neural networks is then the Ising energy function. We will then take the example of Hopfield networks and Boltzmann machines which are energy-based models to demonstrate the interest of such an approach.

Early works on artificial neural networks [9], [11], [110] pioneered a successful approach to parametrize the dynamical systems via an energy function that the system minimizes while evolving.

These first works intimately mixed the concepts of biological neurons, physics and machine learning to produce the first models of artificial neural networks able to perform complex tasks. Until the publication of these works, the formal neuron of McCulloch-Pitts, the perceptron, the linear associative memories, ... had opened the way to non-linear information processing but were limited to simple tasks.

At that time, those works were greatly influenced by the modeling energy-based physical systems. In fact, the input-output relation approximated by a neural network can also be described as obtaining

an output whose probability knowing the input is maximized, or:

$$\begin{gathered} output = f(input, \theta) \\ \Leftrightarrow \\ p(output = y) = \max_{\theta}(p(output|input)) \end{gathered} \tag{2.7}$$

In the ideal case we would like $p(output|input) = 1$. Since the output is part of the neural network, maximizing $p(output|input)$ is equivalent to maximizing the probability that the output neurons are in the y state.

This work has been used as a basis for some types of primitive neural networks that we will detail. This influence can be seen in the way artificial neural network systems are parameterized by an energy function. Indeed, it was natural to use the energy as the parametrizable feature for physics-oriented scientists as the state of physical systems made of many interacting units can be described with the Boltzmann distribution. The probability of being in a given state s_i exponentially depends on the energy of the same state:

$$p(s_i, \theta) \propto \exp^{-\frac{E(s_i,\theta)}{k_B T}} \tag{2.8}$$

where θ stands for a set of tunable parameters that shape the energy function. Now one can use the ability of a physical system to evolve toward states that have high probability according to its energy function - and indirectly to its parameters - to perform the input-output mapping that artificial neural networks do. Inputs could be the initial state of the system or could be some parameters fed to the system at initialization. The equilibrium state reached after some evolution would be on average the state that have the highest probability and thus the lowest energy from all the possible states.

If one would like to do for instance image classification, one could encode the class of an image as the state of specific units or features in the system measured at the equilibrium state.

The central question, then, is what could be the energy function of a physical system and how can we parametrize it?

An initial tentative to introduce parametrization in the energy function of a dynamical system for it to act as a neural network is the work of Little [110]. In this work Little draws a parallel between simplified biological neural networks and spin glasses (the concept of the spin glass system of coupled spins is developed in Section 2.2.2). His work built upon the observation that similarly to spin glasses where there is an long-range order which indicates a correlation between spins at long distances, there exist temporal correlation between spaced neurons. He called it "persistent activity" and hypothesized that such activity could explain how the brain functions. A long term correlation between neurons could have explained a mechanism of memory in the brain. Little tried to find conditions where such persistent activity arises between coupled neurons alike a physicist searches for parameters at which a phase transition occurs in spin glasses. For his work, Little extensively used the nomenclature of spin glasses as his neurons are binary units and the coupling - or synaptic strength - between neurons are symmetrical.

Beyond the mathematical arguments he found that guarantee the convergence of such a system of coupled neurons toward a state that exhibit such persistent activity, this work is a founding work that explicitly draw the parallel between an energy-based physical model - the spin glass - and neural networks.

Despite this stunning work, Little did not include any learning capability or mechanism in his study. His work leverages the analogy between a simplified version of neurons and spins but the

equivalence was not based on the Ising energy of coupled spins. He focused on the dynamical properties of such a system of coupled neurons but did not tackle the power of such systems to perform computation.

After having discussed why using the energy function of a physical system as a parameterizable quantity is interesting for computational purposes, we will now introduce the Ising model which inspired Little, and is an energy-based model by nature. This model, which simply models with an energy function how coupled spins behave, will allow us to see how we can parameterize an energy function by playing on the value of the inter-spin couplings. Once this model is introduced, we will then see two concrete applications of the use of the energy function to make calculations, through the examples of Hopfield networks and Boltzmann machines.

2.2.2 . The Ising Model: coupled spins that evolve collectively on a lattice

The Ising model is a statistical physics tools introduced in 1920 by Ernst Ising and Wilhelm Lenz [111] that has been used to describe the behavior of magnetic materials - initially for ferromagnetism.

The model consists in assigning spins $\sigma_i = \pm 1$ to the vertices of a 1 or 2 or 3 or more-dimensions lattice Λ (Fig. 2.6). Adjacent spins σ_i and σ_j are mutually coupled via the symmetric coupling parameter J_{ij}. One can also apply an individual magnetic field h_i to each of the spins. A configuration σ of spins is $\sigma = \{\sigma_i\}_{i\in\Lambda}$.

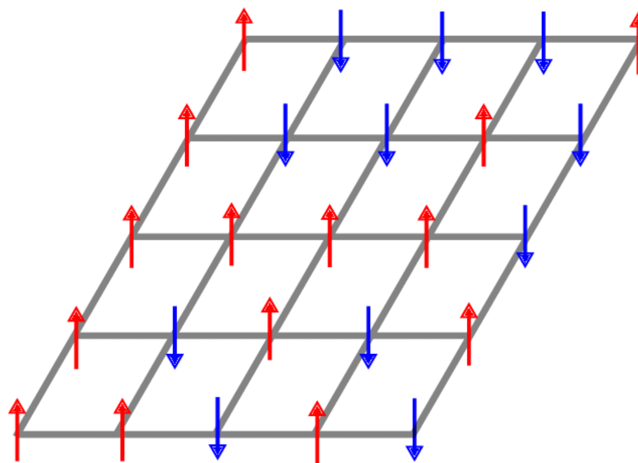


Figure 2.6: Ising model: spins are located at the vertices of a lattice (graph) and are coupled along the edges (local coupling here but it can be more global)

The Ising model describes the orientation of each spin of the system at equilibrium. The equilibrium state can be defined as the state of the spins that minimizes a function that is called the Ising energy and is given by:

$$E(\{\sigma_i\}, \{J_{ij}\}) = \sum_{i\neq j} J_{ij}\sigma_i\sigma_j + \sum_i h_i\sigma_i \tag{2.9}$$

This energy function describes the behavior of two kinds of magnetic materials with different ground states (we omit the magnetic field h_i for simplicity here):

- $J_{ij} > 0$: in this case, E is minimized when the spins σ_i and σ_j are anti-aligned such as $J_{ij}\sigma_i\sigma_j < 0$. Such material is called antiferromagnet and has all its spins anti-aligned on the lattice.

- $J_{ij} < 0$: in this case, E is minimized when the spins σ_i and σ_j are aligned such as $J_{ij}\sigma_i\sigma_j < 0$. Such material is called ferromagnet and has all its spins aligned on the lattice.

In absence of an individual external magnetic field applied to each of the spins, the solutions - or ground states - are degenerate. Even the ferromagnet and antiferromagnet problems are degenerated: if we switch all the spins of the solution, the "switched" set also minimizes the Ising energy function. $J_{ij} * (-\sigma_i) * (-\sigma_j) = J_{ij} * \sigma_i * \sigma_j \Rightarrow E(\{-\sigma_i\}_{i\in\Lambda}) = E(\{\sigma_i\}_{i\in\Lambda})$. We will see later in Section 5 that one can remove the degeneracy of the solution by adding small individual magnetic fields h_i, that can be either randomly set or chosen according to the problem one wants to solve.

2.2.3 . Hopfield networks: encoding patterns to-be-retrieved in the minima of the energy function

Performing computations with such spin-glass-like neural networks came later with the work of Hopfield [9]. Following Little, Hopfield also grounded his work in the analogy between coupled spins of magnetic materials and neural networks. The collective behavior of physical systems constituted of multiple and simple interacting units was indeed an emerging topic of interest. In spin systems, depending on the coupling sign and strength, spins align - or anti-align - toward a stable magnetization that minimizes the Ising energy function (see Section 2.2.2). Hopfield Networks thus take advantage of the ability of a dynamical recurrent neural network composed of numerous simple units that are coupled to evolve collectively to perform computation. Hopfield networks are in this sense Content Addressable Memories[1] (CAMs) that memorize input patterns as a state of the network.

After training, the network can be fed with corrupted patterns - *i.e.* is initialized in a state that corresponds to the corrupted pattern - and then evolves toward the nearest local minimum of the energy function of the network.

A Hopfield network is a recurrent neural network with no self-feedback connections (see Section 3.2 for a detailed description of recurrent neural networks) where alike spins are coupled in the Ising model, are mutually linked with symmetric synaptic weights $W_{ij} = W_{ji}$. The neurons initially had a binary activation function:

$$V_i(t) = \begin{cases} +1 \; if \; \sum_j W_{ij}V_j(t) - \Theta_i \geq 0 \\ -1 \; otherwise \end{cases} \tag{2.10}$$

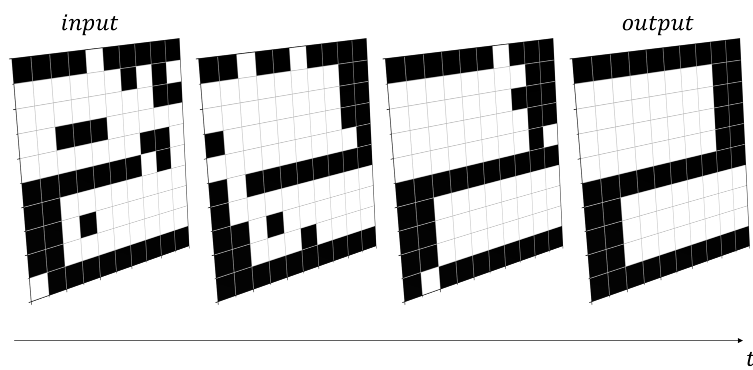
which has been extended to graded - or multi-levels - activation functions in [112]. Neurons in a Hopfield network follow an energy function that is defined as the Ising energy function of the neural network:

$$E = -\frac{1}{2}\sum_{i\neq j} W_{ij}V_iV_j \tag{2.11}$$

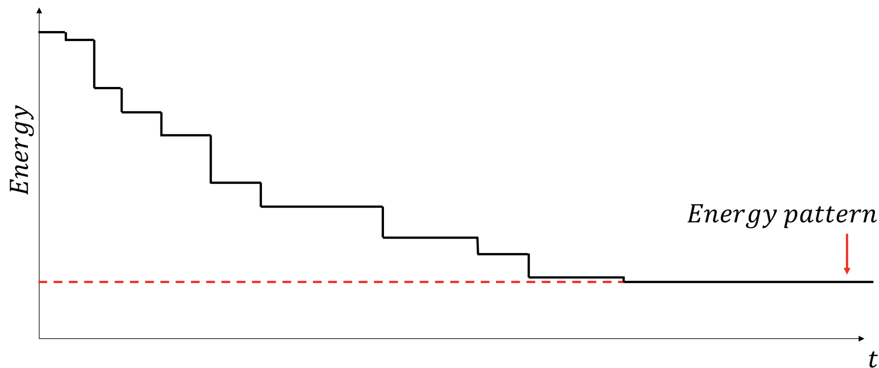
Similarly to spins systems, the neurons in a Hopfield network evolve toward a state that is dictated by the energy function. Moreover this state corresponds to a minimum (local or global) of this energy function. Hopfield showed that the asynchronous updates[2] of neurons guarantee the

[1] Content Addressable Memory is a kind of memory very different from standard memories (like RAM). Whereas a standard memory fetches data according to an address provided by the user, a CAM is addressed by content. That means that a CAM requires a subset of the data to-be-fetched to return the entire data. Hopfield networks are CAMs in the sense of they retrieve pattern from corrupted inputs patterns.

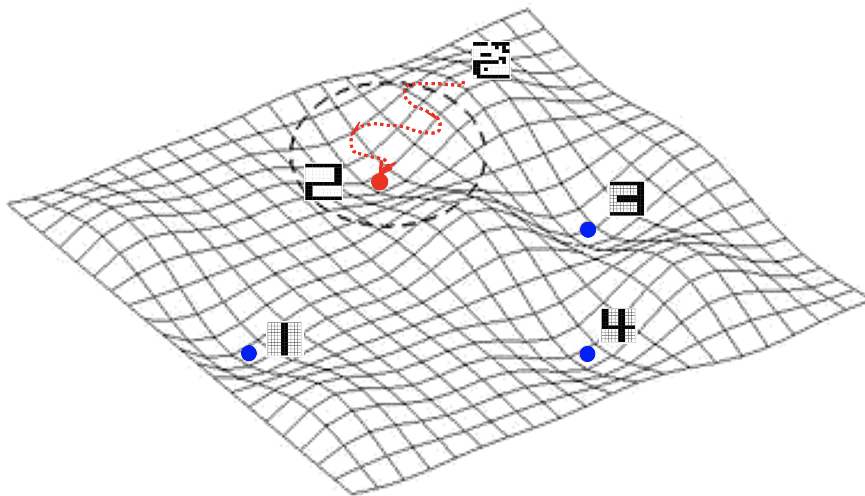
[2] By asynchronous we denote a dynamics where at each time step, a random neuron in the network is selected and its state - or activation - is changed - or not - according to Eq. 2.10



(a) Temporal evolution of the state of the network



(b) Monotone decrease of the energy function driving the dynamics of the network



(c) Despite many local minima encoded, the network converges toward the closest from the input pattern

Figure 2.7: Corrupted patterns that constitute the initial state of a Hopfield network initiate the evolution of the network toward the state that corresponds to the nearest minimum of the energy of the network. This state satisfyingly corresponds, for proper tuning of the parameters, to the non-corrupted pattern.

convergence of the system to a minimum of the energy function. Indeed, if neuron i changes his state at time t from $V_i(t-1)$ to $V_i(t)$, the resulting change in energy is:

$$\Delta E = E(t) - E(t-1) = -\left(\frac{1}{2}\sum_j W_{ij}V_i(t)V_j\right) - \left(-\frac{1}{2}\sum_j W_{ij}V_i(t-1)V_j\right) \quad (2.12)$$

$$\Delta E = -\Delta V_i \left(\frac{1}{2}\sum_j W_{ij}V_j\right) \quad (2.13)$$

We get from Eq. 2.10 that ΔV_i has the same sign as $\sum_j W_{ij}V_j$. So Eq. 2.13 states that each neuronal update in the Hopfield network, the energy decreases - or at least does not increase. As the energy is bounded, it is guaranteed that the system will eventually converge toward a state that is a minimum of the energy function. However it is not guaranteed that this minimum is global. But instead of seeing that as an issue, Hopfield leveraged this ability to encode different outputs in different local minima of the energy function. If one initializes the state of the network as the input value, the system will eventually reach one of those local minima that would be the closest from the initial state.

Contrarily to previous linear associative networks [113], [114], the non-linear dynamics of the Hopfield network better segregates the retrieved pattern from the corrupted input if many patterns have been encoded in the energy function of the system.

Setting the parameters in a Hopfield network. We described above that Hopfield encoded patterns-to-be-retrieved as states that correspond to minima of the energy function. But we have not yet described the procedure used to do this encoding. The procedure is simple and consists in adjusting the weights according to the patterns to store. The learning rule ΔW_{ij} is a Hebbian learning rule [25] that is local as it only depends on the states of the two neurons connected by W_{ij}:

$$\Delta W_{ij} = V_i V_j \quad (2.14)$$

In practice learning is performed with valued $\{V_i\}$ of units corresponding to non-corrupted pattern values. This empirical learning rule shapes local minima in the energy function:

- When V_i and V_j have the same sign, then W_{ij} increases. The energy of the network being given by Eq. 2.11, this results in a decrease of the energy.
- When V_i and V_j have the opposite sign, then W_{ij} decreases. Due to the negative sign in front of the energy of the network given by Eq. 2.11, this results in a decrease of the energy.

By iterating Eq. 2.14 on all the units set at the pattern values, one shapes a minimum in the energy landscape for this specific pattern.

Overall, this procedure allows to minimize the energy of each pattern to store thus the energy landscape contains many local minima that the network will converge to given initial conditions.

In practice we want to store N patterns, so the learning rule is averaged on different patterns:

$$\Delta W_{ij} = \frac{1}{N}\sum_{x=1}^{N} V_i^x V_j^x \quad (2.15)$$

This learning rule allows storing a limited number of patterns. Indeed, Hopfield networks are known to have a low capacity which means that they can typically store $0,14n$ patterns where n is the number of neurons. Furthermore, the learning rule is empirical and does not optimize an objective function which limits the performance of such networks.

Finally, Hopfield networks are auto-associative only memories as they retrieve a pattern that has the same size as the input. This can be a limitation for some applications where the size of the output does not correlate directly to the size of the input. Kosko [10] extended Hopfield networks to Bi-directionnal Associative Memories (BAM) where the network is composed of two sets (A and B) of neurons that are symmetrically coupled with a connectivity matrix M. The units in both of the sets are evolved. This time, the two sets do not need to have the same size and thus one can retrieve an output whose size differs from the input size. The weights are set in a way similar to Hopfield networks:

$$M = \sum_{i}^{N} A_i^\top B_i, \tag{2.16}$$

where the connectivity matrix is the superposition of the correlation between the exact input A_i and the expected output B_i. This learning rule remains empirical and does not optimize an objective function. Next we will describe an algorithm to train such networks that optimizes an objective function and thus improve their capacity.

2.2.4 . Boltzmann machines: tuning the energy function to approximate the data distribution and generate similar inputs

Soon after the initial paper of Hopfield, Kirkpatrick *et al*. [115] came up with the idea to add controlled temperature to the dynamical system in order for the system to be able to escape bad local minima thanks to thermal activation. Kirkpatrick *et al*. showed that their procedure, called Simulated Annealing (see Section 3.1 for detailed explanation about Simulated Annealing), where the temperature is slowly decreased from an initial, large one to a small one allows the system to settle to an equilibrium state that has a lower energy than without the procedure.

Building on both Hopfield networks and Simulated Annealing, Hinton and Sejnowski [11] invented the concept of Boltzmann Machines (BMs). BMs are a kind of stochastic Hopfield network that contrarily to HN have two kinds of units: the visible and the hidden units. BMs are used as generative models that aim at approximating the distribution of the dataset on which they are trained. BMs were initially networks where all units were coupled to each others making the sampling of the model intractable in a reasonable amount of time (see Section 3.1 for discussion about NP-hard problems). Thus Smolensky restrained the BMs to have connections only between the visible and the hidden units, intra-layer connections were excluded. This is the Restricted Boltzmann Machine. We will describe RBMs as they are the BMs used for practical implementations.

The hidden nodes of the RBMs act as feature extractors that are jointly used to approximate the data distribution and used to reconstruct the input vector. The main idea with RBMs is to train the hidden layers to represent latent factors in the input data that explain best the data. Then, based on a sample of the input, the network will reconstruct the entire input based on the state of the hidden layer. RBMs are related to Hopfield network as they "retrieve" the input after training solely based on partial input data but RBMs are trained to approximate the data distribution instead, so there is an objective, rather than HN that have their weights being set by the patterns to be stored. The fact that RBMs optimize an objective function make them more powerful than HN for applications. Furthermore, the learning rule of a Boltzmann machine is simple and local. It approximates the

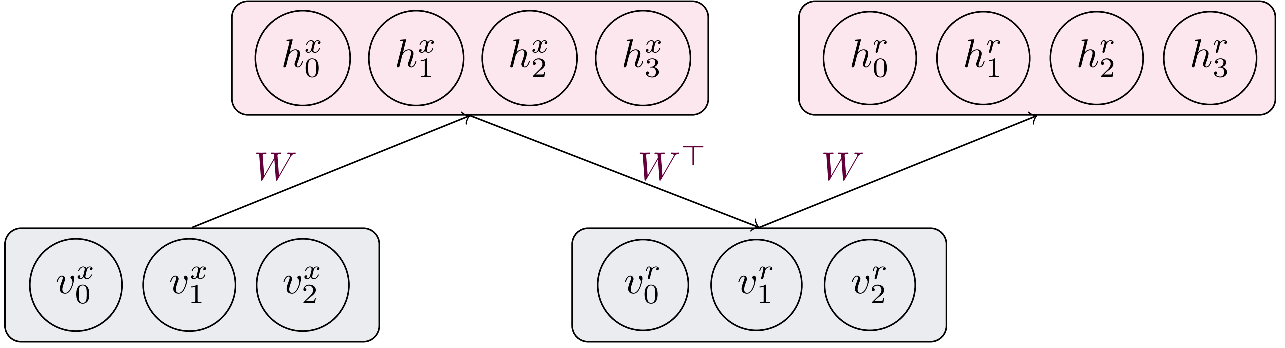


Figure 2.8: Training a Restricted Boltzmann Machine. Visible nodes are in light gray, hidden nodes are in light pink. The visible and hidden nodes are coupled via the connectivity matrix that allows fully connectivity between each layer.

gradient of the log-probability of a given state for the machine. It has been shown to be equivalent to minimizing the Kullback-Liebler divergence [32] that describes the discrepancy between the data distribution and the learnt distribution of the BM. In fact, training BMs computes and minimizes the difference of two divergences. The first divergence is the divergence of the distribution of the model given an input $\mathcal{Q}^0$ - *i.e.* the visible nodes are clamped to the input values - and the distribution of the model itself when the visible nodes are not clamped and the system is let to evolve an infinite time until reaching equilibrium $\mathcal{Q}^\infty$. The second divergence is the divergence between the distribution of the model when one step of evolution is done towards equilibrium ∞^∞ and the distribution at equilibrium $\mathcal{Q}^\infty$. This procedure has been named "Contrastive Divergence" and by minimizing Contrastive Divergence, we want the model to have its distribution at equilibrium as close a possible to the data distribution.

Neurons in RBMs are stochastic binary units. In BMs, neurons are randomly updated - thus the exponential time required to reach equilibrium - and the machine is expected to reach an equilibrium state s^* whose probability is given by the Boltzmann distribution:

$$P(s) = \frac{1}{Z}e^{-E(\beta \cdot s^*)} \tag{2.17}$$

BMs have a two-phases learning procedure following the definition of the Contrastive Divergence we want to minimize over training.

The first phase aims at sampling the distribution of the model given the data so for this phase, the visible units $\{v_I\}$ are set to the input values (the input being binarized). Then the hidden units $\{h_i\}$ are updated until equilibrium is reached. RBMs accelerate $time-to-equilibrium$ compared to BMs as there is no intra-layer connectivity so the probability of a neuron being on or off does not depend on the state of the other hidden neurons. Thus, the state of the hidden layer can be sampled in one step with the following rules.

Each hidden neuron h_i receive an input x_i that is a linear combination of the states of the visible nodes:

$$x_i^h = \sum_j W_{ij} v_j + b_i \tag{2.18}$$

Contrarily to neurons in a Hopfield network, the binary state of the hidden neurons is set according to a given probability that depends on the input x_i. The probability is given by the sigmoid function:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.19}$$

So node i is set to 1 with a probability $\sigma(x_i)$:

$$p(h_i = 1) = \sigma(x_i^h) \tag{2.20}$$

That is the first phase of training a RBM where we get a hidden state that depends on the input data - as visible nodes are set to the data value. The first step allows to sample $\mathcal{Q}^0$.

The second step consists in driving the system one step closer to the equilibrium distribution. The visible nodes, which no longer clamped, are first updated given the states of the hidden nodes:

$$x_i^v = \sum_j W_{ij} h_j + b_i \tag{2.21}$$

and

$$p(v_i = 1) = \sigma(x_i^v) \tag{2.22}$$

This step allows the system to "reconstruct" the input given the vector of latent features that is the state of the hidden nodes.

The hidden nodes are then also updated given the new state of the input nodes with the same rules as Eq. 2.20.

As the procedure of Contrastive Divergence leads the system to approximate the distribution of the data, it is expected that the visibles nodes reconstruct a close version of the input when the training procedure ends, leading to the property of generation of BMs.

The learning rule resulting from this procedure is simple and local:

$$\Delta W_{ij} \propto < v_i h_j >_{data} - < v_i h_j >_{model} \tag{2.23}$$

where the first term subscripted by $data$ refers to the visible and hidden nodes state when the visible node are clamped at the data value. And the second term subscripted by $model$ refers to the state of the visible and hidden nodes after one step of reconstruction.

RBMs can be stacked to build a Deep Belief Network [33], [117] that is trained layer-wise with the same procedure as described above. The hidden state of each layer is the input for the next layer. It has been shown that some level of hierarchy emerges for such networks despite the fact that no global objective function is optimized, but the features learnt with the greedy layer-wise training are less expressive than when a global objective is optimized [118]. Deep Belief Networks are however great feature extractors. Indeed, the hidden nodes of RBMs function as feature extractors that allow to reconstruct the input given features that are activated or not. By stacking multiple RBMs, we add new levels of abstraction on the input data that hopefully better explain the input data. However, in order to perform classification, one has to add an output layer of weights trained with backpropagation along with the whole network made of stacked RBMs used as a feed-forward model. DBNs have been shown to be able to learn CIFAR-10 with a convolutional setting despite an accuracy below that reached with a global obejctive minimized with backprop (the best performance of a RBM on Cifar-10 is $\approx 80\%$ test accuracy whereas backpropagation achieves close to 100% test accuracy).

However, it has been shown [32] that the learning rule Eq. 2.23 does not follow the gradient of the Contrastive Divergence function. Still, training RBMs is unsupervised and does not require labels for the input data until one wants to fine-tune the whole network with backpropagation.

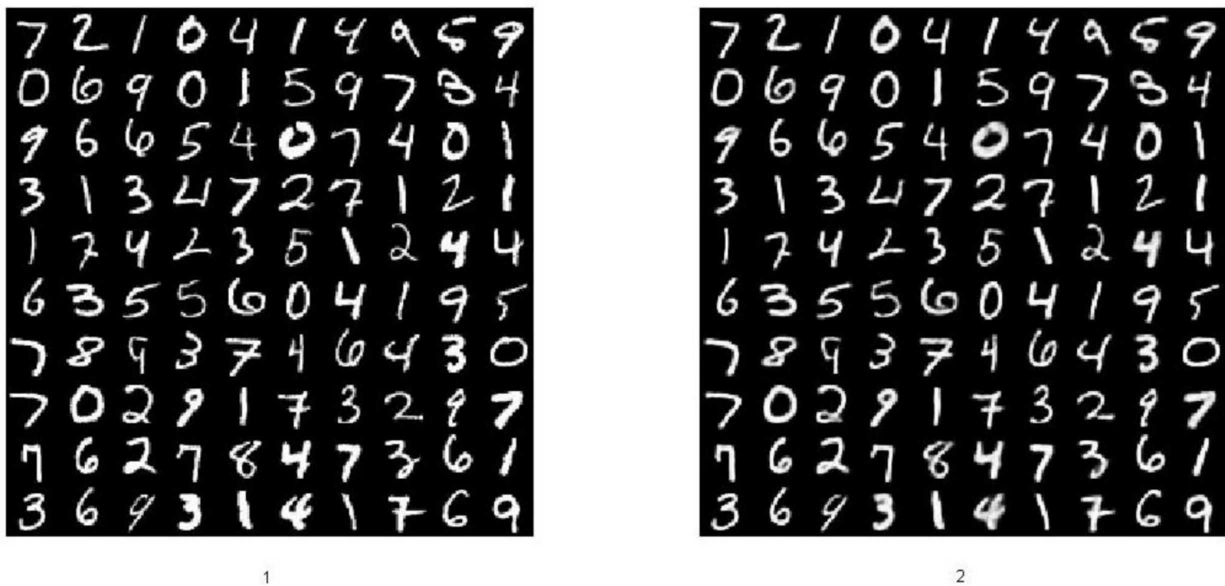


Figure 2.9: 1°) MNIST data used for training an RBM. 2°) MNIST data generated at the visible nodes of the same trained RBM. From [116]

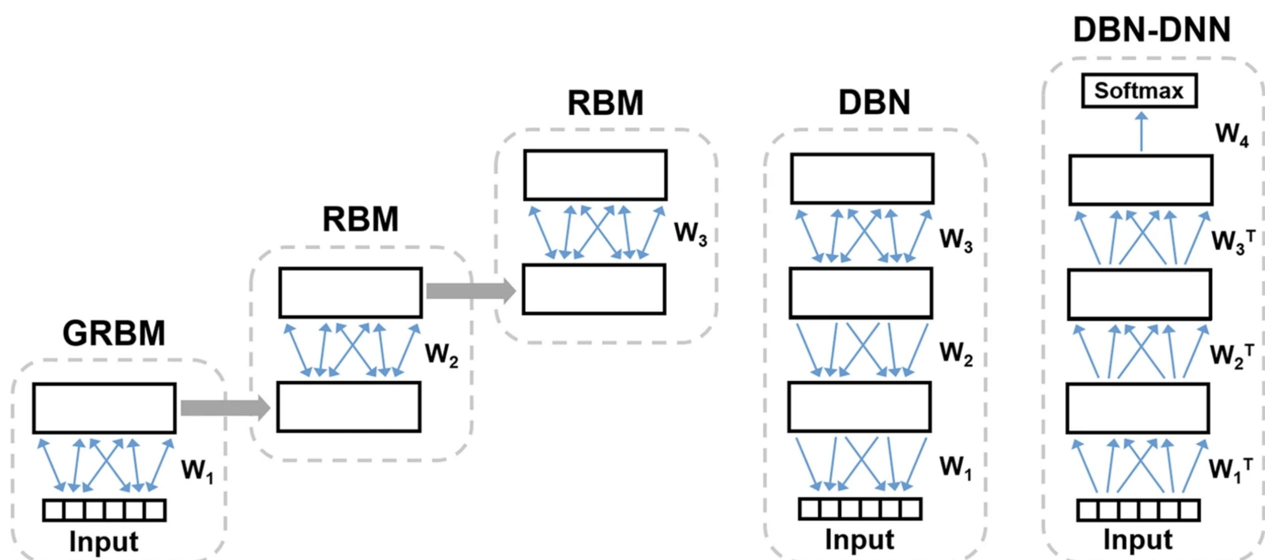


Figure 2.10: Deep Belief Network (DBN). Each layer is trained as a RBM. All layers are then stacked to form the Deep Belief Network that is fine-tuned with backpropagation, to be then used as a pure feed-foward Deep Neural Network in order to perform classification for instance. From [119]

2.2.5 . Conclusion

Along the two last sections we have seen that neural networks and physics are intimately entangled. Using standard components in an unconventional regime or emerging devices to compute offer a new low-power and potentially fast perspective for the field of deep learning. We could obtain orders of magnitude reduction in terms of energy consumption by using the natural tendency of those physical systems to evolve toward states that minimize their energy function. We introduced

the parametrization of that energy function to drive the computation performed when the system relaxes to equilibrium.

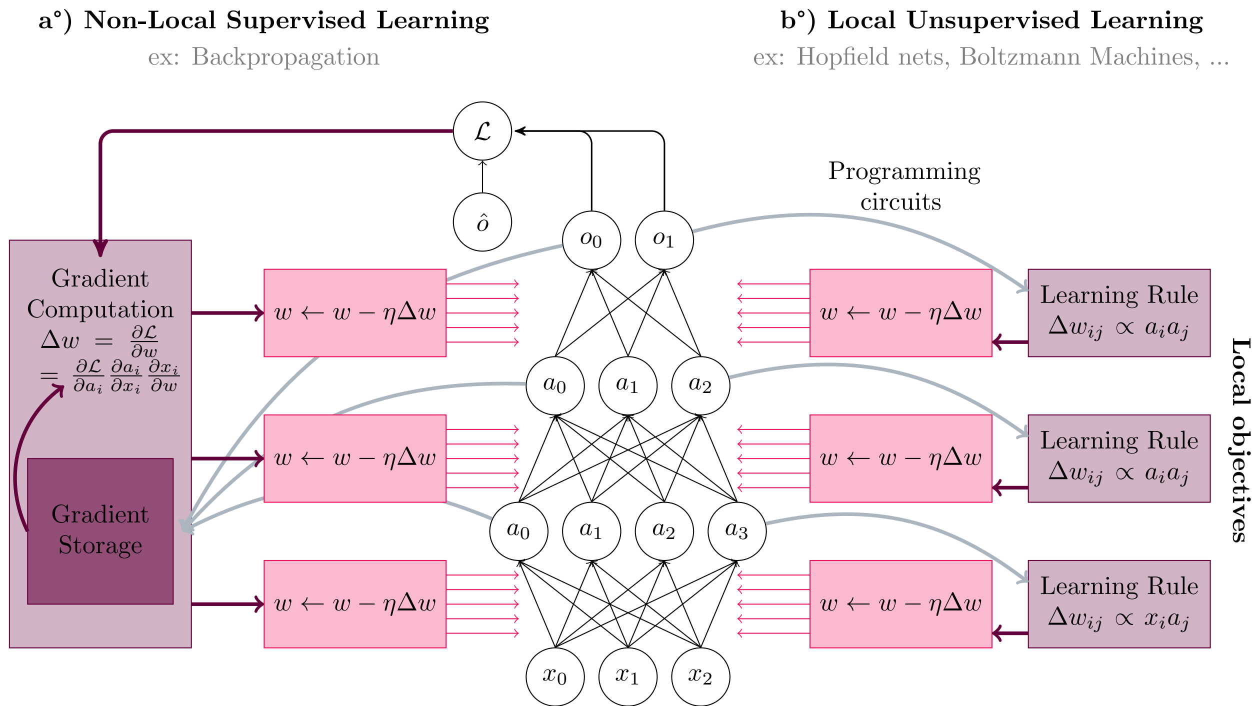


Figure 2.11: Circuitry dedicated for optimization based on a°) a global objective . b°) a local objective or heuristic approach

The algorithms that have been derived to train energy-based systems optimize local objectives (Fig. 2.11), at the layer level for Boltzmann Machines for instance. Those local learning rules are really attractive for hardware implementation as they require only little circuitry to compute and apply the updates to the parameter of the hardware network (Fig. 2.11).

However, we have already discussed that this kind of optimization does not lead to state-of-the-art results due to the non-coherent optimization where no global hierarchy of the features detected at the input emerges. Gradient-based learning algorithms that optimize a global objective function are today the incontestable algorithms for doing this global optimization but they require a lot of memory to store every intermediary operation done during the inference phase (Fig. 2.11) and expensive computing resources to compute the chain-rule of derivation.

Finally, a crucial step to master when it comes to realizing the actual hardware network is to be able to apply the good parameter update, whether the optimization step is done with local learning rules or a global optimization with backpropagation for instance. Indeed we have seen that emerging devices that are good candidates for hardware neural networks such as the memristor exhibit intra-device and device-to-device variability that, if not carefully treated, can preclude on-chip training.

Hence, the challenge to overcome for a neuromorphic chip designer: design a low-power and fast chip but with limited learning capabilities or a chip that reach state-of-the-art accuracy but that is energy consuming and slow, both require to master the programming step of the devices.

2.3 . Summary

We have shown that parametrizing physical systems via their energy function - a quantity that they minimize naturally by evolving towards a state of equilibrium - is interesting from a hardware point of view, in particular with local learning rules. These learning rule unfortunately do not allow the optimization of a global objective and thus, in principle, do not allow the emergence of a hierarchy of the characteristics detected on the inputs.

This is the great current dilemma concerning unconventional hardware implementations for training neural networks on chip. A chip designer must then choose between high accuracy on a task but with a complex update computation system that consumes a lot of energy and memory, and lower accuracy but achieved with a system that consumes little energy both for inference and for the calculation of parameter updates: Fig. 2.11.

This dilemma explains why hardware neuromorphic approaches have, for the moment, made the compromise of having local and simple learning rules that do not reach SOTA results but offer a strong computational and energetic simplicity due to their locality and based on the simple state of neurons, and not on other operations such as the derivative of the state.

The last black spots of implementations based on non-conventional physical components are the variability and noise that these components have due to their still very experimental nature. These defects alter the performances that such systems can theoretically achieve. Thus, complex peripheral systems are used to mitigate these defects but in return they consume a lot of energy.

These two problems were the starting point of the work carried out during this thesis that we will describe in the following chapters.

Chapter 3

Introduction to Ising Machines, Equilibrium Propagation and Binary Neural Networks

Before diving into the work done during the thesis, we will describe in this chapter the key concepts useful for understanding the following work.

First, we will introduce the concept of Ising Machines. We have seen in Section 2.2 that systems of coupled spins are model systems for neural networks. Indeed spins collectively evolve toward a state that minimizes an energy function that we can parametrize through the couplings similarly to synaptic weights in order to drive the system to do interesting computation. Ising Machines are hardware that emulate a system of coupled spins and aim at sampling the ground state of a given configuration set by the tunable value of couplings. Ising Machines seem to be the perfect hardware to train energy-based neural networks. However we have already discussed the impossibility to get high accuracy on standard tasks with energy-based models because the optimization is not global. Combined with some technical constraints that used to prevent fast reconfigurability of the couplings, this has precluded to do supervised learning on that kind of hardware up to now. Nonetheless, Ising Machines have been applied to an extremely different kind of problem which is solving combinatorial optimization problems. Such problems boil down to finding the set of coupled discrete binary variables that minimize an objective function given by the problem to solve. It is important to to have fast solutions of these problems as they are ubiquitous in our daily-life: logistic, planning, chip design,... But such problems are known to be NP-hard which means that no solving algorithm scale polynomially in time with the size of the problem. Fortunately, as the Ising problem is NP-complete, any objective function of that kind can be mapped polynomially to the Ising energy function that Ising Machines minimize, thus the interest of using Ising Machines to solve those problems. There exist several Ising machines based on different physical substrates, and to highlight the potential of such hardware to train neural networks, we will review existing implementations regarding to the simplicity to update the parameters of the machine.

Next, we will introduce the key learning algorithm for this thesis: Equilibrium Propagation (EP) [14]. EP is the first learning algorithm to do supervised learning in energy-based models such as the Ising Machine, hence its importance. As mentioned in the previous section, non-conventional hardware implementations have long lacked an algorithm that combines supervised learning with a local learning rule. EP, introduced in 2017, is an algorithm that proposes to fill in this gap. EP leverages the property of physical dynamical systems to converge to a fixed point - given a static input - to perform both the inference phase and the error backpropagation phase with the same dynamics. Thus, EP reduces by a large margin the peripheral circuitry required to compute the gradient of the global cost function of the system of interest and is a good candidate for low-power on-chip supervised learning. However, only a small experimental implementation of EP exists [120] and still based on standard hardware that could not scale to a larger task without area and energy consumption overhead, where emerging nano-devices could. In fact, as mentioned before, we do not see yet large scale hardware implementations made of nano-devices because their are highly variable, exhibiting both intra-device and device-to-device variability that limit their precision and prevent the on-chip training of that hardware. One way to overcome this issue is to take inspiration from

works that reduce the size of the deep learning models by quantizing the parameters of the neural network. So finally, we will develop in more details the concept of binary neural networks already introduced in the previous chapter which is the extreme case of quantized neural networks. Binary neural networks are artificial neural networks where the synapses and/or the activation function of the neurons can be binarized, *i.e.* take the value 0/1 or -1/+1. Binary neural networks have found a wide application where the memory and computation budgets are limited (embedded applications for example). The optimization of such networks where the cost function is discretized is not straightforward and we will detail the mechanisms developed to allow the supervised optimization of such binary networks. Then, we will describe some hardware implementations of BNNs with either mature CMOS-technology or emerging nano-devices. Finally, BNNs are very similar to some extent to the systems of coupled spins from which the Ising Machine samples the ground state, hence the interest of merging EP, BNN and IMs to demonstrate a large-scale implementation of a physical energy-based system trained with EP.

3.1 . Ising Machines: hardware minimizers of an energy function

Until now, we have focused our efforts on describing attempts to solve problems with machines learning techniques. Especially, we described how physicals systems can behave as neural networks. The example of Hopfield networks is the most striking.

But as well as Hopfield networks are Ising systems where we update the couplings in order to store patterns to be retrieved, in this section we extend the power of such Ising systems by describing another way to use them for computing. For this computing scheme, the couplings are fixed and depend on the problem to be solved and the solution is given by the state of the spins at equilibrium. Such problems are called "combinatorial optimization problems" and are ubiquitous and cover a wide range of practical uses: logistic planning, chip layout design, portfolio management, ...

In this context, combinatorial optimization can somehow be related to the optimization of a neural network. However, instead of using the dynamics of the system to compute the updates required to minimize an objective function as proposed by EP (Section 3.2), we use here the Ising system to solve the combinatorial problem by itself, as the solution of the problem is directly the equilibrium state of the system.

Generally, solving a problem belonging to this class is unsolvable in reasonable time even with deep learning, thus the name of such problems "NP-hard problems", where "NP" stands for "Non-Polynomial" as no algorithm exists to solve those problem with a time that scale polynomially with the size of the input, at least not on deterministic machines. It turns out however that there exist polynomial correspondences between the different NP problems [121]. Luckily, one combinatorial optimization problem is finding the ground state of an Ising system that is grounded in physics and some physical systems naturally solve it - *i.e.* evolve toward a state that minimizes its energy. This behavior is expected to fasten the time-to-solution of those NP-hard problems. And there exist different hardware implementations of such controllable Ising systems that are called Ising Machines.

For this section, we will rely on the terminology of the Ising model introduced in Section 2.2.2. In the following subsections we will describe a bit more what is a NP-hard problem and take the example of the Max-Cut problem to illustrate such classes of problems. We will also link the resolution of a NP-hard problem to the resolution of an Ising problem - property on which lies the power and attractivity of Ising Machines. Finally we will browse a panorama of existing hardware implementations of Ising Machines and discuss the facility of updating the couplings on each machine, investing whether we could emulate neural networks - where the updates of the parameters are numerous - such as the Hopfield network on those machines.

3.1.1 . Solve an Ising Problem, solve them all

With Hopfield networks we saw how local minima of the Ising energy could encode patterns to retrieve. Here we focus on getting the global minimum or ground state of the Ising energy. Despite the fact that physical systems can be stuck in local minima, the solution of a combinatorial problem is unique and corresponds to the ground state of the corresponding Ising energy.

Solving an Ising problem: Finding the ground state when all couplings J_{ij} have the same value, or at least the same sign (ferromagnetism, antiferromagnetism) is an interesting problem and has led to striking results in condensed matter physics in the last century.

However, for computing, a more promising case is when the couplings of the Ising energy function are all different with possibly different signs. This kind of problem - getting the ground state of the energy function - is in practice intractable in a reasonable amount of time (polynomial) [122]. The

time to solution scales in a power law with the number of spins N in the problem:

$$T_{sol} \propto O\left(2^N\right) \tag{3.1}$$

In the terminology of Ising Machines, solving an Ising problem means getting the set of spins that minimizes the Ising energy function. Getting the exact ground state of an Ising problem is intractable in a reasonable amount of time with computer simulations. But depending on the application, solving the Ising problem can be reduced to getting an approximated solution whose energy is close enough to the desired energy. This method can speed up the time to solution for computer-based simulations.

The alternative to computer-based simulations is to use an Ising Machine that given some coupling parameters and a procedure that drives the system toward the ground state (mostly preventing the system to be stuck in local minima) will eventually evolve toward a state that minimizes the Ising energy function. But, depending on the implementation, the offered connectivity and the driving procedure, the system might end up in a local minimum more or less close to the ground state. Since the configuration of the system follows the Boltzmann distribution $p \propto e^{-\beta E}$, we can hope to get the ground state by sampling multiple times the Ising Machine to get the state with the maximal probability. It should correspond to the state with the minimal energy. This "multi-sampling" methods is allowed by the fast time-to-solution Ising Machines offer to solve Ising problems. It is also a good way to harness the stochastic nature of some Ising Machines.

Mapping a NP-hard to an Ising problem: Solving the Ising problems for physics-driven goals such as the ferro or antiferro-magnetic cases has emulated lot of works in the past century, it was the main motivation for building Ising Machines. Most efforts where on the statistical physics side as most physics-driven Ising problem have identical couplings and thus we can derive the partition function of the system. From there we can study the equilibrium states, the free energy, investigate whether there is a phase transition, ...

The main motivation for building Ising Machines is due to [122]. Barahona has shown that there is a polynomial equivalence between NP-hard problems, and most strikingly, between NP-hard problems and the NP-hard problem that is solving an Ising problem. That means that if one can reformulate the objective function of a given NP-hard problem to a corresponding Ising energy function, then if one can minimize this energy function, one can solve the initial NP-hard problem. This is why having fast and accurate Ising Machines has been researched for years. Fully-re-programmable Ising Machines can, in principle, solve any NP-hard problem depending on the size of the problem that can be mapped onto the machine.

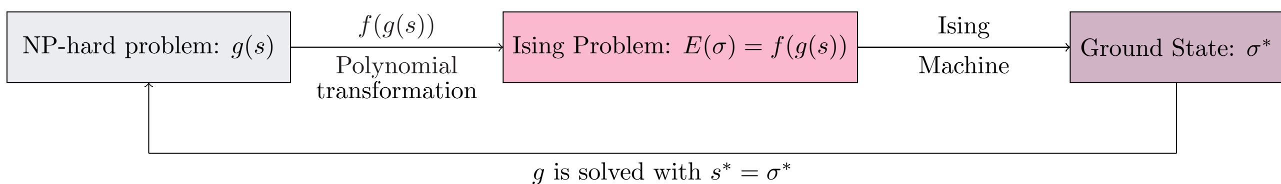


Figure 3.1: Mapping and solving a NP-hard problem with an Ising Machine

In the next section we take the example of solving the NP-hard problem that is the Max-Cut problem by mapping the objective function of the Max-Cut problem to the Ising energy function. This example will make obvious the interest of using Ising Machines.

Example of solving the NP-hard Max-Cut problem: Max-Cut is a problem where one wants to "cut" an undirected weighted graph G in two subsets S and $G\backslash S$. S is called a "cut".

An undirected graph is a graph composed of vertices V that are spins (can take ± 1 as a value) and edges E that are bi-directional symmetric weights between two vertices. The subset S contains both the vertices in S and the edges connecting those vertices. For clarity we denote V^+ the vertices that belong to S (carrying a spin 1) and V^- the vertices that belong to $G\backslash S$ (carrying a spin -1).

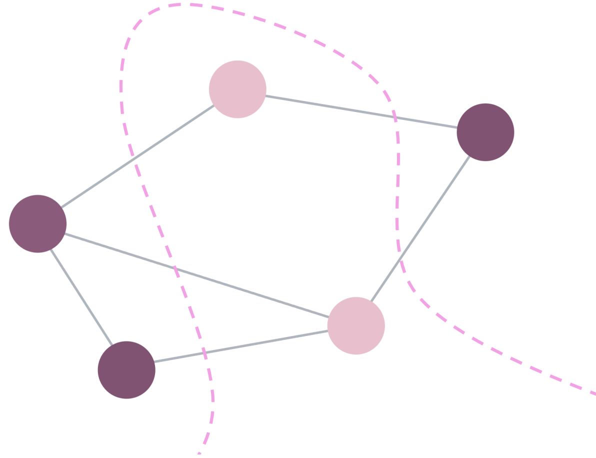


Figure 3.2: Illustration of the Max-Cut problem. All couplings are set at equal value. Here the Max-Cut is the cut that maximizes the number of edges between S (Purple nodes) and $G\backslash S$ (Pink nodes)

The size of the cut S is $C(S)$ and is the sum of the weights of the edges connecting S and $G\backslash S$:

$$C(S) = \sum_{i \in V^+,\, j \in V^-} J_{ij} \tag{3.2}$$

The Max-Cut problem is solved when $C(S)$ is maximized. A special case if when all the weights have the same value. Then the max-cut also maximizes the number of edges between the two subsets. Max-Cut is known to be a NP-hard problem. Here we describe the Max-Cut problem because its resolution is directly linked to the resolution of an Ising problem - *i.e.* the minimization of the Ising energy function.

We now want to rewrite Eq.3.2 as a function the total energy of the system - that is in fact the Ising energy of the graph G:

$$C(S) = f(E(G)) \tag{3.3}$$

We start by writing the Ising energy of the whole graph G:

$$E_{Ising}(G) = \sum_{i,j \in G} J_{ij}\sigma_i\sigma_j \tag{3.4}$$

We decompose the energy function with the contribution of each subset: V^+, V^- and $\delta(V^+, V^-)$ which is the set of edges connecting V^+ and V^-. The energy function now reads:

$$E_{Ising}(G) = \sum_{i,j \in V^+} J_{ij}\sigma_i\sigma_j + \sum_{i,j \in V^-} J_{ij}\sigma_i\sigma_j + \sum_{i,j \in \delta(V^+, V^-)} J_{ij}\sigma_i\sigma_j \tag{3.5}$$

The two first sums can be simplified because for both, the product $\sigma_i\sigma_j$ is always equal to 1. Conversely, for the last sum, $\sigma_i\sigma_j = -1$ as σ_i and σ_j belong to the two different subsets so they have opposite signs. Eq. 3.5 simplifies as:

$$E_{Ising}(G) = \sum_{i,j\in V^+} J_{ij} + \sum_{i,j\in V^-} J_{ij} - \sum_{i,j\in\delta(V^+,V^-)} J_{ij} \tag{3.6}$$

The first two sums can be aggregated in a single sum on the whole graph to which we subtract the edges between the tow subset (i.e. we subtract the third sum of Eq. 3.6):

$$E_{Ising}(G) = \sum_{i,j\in G} J_{ij} - 2\sum_{i,j\in\delta(V^+,V^-)} J_{ij} \tag{3.7}$$

The size of the cut $C(S)$, defined in Eq. 3.2, has appeared as the third term if Eq. 3.5 and the second term in Eq. 3.7:

$$E_{Ising}(G) = \sum_{i,j\in G} J_{ij} - 2C(S) \tag{3.8}$$

If we omit the first term that is now a constant - it does not depend on the state of the vertices $\{\sigma_i\}_{i\in G}$ of the graph G - then we immediatly see that minimizing $E_{Ising}(G)$ is equivalent to maximizing the size of the cut $C(S)$. So in that case, we have reformulated the Max-Cut problem into an Ising problem that is solvable with an Ising Machine.

3.1.2 . Panorama of existing Ising Machines

Ideally we would like to solve an Ising Problem - *i.e.* get the ground state of the Ising energy with a given set of couplings - with a "true" Ising systems *i.e.* real spins with tunable couplings. In practice it is not easily feasible because the couplings between spins in a bulk material is given by the material properties and are not tunable at the individual scale and it would be difficult to go beyond nearest-neighbor interactions and thus would constrain the type of suitable problems.

One solution is to emulate such a system by encoding the spins values (±1) in a physical property of some physical systems. Such property has to be sensitive to an external signal in order for us to control its couplings. Such hardware systems aiming at solving an Ising Problem are called Ising Machine. But because they are not "true" Ising systems, they might not evolve naturally toward the ground state of the Ising energy function. To ensure the convergence toward this ground state, one has to apply to the system a dedicated procedure. Depending on the architecture and the media of the implementation, Ising Machines can have limited or all-to-all connectivity.

Simulated Annealing

Before describing actual hardware Ising Machines, we will pause and talk about an algorithm that has been key for getting the ground state of the Ising energy function in software simulations and is still used in many different Ising machines implementations.

Before Ising Machines existed, the only possibility was to use software simulations to solve a combinatorial optimization problem. If one wants to simulate an Ising system that minimizes its Ising energy, then one has to drive the system with a certain dynamics. The numerical simulation does not evolves by itself towards the ground state of the Ising energy function.

The naive and straightforward way to minimize the Ising energy function is to use the Glauber dynamics [124] to simulate the dynamics of the network of coupled spins, as detailed in Alg. 1. The

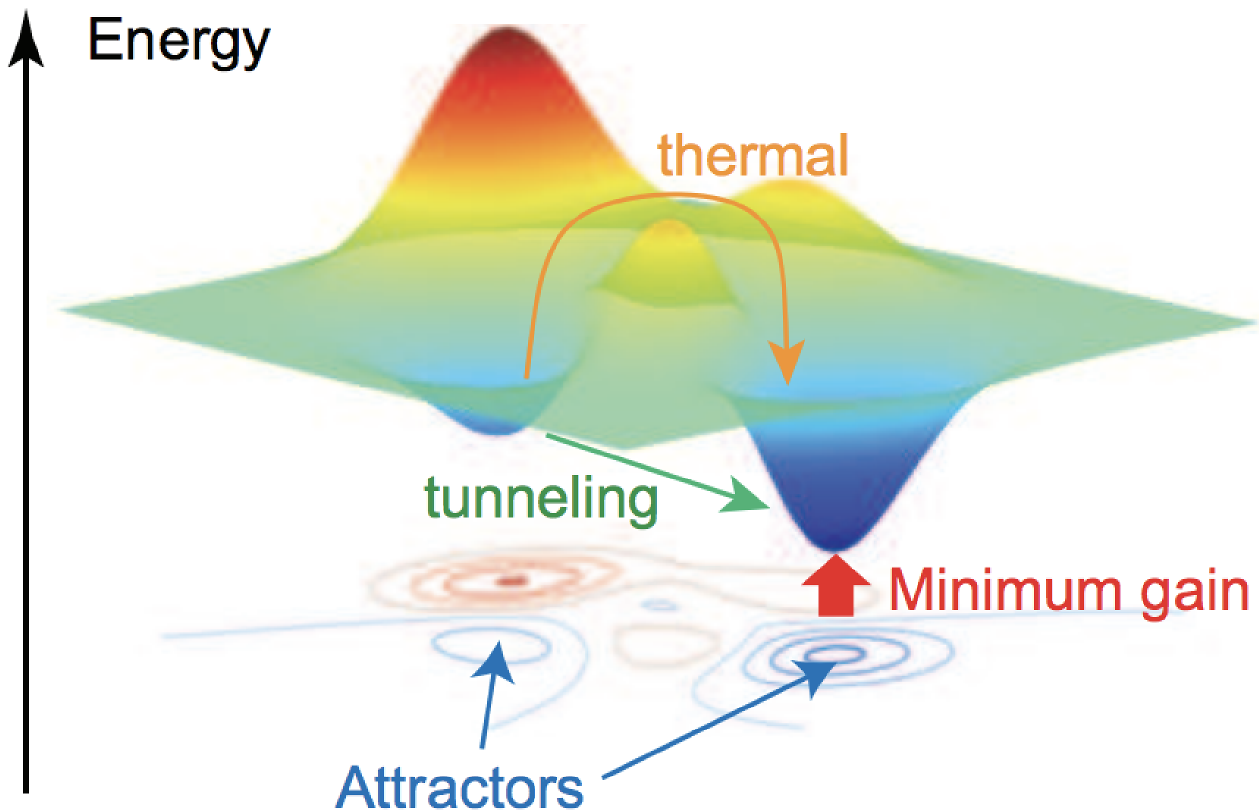


Figure 3.3: Ising energy function with different possible procedures to escape from local spurious minima. From [123]

system evolves at a given temperature that allows spins to escape from local minima if $k_B T > \Delta E$ where ΔE is the energy barrier between two spins configurations. This algorithm has proven to perform well for the ferromagnetic case where all couplings are set equals. But when the couplings are different from one another, the energy landscape complexifies and the system might end up in a local minimum instead than the global minimum that is desired.

This is because the temperature for this algorithm is kept fixed during all the flipping procedure. This is great for physics research as one can get physical properties at given temperature and then compare them to experimental data measured at the same temperatures. But for our case, it is at the expense of the quality of the solution of our combinatorial optimization problem.

To overcome this issue, Kirkpatrick *et al*. [115] proposed the algorithm of simulated annealing, that takes inspiration from metallurgy where we re-heat a material in order to crystallize in the correct order and to remove all defaults that could have been inside the bulk after initial fabrication.

They revisit the Glauber dynamics [124] (Alg. 1) where now the temperature is time-dependent. They propose to start the dynamics with a high temperature in order for the spins to flip a lot in the beginning (even if $\Delta E > 0$ because if T is high, then $\beta \approx \frac{1}{T}$ is low and the probability to flip is $e{-}\beta \Delta E \approx 1$) and that causes the system to explore the whole energy landscape and escape from local spurious minima. Then the temperature is slowly decreased to eventually reach $T = 0$. We retrieve this idea of decreasing a parameter for escaping from local minima with the use of learning rate decay in neural network gradient-based optimization techniques.

In Fig. 3.3, the thermal energy provided by temperature T allows the system to escape from

a spurious local minimum to land in the ground state. With SA, the systems initially explores the whole energy landscape - or equivalently, the landscape of configurations. Then as the temperature decreases, the system has less amplitude to explore large areas of the energy landscape and progressively settles at minima of the energy.

Algorithm 1 Glauber dynamics [124]

1: **Input**: Graph G with spins $\{\sigma_i\}_{i\in G}$, couplings J_{ij}, Inverse Temperature β
2: **Output**: $\sigma_{ground} = \min_{\{\sigma_i\}_{i\in G}} E_{Ising}(\{\sigma_i\}_{i\in G}, \{J_{ij}\})$
3: **for** $t \in [0, T]$ **do**
4: 1. Randomly choose a spin $\sigma_0^{x,y}$ whose coordinates are x, y
5: 2. Compute ΔE if $\sigma^{x,y}$ flips:
6: $\Delta E = (\sigma_{flip}^{x,y} - \sigma_0^{x,y}) * [J_{x,y;x+1,y}\sigma^{x+1,y} + J_{x,y;x-1,y}\sigma^{x-1,y}$
7: $+J_{x,y;x,y-1}\sigma^{x,y-1} + J_{x,y;x,y+1}\sigma^{x,y+1}]$
8: and since $\sigma_{flip}^{x,y} = -\sigma_0^{x,y}$ we can simplify:
9: $\Delta E = -2\sigma_0^{x,y} * [J_{x,y;x+1,y}\sigma^{x+1,y} + J_{x,y;x-1,y}\sigma^{x-1,y}$
10: $+J_{x,y;x,y-1}\sigma^{x,y-1} + J_{x,y;x,y+1}\sigma^{x,y+1}]$
11: 3. Decision to flip the spin:
12: **if** $\Delta E < 0$ **then** ▷ Gradient Descent on the energy
13: $\sigma^{x,y} \leftarrow -\sigma^{x,y}$
14: **else**
15: $\sigma^{x,y} \leftarrow -\sigma^{x,y}$ with $p \propto e^{-\beta\Delta E}$ ▷ Boltzmann probability
16: **end if**
17: **end for**

Algorithm 2 Simulated annealing

1: **Input**: Graph G with spins $\{\sigma_i\}_{i\in G}$, couplings J_{ij}, Inverse temperature annealing schedule (initial and final β + cooling law): [β^0,β^f]
2: **Output**: $\sigma_{ground} = \{\sigma_i\}_{i\in G}$
3: **for** $\beta \in [\beta^0, \beta^f]$ **do**
4: **for** $t \in [0, T_{per\ \beta}]$ **do**
5: 1. Randomly choose a spin $\sigma_0^{x,y}$ whose coordinates are x, y
6: 2. Compute ΔE if $\sigma^{x,y}$ flips:
7: $\Delta E = (\sigma_{flip}^{x,y} - \sigma_0^{x,y}) * [J_{x,y;x+1,y}\sigma^{x+1,y} + J_{x,y;x-1,y}\sigma^{x-1,y}$
8: $+J_{x,y;x,y-1}\sigma^{x,y-1} + J_{x,y;x,y+1}\sigma^{x,y+1}]$
9: and since $\sigma_{flip}^{x,y} = -\sigma_0^{x,y}$ we can simplify:
10: $\Delta E = -2\sigma_0^{x,y} * [J_{x,y;x+1,y}\sigma^{x+1,y} + J_{x,y;x-1,y}\sigma^{x-1,y}$
11: $+J_{x,y;x,y-1}\sigma^{x,y-1} + J_{x,y;x,y+1}\sigma^{x,y+1}]$
12: 3. Decision to flip the spin:
13: **if** $\Delta E < 0$ **then** ▷ Gradient Descent on the energy
14: $\sigma^{x,y} \leftarrow -\sigma^{x,y}$
15: **else**
16: $\sigma^{x,y} \leftarrow -\sigma^{x,y}$ with $p \propto e^{-\beta\Delta E}$ ▷ Boltzmann probability
17: **end if**
18: **end for**
19: **end for**

Simulated annealing has been applied to many combinatorial optimization problems [115]. SA is theoretically guaranteed to end in the ground state of the problem but only if the temperature is very slowly decreased [125]. In practice, the time required to guarantee the convergence can be greater than a brute-force search where all combinations are sequentially tested ($T \approx O(N^2)$) and limit the applications if one wants the exact ground state of the system.

Despite the fact that SA is a software algorithm that scales poorly with the size of the problem, it is still interesting to understand to process of driving a physical system via an annealing parameter (the temperature here) toward a state that is the solution of our problem.

In practice, simulated annealing could be realized by applying a kind of noise annealing by using either thermal [126] or electrical [127] effects to a system that is stochastic such as stochastic magnetic tunnel junctions or noisy memristors.

3.1.3 . Existing Ising Machines

Programmable Ising Machines are positioned between processors capable of high abstraction/ virtualization such as CPUs or GPUs and custom neural networks hardware implementations that are very task specific. Indeed, general-purpose processors can emulate networks that either have an architecture that largely differs from the processor layout and/ or that have a much larger number of processing units (spins) than the number of processing units of the chip at the cost of being energy-inefficient (see Section 1.2). On the contrary, special-purpose implementations can be very energy-efficient at the cost of being extremely optimized at performing one kind of operation, or more largely, embedding one kind of architecture.

Ising Machines are hardware that are built in order to find the ground state of an Ising Hamiltonian, so they are special-purpose hardware in this sense. However, if we are able to apply different parameters to the machine, we can solve different combinatorial optimization problems as long as they can can be mapped to the Ising energy function, so Ising Machines are general-purpose in this sense.

There exist plenty of different implementations that rely on different hardware and procedures to drive the system toward the ground state of a given Ising Hamiltonian. As Ising Machines are initially designed to solve combinatorial optimization problems where the parameters are fixed for a given problem, the usual metric to evaluate the performance of the IMs is the $time-to-solution$ (see Table 3.1) offered by a given IM. More precisely, authors report the evolution of $time-to-solution$ when the size of the problem increases. The best IMs are those that have the lowest slope (see D-Wave scalability vs other hardware scaling laws in Table 3.1).

Here we will focus on a key feature other than the time-to-solution of the IMs as our aim is to use IMs as a platform to train neural networks on-chip. Contrary to combinatorial optimization problems where the coupling parameters are set by the problem to solve and kept fixed during the sampling procedure, the parameters of neural networks are continuously being updated during the training process of a neural network. Thus it is important for such potential application to have an IM that can easily update its coupling parameters. For the work developped in this thesis, it is important to have the greater balance between facility and bit accuracy (see Table 3.1) but our main concern is the easiness of the interfaceability between our desktop computer and the IM. However, for practical applications, e.g. embedded applications, we could find other hardware more interesting than the one we will use for our work as the constraints are not the same (size, power).

Hence, in the following small review of existing IMs, we will discuss the ease with which the parameters can be set and modified in the existing hardware IM. We will also discuss the connectivity between spins on the IMs. Indeed, solving combinatorial optimization problems (COP) or training

neural networks typically require a high connectivity between the spins/ neurons on the IM. An all-to-all connectivity is sought as with careful setting of the connectivity matrix one can represent any COP or neural network on the IM. But hardware constraints often diminish this capacity and the IM ends up with local-connectivity only. The latter case makes an embedding step necessary to represent any problem with the specific IM. We will review the existing implementations regarding the mechanisms that drive the system toward the ground state: either with an artificial controlled and sequential dynamics of the spins or with a dynamics that is intrinsic to the hardware with which the IM is made.

Implementation	**CMOS** [128]	**Memristors** [127]	**CIM** [129]	**D-Wave** [129]
Number of spins	2000	60	100, 2000, 10000	2000, 50000
Time-to-solution	$10\mu s$	$0.3\mu s$	$600\mu s$	$10^4 s$
Energy-to-solution	$0.45mJ$	$0.22\mu J$		$250MJ$
Connectivity	All-to-all	All-to-all	All-to-all	Sparse
Convergence mechanisms	Artificial	Mixed	Phase-sensitive gain	Qubit projection
Ground state probability	ae^{-bN}	ae^{-bN}	ae^{-bN}	ae^{-bN^2}
Time-to-solution	$ce^{-d\sqrt{N}}$	$ce^{-d\sqrt{N}}$	$ce^{-d\sqrt{N}}$	ce^{-dN}
Parameters precision (bits)	16	1	2 [130]	5-6
Parameters update	Accurate	Stochastic [95]	Accurate	Accurate
Operating Temperature	Room temp.	Room temp.	Room temp.	10mK

Table 3.1: Comparison between different existing implementations of Ising Machines. Time–to-solution is equal to the annealing time times the number of repetition one has to do on the IM with the same problem in order to get the ground state with probability $> 99\%$. Time-to-solution and energy-to-solution are given for a MaxCut problem with $N = 60$ spins. The probability to reach the ground state and the time-to-solution scaling laws are function of the number of spins of a dense problem (Sherrington-Kirkpatrick or Max-Cut problem). Data taken from [123], [131].

Artificially dynamical hardware IM: . The first kind of hardware IMs are hardware that do not naturally evolve toward the ground state of the Ising Hamiltonian but also that do not have an intrinsic dynamics. We call such hardware "artificially dynamical IMs".

Those implementations most often rely on Simulated Annealing, or a derived procedure, in order to drive the system to low energy states. But they also rely on artificial rules to update the spins given the configuration of the whole system.

Here we focus on two hardware implementations of artificial IMS:

- the first one based on standard CMOS technology
- the second on memristors.

Several CMOS-based IMs have been built on FPGAs (Fujitsu, Toshiba [128]). This allows emulating Simulated Annealing much faster than with GPUs as FPGAs are designed to optimize every operation required for doing a Simulated Annealing (or analogous) procedure. The spins are virtual, and their connectivity is often, in practice, limited to local-only connectivity. The parameters are easily updated as the FPGA can be straightforwardly interfaced with a computer and there is no underlying physical process in applying updates to the parameters, it is just about changing bits in a register that stores the parameters. However, it still needs a large memory to store all the

parameters. For experimental work in a lab it is transfer times are manageable, but it could prevent the use of such an IM for practical implementation such as on an edge device.

The second kind of implementation [127] is also based on artificial spins emulated with threshold units regarding an input current that is derived from the Ising Hamiltonian to minimize:

$$\sigma_i(t+1) = \begin{cases} +1 \; if \; \sum_j J_{ij}\sigma_j(t) + \eta_i > \Theta_i \\ -1 \; if \; \sum_j J_{ij}\sigma_j(t) + \eta_i < \Theta_i \end{cases} \tag{3.9}$$

where Θ_i is an individual tunable threshold and δ_i is the individual controllable noise that is added by the dedicated memristors.

Those spins are all-to-all coupled with a cross-bar array where the memristors, placed at each cross-point, encode the coupling strength between two artificial spins. Contrarily to the previous artificial approach where SA was performed thanks to an artificial temperature, here the authors leverage the controllable noisy behavior of a type of memristors to add controlled noise to the system in order for it to escape from spurious local minima. The cross-bar array allows to quickly compute, in an energy-efficient way, the switching criteria for each spin of the system (the sum of Eq. 3.9).

Such a system is interesting for learning as it leverages the principle of in-memory computing by the use of a memristor cross-bar array. Additionally, the updates of the memristors can be performed extremely fast and are very energy efficient. By storing the coupling values in the conductance of the memristors, we get rid of the memory requirements that are large for such systems when the number of spins becomes large.

These implementations show promising results [128], [131] but exhibit limitations. For CMOS-IMs built on FPGAs, the memory requirements and the non-locality of the coupling storage can make this IM unsuitable for edge applications as it would require large memory and energy budgets. IMs based on memristors cross-bar arrays are more promising for embedded applications as the couplings are locally stored and the IMs leverage the Kirchhoff laws for the computing part of the update. However, this hardware implementation still suffers from the variability of memristors, and the authors could only store 0 or $+1$ couplings between the spins. Also, it remains to be demonstrated that they could scale this approach to larger cross-bar as the larger problem they solve is made of 60 spins.

In the next paragraph, we will introduce IMs that are designed in such a way that the system reaches the ground state via the intrinsic dynamics of the system. These IMs deviate from the implementation of artificial IMs as no control is done on the actual spins in order to flip but only a global annealing parameter is applied on the system to slowly guide the system to reach the ground state.

Intrinsically dynamical IMs. These new approaches, that are based on unconventional hardware, have been developed in order to overcome the drawbacks of Simulated Annealing. SA can be lengthy and not suitable for energy landscapes that have narrow and deep local minima [132], [133]. However, such energy landscapes are of great interest for us as they are similar to those encountered in machine learning problems [133]. Those approaches aim at better finding the ground state of the system by either leveraging tunneling effects in a quantum system or by having a from-below-approach that makes the system reach the ground state from below the energy landscape.

The first remarkable implementation is the quantum annealer proposed by D-Wave [134]. This implementation encodes the spins in a quantum property of superconducting qubits that is the circulation of a superconducting current in a superconducting loop. That current circulate either $|+\rangle$ or $|-\rangle$, the clock-wise or anti-clock-wise circulation, and as a quantum state it can in a superposition

of the two possible states: $\alpha |+\rangle + \beta |-\rangle$. Leveraging the quantum nature of the state of the current, D-Wave use a procedure called "Quantum Annealing" (QA) that is similar to SA but builds on tunneling effects rather than thermal activation to escape from spurious local minima. This approach is appealing and this hardware has been used for many applications [135]–[142]. Moreover, this hardware is highly optimized at quickly setting the parameters on the chip for solving a given problem as the IM runs on the cloud in order to maximize the availability of the chip to multiple users. Multiple DACs[1] are dispatched on the chip to allow programming in a highly parallel way the magnetic memories that control both the local individual bias and the coupling strength.

However, the magnetic couplers that connect the superconducting loops necessitate the qubits to be spatially close, which restricts the connectivity to local coupling only, with a geometry that depends on the architecture[134]. In order to represent a given problem on the chip, one has to go through an embedding step where multiple hardware spins are used in order to represent a single spin from the problem formulation. Those hardware spins are strongly coupled with a ferromagnetic coupling such that they encode the same information. It results that the size of the problem we can embed on the chip is dramatically reduced. Consecutively, $time-to-solution$ is increased [123] as the information about the spins states has to flow through many more hardware spins compared to a situation where all-to-all coupling is implemented.

Nevertheless, the D-Wave IM is today the intrinsically dynamical hardware that is the most suitable to be used in laboratory for performing experiments that demonstrate that we can train artificial neural networks only with the dynamics of that system. But the D-Wave IM would be unusable for edge applications. Indeed, it runs inside a dilution fridge for being operated at very low temperature to improve the coherence time which assures the quantum regime and the coupling scheme is wasteful in terms of chip area as multiple qubits are required to implement a single spin.

A last interesting hardware IM that is naturally dynamical is the Coherent Ising Machine [143]–[147] .This hardware encodes the spins in the phase of degenerate optical parametric oscillators (DOPOs).

This hardware reaches the ground state of the Ising Hamiltonian following a very different approach than previously described. These previous approaches rely on either Simulated or Quantum annealing that drive the system through the energy landscape from above in such a way that it can be trapped in local minima. On the contrary, a CIM approaches the energy landscape from below (see Fig. 3.3 "Minimum gain") such that the first point of the energy landscape the CIM reaches is the state of lowest energy or the ground state.

The DOPOs are driven by gain and loss terms which affect both the phase and the amplitude. The balance of gain and loss terms define an oscillation threshold which depends on the Ising Hamiltonian to minimize that corresponds to a loss term. We drive the system to the ground state by gradually increasing a gain term that yields oscillations both in phase or out of phase depending on the loss terms.

It is expected that this property of approaching the energy landscape from below could exhibit better results [123] and the problem of getting stuck in local minima is greatly diminished.

However, the CIM is run in a sequential way. Indeed, the DOPOs, that are light pulses, are sequentially sent to a light fiber that is long enough in order to contain many of these DOPOs. Then, the DOPOs are sequentially read out and the gain and loss terms are individually applied on each DOPO depending on the state of the other DOPOs. In principle, this scheme increases the time-to-solution but as we deal with light, the speed of the DOPOs is very high so it results in a fast IM, much faster than D-Wave. Luckily, this scheme allows to couple all DOPOs with all

[1] Digital to Analog Converter

others DOPOs which allows to solve very dense Ising problems. Despite the fact that updating the couplings is very easy to do as they are stored in a side FPGA, they still require a great amount of memory to store all the parameters, which, added to the fact that the IM is spatially large, would prevent the use of such IM for embedded applications.

Summary. In conclusion, today IMs are either implemented through emulators that run their equations using simulated annealing (CPUs, GPUs, FPGAs and memristor cross-bar arrays) or through physical IMs that truly minimize their energy to solve the problem (D-Wave, coherent Ising machines). This is the later class of IMs that will be our focus within this thesis as we wish to harness physics, and in particular the ability of a physical system to minimize its energy, for neural network computations. We have chosen D-Wave as a platform for our implementation of learning in Chapter 5 because it is easily accessible via the cloud, through an API that allows modifying the coupling parameters. However, as mentioned above, this implementation is not low power, and does not store parameters as physical quantities, but in the memory of a computer from which it loads. The only in-memory computing IM is the one of [127], with parameter stored as memristors, but the implemented system only emulates the equations of the IM and does not minimize its energy for computing. The ideal system for our study, consisting of an in-memory system with adjustable, non-volatile parameters, and that naturally minimizes its energy to compute, remains to be implemented and is today the topic of very active research (refs).

3.2 . Equilibrium Propagation: supervised learning with energy-based models

In Section 1.1.3 we described the process of "learning" in deep neural networks that refers to prescribing updates to the parameters of the neural network in order to optimize (most often minimize) an objective function. We extensively took the example of backpropagation as a supervised learning framework to describe a gradient-based optimization approach but also to motivate the need of novel physics-driven hardware because of the limitations of backpropagation. In the mean time we introduced energy-based models that are systems that compute solely thanks to the natural behavior of physical systems to evolve towards states that minimize their energy function. Despite the fact that energy-based models are really suited to support hardware implementations, no supervised learning of a global objective function has been proposed with such approaches and thus those models suffer of an accuracy drop compared to models trained with backpropagation.

Equilibrium Propagation (EP) [14] is an alternative supervised learning framework published in 2017 that, as the title of the paper says, "bridges the gap between energy-based models and backpropagation". EP is a supervised gradient-based optimization technique that - contrarily to backpropagation [148] - computes the parameters updates with the same operations for both the inference and the error-backpropagation phases - similarly to what is done with the energy-based models described in introduction. Thus EP is a good candidate for being a supervised learning algorithm to train in a local fashion unconventional hardware.

EP is grounded in the energy nature of the models it trains. Thus to introduce EP we will start with a physical description of EP to get an intuition of why and how it works. Then we will describe more carefully EP in machine learning terms.

3.2.1 . Physical intuition behind Equilibrium Propagation

Physical Intuition. In this section we will start from the definition of the energy-based models defined in the Section 2.2 by taking the example of Hopfield networks. In this type of network, the neurons evolve towards the minimum energy state closest to the initial state of the system (=input). The energy function of the network is parameterized and by adjusting the parameters in a suitable way, we can create minima corresponding to the different patterns to be stored. Hopfield networks are said to be self-associative since they associate to an input (=initial state) an equilibrium state which size corresponds to the size of the input. We have seen the limitations of Hopfield networks and in particular the fact that no hierarchy emerges in this type of network due to the unsupervised learning and the self-associative behavior.

We will now work with a slightly modified Hopfield network in order to perform supervised learning. In fact, we will still have symmetrical connections between neurons but in our case the neurons will be connected in such a way as to create an architecture where some neurons will be assigned to the inputs and will be static during the evolution of the system, others to the outputs and the rest being hidden neurons. This type of system is always described by an Hopfield energy function, the only modification relies in the connectivity matrix of the system. It is kept symmetric but the neurons are now coupled in a way that define a layered architecture, similarly to standard neural networks. For this purpose, we introduce a new notation where s is the set of states of the neurons in the network and $s = \{h, y\}$ where h is the set of the hidden neurons whereas y is the set of the output neurons. The symmetry of the couplings is very important in this case since it will allow signals applied to both input and output neurons to influence the dynamics and thus the equilibrium state of the network. We will take advantage of the fact that we can have a top-down signal (input

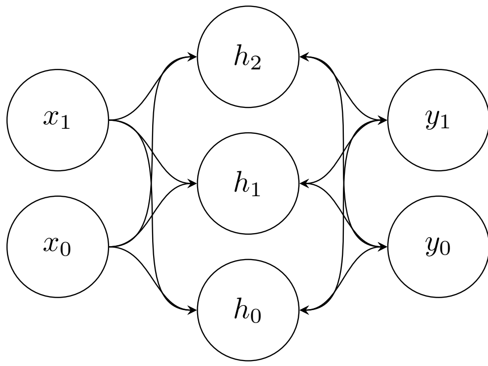


Figure 3.4: Free phase

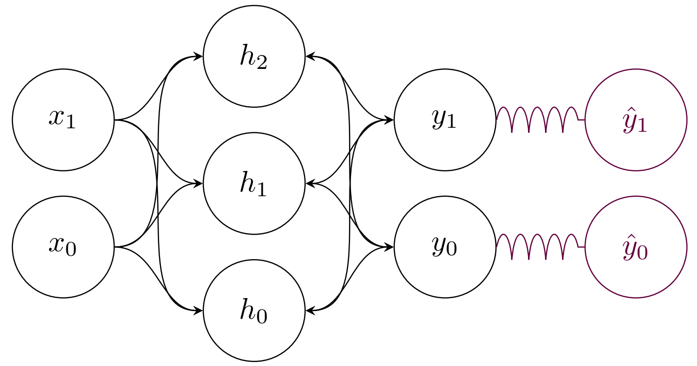


Figure 3.5: Nudge phase

$\leftrightarrow$ output) but also a bottom-up signal (output $\leftrightarrow$ input) to perform supervised learning using only one type of calculation for the inference and error propagation phase.

The classification task we are interested in here amounts to assigning a category to input data. The challenge of learning is to update the parameters of the system so that it predicts the category of the data presented to it. This is achieved with energy-based models by updating the parameters of the energy function in order to maximize the probability of predicting the right class given the inputs and the system parameters (see Section 2.2).

In most cases, one output neuron is assigned per class to be predicted in the database. In a similar way, the target - *i.e.* the corresponding class of the presented input - is said to be one-hot encoded[2]. The predicted class is the one corresponding to the neuron with the highest activity when the network has reached the steady state.

We have seen with Hopfield networks a learning rule that allows to lower the energy of a state (configuration of neurons) to be memorized in order to sculpt the corresponding local minima so that the network converges to one of the memorized patterns (see Section 2.2.3). But in the classification task we do not want to memorize the pattern. We just have the inputs, the class, as well as the network dynamics which associates to a given input an equilibrium state of the network, and thus an output layer state from which we make a class prediction.

The EP learning rule is based on the comparison of two states. The first state corresponds to the state of the system, given only by its dynamics, when an input is presented (s^{model}, as introduced in the introduction of this section, s is independently the state of a hidden h or an output neuron y). The second state corresponds to the same system, with the same input, after it has been slightly perturbed at the output layer in the goal of making a better prediction s^{target} - thus decreasing the computed cost function at the output layer (see Fig. 3.5). In this second state, through the set of symmetric connections, the perturbation applied to the output layer propagates throughout the network, thus encoding the error signal in the change of activity of each neuron. From these two states, we will then lower the energy of the end state of the perturbation phase which has the lowest cost function value calculated at the output and at the same time increase the energy of the first state which is given by the intrinsic dynamics of the network when the inputs are presented. Thus, EP is a learning algorithm with two distinct phases but based on the same type of computation which is the dynamics of the network.

Since the network is always described by a Hopfield energy function, we decrease the energy of

[2]One-hot-encoding example: if the actual label is 3 and the dataset can be classified into 10 classes (0 to 9 let's say), the one-hot-encoded target vector is then: $[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]$

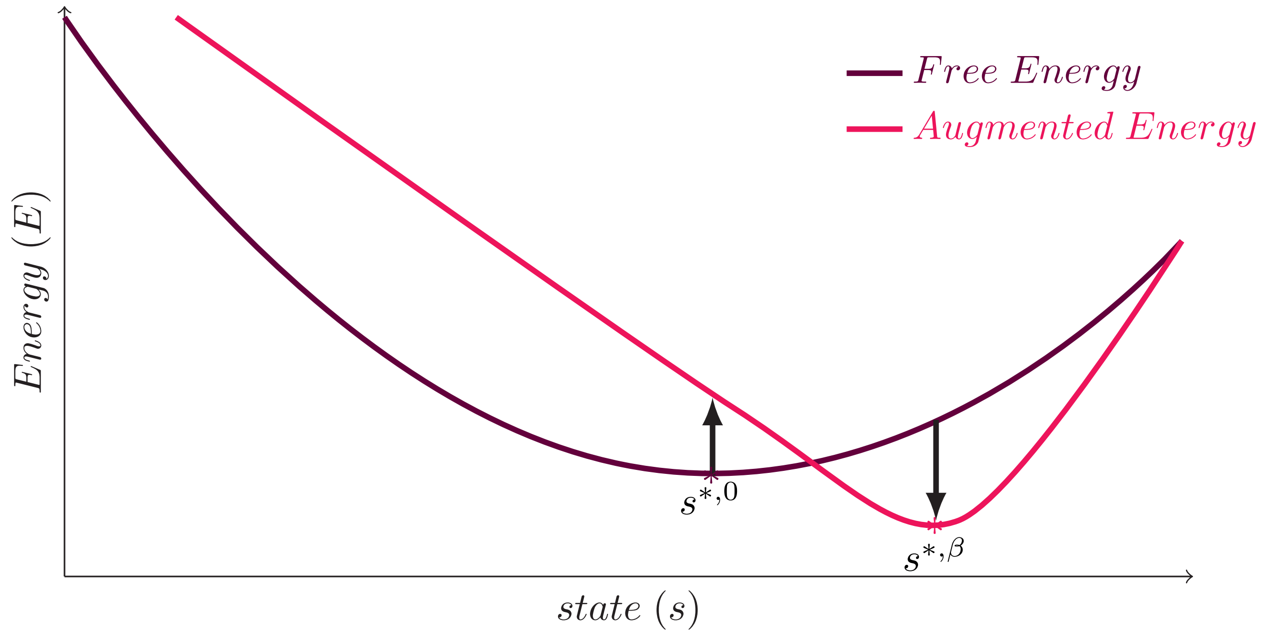


Figure 3.6: Two-phases training procedure of EP: the system successively reaches the minimum of the free then the augmented energy functions. The energy of the state that predict a wrong class is increased while the state that better predict the target vector has its energy diminished.

a state with the same learning rule as the Hopfield networks:

$$\Delta W_{ij}^{-} \propto s_i s_j \tag{3.10}$$

Similarly, we increase the energy of the other state by taking the opposite of the energy minimizing rule:

$$\Delta W_{ij}^{+} \propto -s_i s_j \tag{3.11}$$

We then sum the two contributions to obtain the following learning rule:

$$\Delta W_{ij} \propto \Delta W_{ij}^{-} + \Delta W_{ij}^{+} = [s_i s_j]^{target} - [s_i s_j]^{model} \tag{3.12}$$

This rule is similar to the one prescribed for Boltzmann machines. However, whether it is for the "model" or "target" contribution of the two phases, the inputs are always presented to the network, contrary to the Boltzmann machines where the second phase consists in the reconstruction of the inputs via the state of the hidden layer.

Now it remains to define how to perturb the system so that the perturbed state approaches an ideal prediction : output = target in order to realize the second phase of EP.

A first method consists in doing Contrastive Hebbian Learning (CHL) [149]. In the second phase, CHL proceeds by setting the output units to the values of the target one-hot encoded vector. The hidden neurons of the network will then evolve towards a second steady state which, from the inputs presented, gives the clamped state of the output neurons. In that case, the learning rule Eq. 3.12 minimizes the contrast function $J = E^{target} - E^{model}$ which is the difference between the network energy of the two phases. However, despite the fact that this learning is similar in spirit to the objective of the training with EP, it has been shown that the Contrast function can take negative values as the system can settle to different energy modes for both the inference and the clamped

phases, which alters the training. Also, CHL computes the gradient of the contrast function which is different than computing the gradient of a cost function computed at the output layer. Indeed, the emergence of the hierarchy is dictated by the optimization of that cost function that is only computed at the output of the network. The CHL optimization method minimizes an objective that is global so no hierarchy is expected to materialize, thus the accuracy is expected to be below what a supervised backpropagation type method offers on a classification task.

This objective function, that is dictated by the kind of optimization and the energy of the network, finally prevent a flexible choice of function to be optimized by EP.

Building on this work but with a different scheme for the second phase, Scellier and Bengio [14] showed that by applying an elastic restoring force to the output neurons that drive them toward the target state, in the limit where the spring restoring constant β is small, a learning rule similar to Eq. 3.12 computes the gradient of the Mean Squared Error cost function between the output layer of the network and the target vector. Applying a small force on the output neurons enables to avoid the system to settle in a different energy mode, which was a drawback of CHL.

In fact, applying this elastic restoring force amounts to modifying the energy function of the network during the second phase. We add to the initial energy function E the cost function C modulated by the stiffness constant β which gives the increased energy function: $F = E + \beta C$. Thus, while the system has evolved towards the state that minimizes E in the first phase, in the second phase it will evolve towards the state that minimizes both E and the cost function C, which is the goal of the second phase procedure.

Thus, EP makes supervised learning possible by relying only on the property of the network to evolve towards its most probable state (of minimum energy). Moreover, the proposed learning rule depends only on the neurons adjacent to the weight in question, so it is a local rule which is, as said in the introduction, very desirable for hardware implementations.

The energy function that describes the system of interest gets its name from the fact that physical systems evolve towards the most probable states, and thus states with minimal energy in statistical physics. But in the context of EP, the energy function of a system can also be understood as some parametrizable physical quantity that the physical system minimizes through its evolution. We will describe two works that each use a very specific "energy" function for dissipative electrical networks.

Typical energy functions for two implementations of EP. The first work to apply EP to a real physical system is the work of Kendall *et al*. [150] who proposes a learning scheme based on EP to train an artificial hardware neural network whose synapses are resistive components (memristors can be such devices for example - see Section 2.1) and the non-linearity is implemented by adding diodes at the level of the nodes.

In the paper, they show that this particular class of non-linear analog neural network - where the synapses are resistive elements - naturally evolves by minimizing a quantity that is the total pseudo-power of the system. If we denote the neurons by the electrical potential *i.e.* $s_i = V_i$ and the conductance of the resistive device coupling nodes i and j as g_{ij}, then the total pseudo-power of the network reads as:

$$\mathcal{P}(V_1, V_2, ...) = \sum_{i=1}^{N} g_{ij}(V_i - V_j)^2 \tag{3.13}$$

and the network evolves according to the Kirchhoff laws toward the equilibrium state $(V_1^*, V_2^*, ...)$

defined by the condition:

$$\frac{\partial \mathcal{P}}{\partial V_i}(V_1^*, V_2^*, ...) = 0 \tag{3.14}$$

The set of voltage nodes $\{V_1, V_2, ...\}$ includes the input voltage nodes, which prevents the trivial solution $\{V_i^* = 0\}_{i=1\rightarrow N}$ by imposing boundary conditions to the network.

Given this condition, the authors derive the corresponding EP learning rule that computes the gradient of the global loss function $\mathcal{L}$:

$$\frac{\partial \mathcal{L}}{\partial g_{ij}} = \lim_{\beta \to 0} \frac{1}{2\beta} \left(\left(\Delta V_{ij}^{\beta} \right)^2 - \left(\Delta V_{ij}^{0} \right)^2 \right) \tag{3.15}$$

where ΔV_{ij} stands for the voltage drop $V_i - V_j$ accross the resistive device g_{ij}. Indices ^{0} and $^\beta$ stand for the voltage drops measured after the first phase where only the inputs are fed to the network and after the second phase where the output neurons are nudged toward the target state as described in the section above.

Kendall *et al*. [150] performed numerical simulations with SPICE with ideal programmable resistors (not with a realistic model of memristors that incorporate the non-idealities that alter the training for instance). They report accuracy on MNIST [151] at state of the performance reached with standard neural network trained with EP. This work is one step toward hardware realization but still relies on numerical simulations, far from realistic hardware device issues.

The only hardware realization of training a dynamical system in-situ is the recent work of Stern, Dillavou *et al*. [120], [152]. Similarly to what proposed Kendall *et al*., they built a small resistive neural network. The synapses are standard[3] programmable resistive elements. They use Coupled Learning [153] as the learning algorithm that is somehow a mix between CHL and EP: the authors clamp the output unit in a state that is a step closer to the desired target state - not at the target state as with CHL, so the output units are kept fixed over time for the second phase, unlike EP where output neurons are dynamically nudged towards the target state. To accelerate the training process, they simultaneously run two networks that have the same parameters, one doing the first phase, the second having its output nodes being clamped toward the target state. That way, no memory is required to store the first state as the update is directly computed from the measurements of the two circuits after they have reached equilibrium. They also simplify the resistor update scheme by binarizing the gradient applied to the resistive elements. They successfully achieve training on the iris dataset that is the largest task they can solve with the hardware realization. This approach may not be scalable due to the lack of non-linearity in the system.

These two works pave the way of hardware implementations trained by EP. Now, EP still suffers from pure software simulations that slow down the training process and prevent EP to scale to larger tasks (such as ImageNet [154], even if recent work has been carried out on ImageNet 32x32 [155]).

However, in order to scale to large tasks, the number of parameters will increase accordingly and for a hardware implementation, that means to use a greater number of components. But the only way to design such a large-scale hardware implementation without a large energy-consumption overhead implies to use emerging nano-scale devices as envisioned in [150]. Yet, these devices are, mostly, still experimental and exhibit a lot of device-to-device variability and are noisy. All these imperfections can alter or even prevent any training with EP (see Section 2.1). We will see in the next chapter (Chapter 4.3) how the precision of both the parameters and the neurons can be reduced in order to use nano-scale devices in a regime where they exhibit no variability and no noise.

[3] The authors use the AD5220 element that is a discrete CMOS component with 128 possible resistive states.

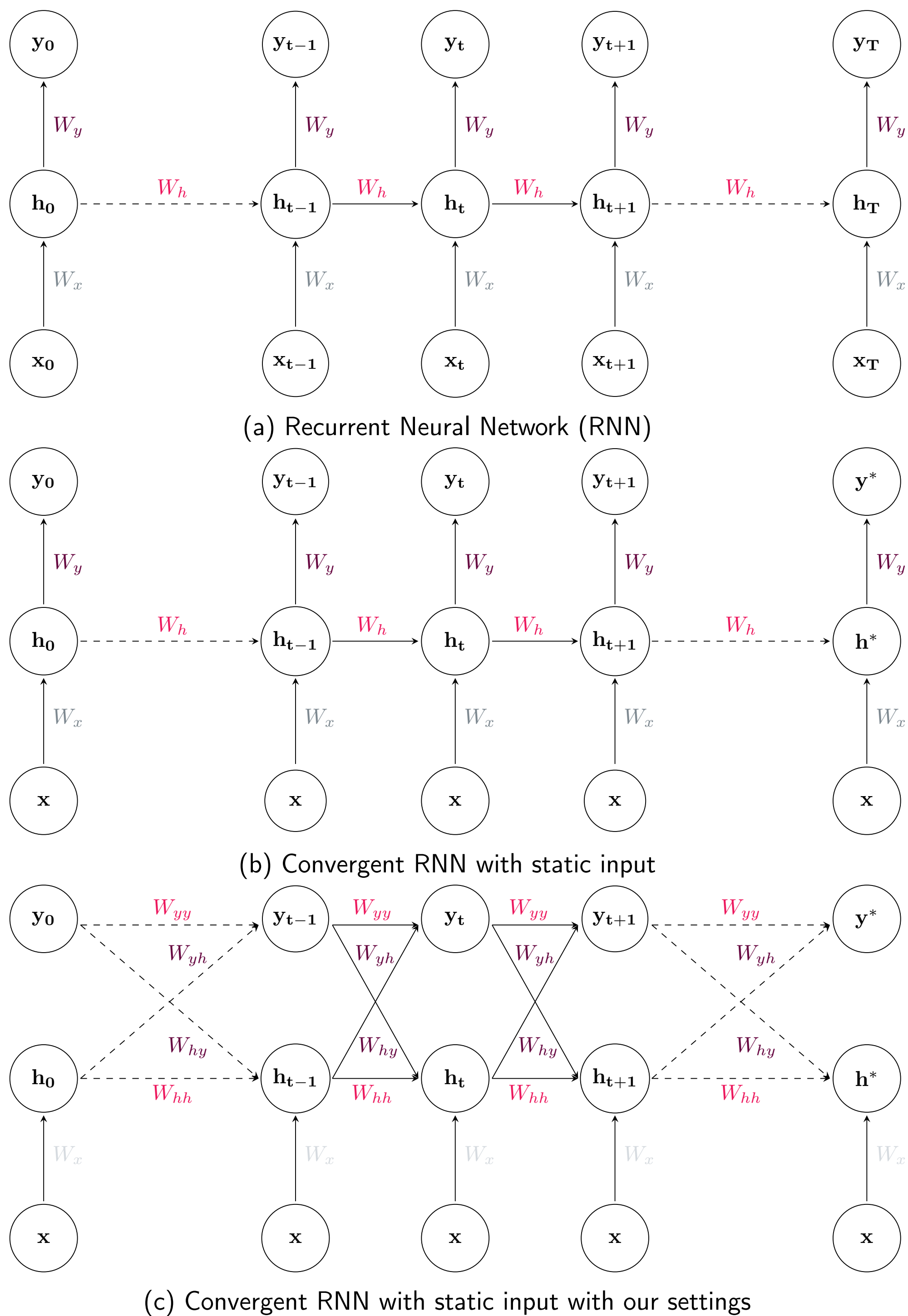
3.2.2 . Machine Learning description of EP

So far, we have introduced EP with physical intuitions of how and why EP works. We now derive EP in a more rigorous way, using the nomenclature of machine learning.

EP is an ubiquitous learning framework that can be applied to all kinds of supervised tasks as long as the inputs are static over time (no time-series or natural language processing, for the moment): classification, generation, ... and the research has up to now only focused on the image classification tasks for which a specific class is assigned to input data .

In this context, we will focus on neural networks composed of an input layer that represents the data to be classified, an output layer where the input is classified and hidden layers that non-linearly transform the input in order to do the classification. However, conversely to standard pure-forward neural networks, EP applies to recurrently connected neural networks (the symmetric connections we talked about in the previous section).

Convergent Recurrent Neural Networks. Recurrent neural networks (RNNs) (Fig. 3.7a) are neural networks that have a temporal dimension as the input of the hidden neurons is both a new input and the previous state of the same hidden neurons. So RNNs have been extensively used to process data that have a temporal dimension such as time series or natural language processing. However, the networks trained by EP can not handle yet this temporality in the input as the system is required to reach an equilibrium state given a static input (see [156] for ideas to deal with time-varying inputs). RNNs function in the following way: at each time-step a RNN updates its output and its hidden state given a new input and its past hidden state (Fig. 3.7a). A typical use case is to use RNNs for natural language processing. We could want to generate the sequel of a text based only on a small text entered by a user. The RNN will predict the next word to generate based on the last word of the text and the previous hidden state (Fig. 3.7a) that is the equivalent of a "memory" in order to predict a word that makes sense in the context of the words already generated. We illustrate this kind of RNN in Figure 3.7a with a visual example where the input value changes at each time step, which triggers the update of the hidden and the output units.



(a) Recurrent Neural Network (RNN)

(b) Convergent RNN with static input

(c) Convergent RNN with static input with our settings

Figure 3.7: Three kinds of RNNs. We denote x the input nodes, h the hidden states, y the output states. (a) Computational graph of a standard RNN: hidden and output states change at each time step according to previous state and new input data. (b) RNN with static input: hidden and output states change at each time step according to previous state while the input does not change: the hidden and output states evolve toward a fixed point. (c) Convergent RNN with static input with a configuration closer to the settings of EP: the network still evolves over time according to the fixed input and here the hidden and output layers are mutually coupled.

EP relies on a subclass of RNNs that we call convergent RNNs with static input. Such networks have their input fixed over time (Fig. 3.7b). Thus, the system will eventually converge toward a fixed point denoted by the star in Fig. 3.7b. This fixed point is called equilibrium point in the Equilibrium Propagation framework and corresponds to the state of minimum energy of the physical systems described in the previous section.

We describe more carefully the kind of RNN that will be used with EP in Fig. 3.7c. Unlike standard RNNs, for our application, the hidden and output states mutually influence each other symmetrically between two consecutive time steps due to the symmetric nature of the synapses in our networks. As already stated in the previous section, this particular setting allows a change in activity at the output layer to propagate in the system and is crucial for error-backpropagation for training the system with EP.

In fact this description is similar to simulate a dynamical system with discrete dynamics such as when we use the Euler solver to solve an ODE where each time-step in the recurrent neural network corresponds to a time-step in the dynamics of the dynamical system - the ODE of the network is given by its energy function.

Energy-based model. We have already described what are energy-based models in Section 2.2. EP relies on the property of such models to evolve toward an equilibrium state that corresponds to a minimum of the energy function. Ideally this minimum is the state that correctly predicts the class of the input. Stating that we know the energy function of the system $E(x, s, \theta)$, the network follows a dynamics that minimizes the energy over time. Similarly to Hopfield networks, one dynamics that guarantees convergence is to compute the gradient of the energy with respect to the states of the neurons and update them in order to minimize the energy:

$$\frac{ds}{dt} = -\frac{\partial E}{\partial s}(x, s, \theta) \tag{3.16}$$

With sufficiently small steps the system eventually converges toward an equilibrium state s^* given by the condition:

$$\frac{\partial E}{\partial s}(x, s^*, \theta) = 0 \tag{3.17}$$

Machine learning settings. We assume we have a convergent recurrent neural network whose dynamics derives from an energy function E (Eq. 3.16) that depends on x the input, s the state of the units in the system and θ, the set of parameters of the system: $E = E(\theta, s, x)$.

We also define a cost function ℓ that describes the discrepancy between the output units at the first equilibrium point $y^{*,0}$ and the target state $\hat{y}$. The loss function $\mathcal{L}$ is the cost function evaluated at the first equilibrium state: $\mathcal{L} = \ell(y^*, \hat{y})$.

Then the goal of the training is to minimize the cost function computed at the first equilibrium point by adjusting the parameters θ of the system with the condition of the system at equilibrium:

$$\begin{cases} \min\limits_{\theta} \ell(y^{*,0}, \hat{y}) \\ \frac{\partial E}{\partial s}(\theta, s^{*,0}, x) = 0 \end{cases} \tag{3.18}$$

We will now describe the training procedure of EP.

A two-phases procedure. The training procedure boils down to sculpting the energy landscape of the system such that when presenting a new input, the system eventually converges toward a state that predicts the correct class for the input.

As mentioned in the previous section, to compute the parameter updates that minimize the cost function at the output layer, Equilibrium Propagation proceeds in two phases.

For the first phase, we fix the input units at the values of the input and we let the system evolve according to its energy function (Fig. 3.8) until it reaches the first/ free equilibrium point $s^{*,0}$ given by the condition 3.17.

A standard approach to compute the updates of the parameters in order to minimize the cost function would be to use backpropagation through time - as we deal with recurrent neural networks. But now we describe the new EP procedure that computes the same updates by simply relying on the same dynamics of the system as for the inference phase.

To perform the second phase where the error at the output layer propagates in the network, Scellier & Bengio introduced the concept of augmented energy function F to drive the dynamics of the system during the second phase. F is the sum of the initial energy function E and the cost function computed at the output layer of the system after the free phase. The cost function is modulated by the nudging factor β which gives more or less weight to the cost function with respect to the total augmented energy F.

With the same notations as previously, we denote β the nudging factor and $\hat{y}$ the target for the output layer. The augmented energy function reads as:

$$F(x, s, \theta, \beta, \hat{y}) = E(x, s, \theta) + \beta * \ell(y, \hat{y})) \tag{3.19}$$

For second phase of EP - or nudging phase - the system starts in the first equilibrium state $s^{*,0}$ and then evolves according to the augmented energy function $F(\theta, s, x, \beta, \hat{y})$ (Eq. 3.19).

The dynamics of the units is now given by:

$$\frac{ds}{dt} = -\frac{\partial F}{\partial s}(\theta, s, x, \beta, \hat{y}) = -\frac{\partial E}{\partial s}(\theta, s, x) - \beta \frac{\partial \ell}{\partial s}(y, \hat{y}) \tag{3.20}$$

If ℓ is the Mean Squared Error[4] between the output units and the target vector, then the output units feel a spring-like force toward their target state as denoted in Fig. 3.9 (in the previous section we introduced the same spring-like force to nudge the neurons).

The system eventually converges toward a second fixed point - or second equilibrium state - that we denote $s^{*,\beta}$ that minimizes the augmented energy function F.

Learning rule. The strength of Equilibrium Propagation is that Scellier & Bengio have analytically shown that given these two equilibrium states $s^{*,0}$ and $s^{*,\beta}$, one can compute updates of the parameters of the system and that these updates are exactly equal to the gradient of the objective function $\mathcal{L}$ with respect to each parameter of the system:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \lim_{\beta \to 0} \frac{1}{\beta} \left(\frac{\partial F}{\partial \theta}(\theta, s^{*,\beta}, x, \beta, \hat{y}) - \frac{\partial F}{\partial \theta}(\theta, s^{*,0}, x, 0, \hat{y}) \right) \tag{3.21}$$

If the energy of the network is a Hopfield-like energy function where ρ is the activation function of the neurons, then the learning rule for a weight is simply:

[4]Mean Squared Error (MSE): $\ell(y, \hat{y}) = MSE(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$. Then $\beta \frac{\partial \ell}{\partial s}(y, \hat{y}) = \beta(y - \hat{y})$ hence the name "spring-like" force to drive the output neurons toward their target state.

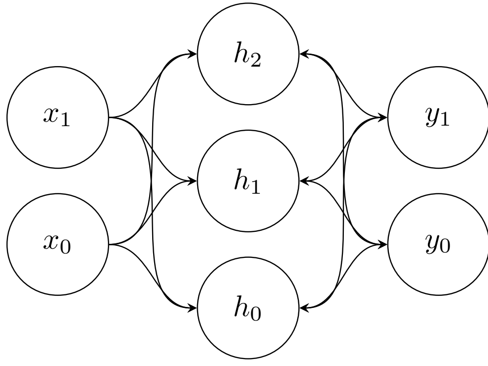


Figure 3.8: Free phase

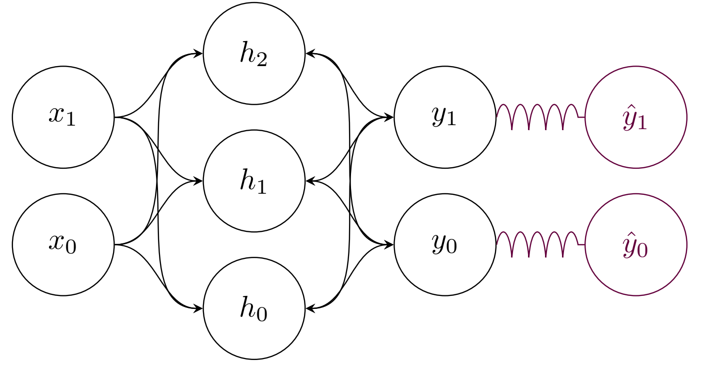


Figure 3.9: Nudge phase

$$\Delta W_{ij} = \frac{1}{\beta}\left(\rho(s_i^{*,\beta}) * \rho(s_j^{*,\beta}) - \rho(s_i^{*,0}) * \rho(s_j^{*,0})\right) \tag{3.22}$$

where the local nature of the learning rule is clearly visible as the update of the weights W_{ij} only depends on the state of the neurons it connects.

Finally, once we have computed the gradient, we update the parameters with simple stochastic gradient descent:

$$\theta \leftarrow \theta - \eta \frac{\partial \mathcal{L}}{\partial \theta} \tag{3.23}$$

where η is the learning rate that modulates the size of the step for the stochastic gradient descent.

The whole algorithm is summed up in Alg. 3.

Algorithm 3 Training a convergent dynamical recurrent neural network with Equilibrium Propagation - we use Euler solver for solving the ODE

Inputs: $s = \{h, y\}$, $\theta = \{W, b\}$, $E(\theta, x, s)$, $\{x, \hat{y}\}$
Outputs: $\theta = \{W, b\}$, $E(\theta, x, s)$

1: *I. Inference & Error-backpropagation phases*
2: **for** $t = 0$ to T **do** ▷ 1.Free phase
3: $\quad s_{t+1} = s_t + dt * \left(-\frac{\partial E}{\partial s}\right)$
4: **end for**
5: $s^{*,0} \leftarrow s_T$ ▷ 2. Store first equilibrium state
6: $\ell(s^{*,0}, \hat{y}) = \frac{1}{2}(y^{*,0} - \hat{y})^2$ ▷ 3. Compute Loss (MSE) function
7: **for** $t = T$ to K **do** ▷ 4. Nudge phase
8: $\quad s_{t+1} = s_t + dt * \left(-\frac{\partial E}{\partial s} - \beta\frac{\partial \ell}{\partial s}\right)$
9: **end for**
10: $s^{*,\beta} \leftarrow s_K$ ▷ 5. Store second equilibrium state
11: *II. Gradient computation & Optimization step*
12: $g_\theta = \frac{1}{\beta}\left(\frac{\partial E}{\partial \theta}(\theta, s, x, \beta, \hat{y}) - \frac{\partial E}{\partial \theta}(\theta, s, x, 0, \hat{y})\right)$ ▷ 6. Compute EP gradient
13: $\theta \leftarrow \theta - \eta \cdot g_\theta$ ▷ 7. Update parameters with SGD

EP is equivalent to BPTT: strong guarantee to solve complex tasks. Ernoult *et al*. [157] demonstrated that the parameters updates prescribed by EP were equivalent to those of Backpropagation Through Time (BPTT). BPTT is the equivalent to BP for recurrent neural networks, making it the reference algorithm for the tasks EP can solve.

The work of Ernoult *et al*. [157] has been crucial to theoretically and experimentally demonstrate that indeed EP prescribes parameters updates that are the exact gradient of the loss with respect to those parameters. This is very stimulating because we know that gradient-based approaches that are used to optimize a global objective function give state-of-the-art results on more complex tasks than MNIST such as CIFAR-10 [158], ImageNet [154], ... This equivalence is a guarantee that EP can theoretically achieve the performance of BPTT on those tasks. A recent work [155] has shown promising results on ImageNet 32x32 which contains as many classes as ImageNet but the images are down-sampled from 256x256 pixels to 32x32 images.

3.2.3 . Example: training a simple layered architecture

For clarity, we now apply the EP framework to a simple layered fully-connected architecture with one hidden layer (such as depicted in Fig. 3.8). We still denote the state of the system s but to be clearer we assign $s = \{h, y\}$ where h are the hidden units and y the output units. ρ is the activation function of the neurons, x is an input and $\theta = \{W, b\}$ are the parameters - weights and biases - of the system. The energy of such a system is a Hopfield-like energy:

$$E(\theta, s, x) = \sum_i s_i^2 - \frac{1}{2}\sum_{i \neq j} W_{ij}\rho(s_i)\rho(s_j) - \sum_i b_i \rho(s_i) \tag{3.24}$$

where the weights W_{ij} encompass both the pure feedforward weights between the input units - that are fixed during evolution - and the hidden units, $W^{x,h}$ and the weights between the hidden and the output units, $W^{x,h}$.

Then, following Eq. 3.16 applied to Eq. 3.24, the dynamics of the hidden and output units during the free phase units read as:

$$\begin{cases} \frac{dh_i}{dt} = -\frac{\partial E}{\partial h_i} = -h_i + \rho(h_i)\left[\sum_{j\in x} W_{ij}^{x,h} x_j + \sum_{j \in y} W_{ij}^{y,h}\rho(y_j) + b_i^h\right] \\ \frac{dy_i}{dt} = -\frac{\partial E}{\partial y_i} = -y_i + \rho(y_i)\left[\sum_{j\in h} W_{ij}^{h,y}\rho(h_j) + b_i^y\right] \end{cases} \tag{3.25}$$

The input of the hidden and the ouput units clearly behave differently: hidden units receive input from both the input units and the output units whereas output units only receive signal from the hidden units during the free phase. The system eventually converges to a state that minimizes its energy.

Starting from this first equilibrium state, we add a spring-like force to the output neurons that nudge them toward their target state $\hat{y}_i$ where $\hat{y}$ is the one-hot encoded target vector:

$$\begin{cases} \frac{dh_i}{dt} = -\frac{\partial E}{\partial h_i} = -h_i + \rho(h_i)\left[\sum_{j\in x} W_{ij}^{x,h} x_j + \sum_{j \in y} W_{ij}^{y,h}\rho(y_j) + b_i^h\right] \\ \frac{dy_i}{dt} = -\frac{\partial E}{\partial y_i} = -y_i + \rho(y_i)\left[\sum_{j\in h} W_{ij}^{h,y}\rho(h_j) + b_i^y + \beta(\mathbf{y_i} - \hat{\mathbf{y}}_\mathbf{i})\right] \end{cases} \tag{3.26}$$

Thanks to the recurrent synapses, the change in activity that arises at the output layer propagates in the system, thus the system eventually settles to the second equilibrium point. If $y_i < \hat{y}_i$ then the neuron will be dragged up. If $y_i > \hat{y}_i$ then the neuron will be dragged down. Both changes in activity in the output layer will make the output units closer to their target state and thus reduce the cost computed at the output.

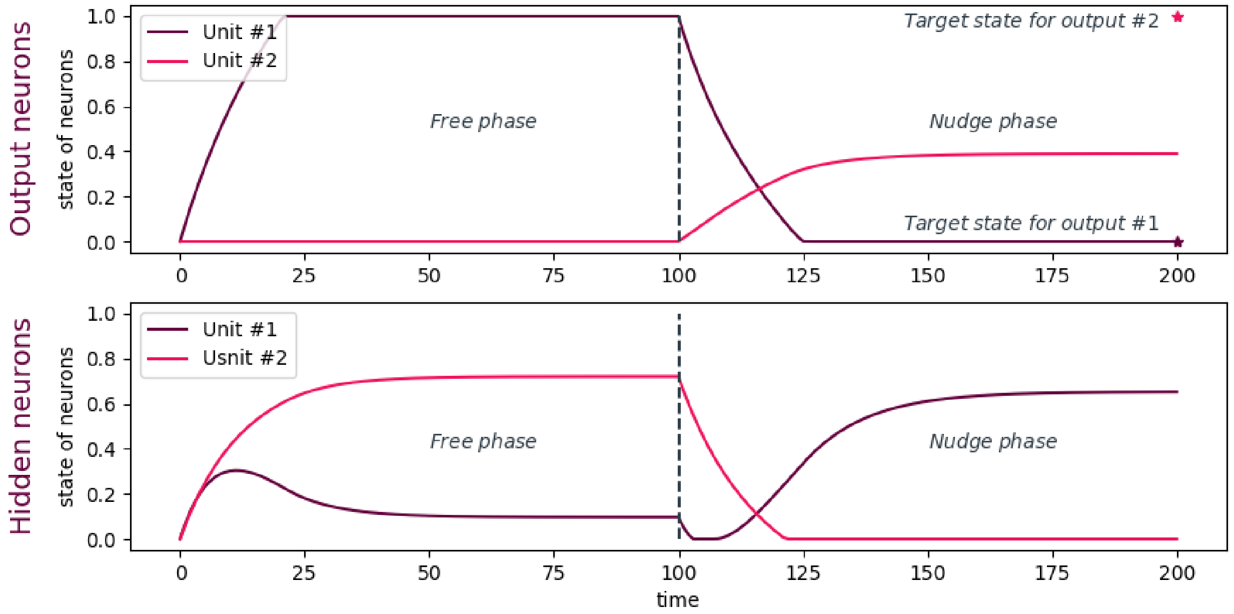


Figure 3.10: Temporal dynamics of the hidden and output neurons in a 1-hidden layer architecture where we have 2 inputs nodes, 2 hidden neurons and 2 output neurons. From $t = 0$ to $t = 100$ the system performs the free phase. At $t = 100$ we turn on the nudging factor so the output units get closer to their target state. Thus this change in activity propagates backward and the hidden units are affected.

We now derive the learning rules for the weights:

$$\begin{cases} \Delta W_{ij}^{x,h} = -\frac{\partial \mathcal{L}}{\partial W_{ij}^{x,h}} = \frac{1}{\beta}\left(x_j * \rho(h_i^{*,\beta}) - x_j * \rho(h_i^{*,0})\right) \\ \Delta W_{ij}^{h,y} = -\frac{\partial \mathcal{L}}{\partial W_{ij}^{h,y}} = \frac{1}{\beta}\left(\rho(h_i^{*,\beta}) * \rho(y_j^{*,\beta}) - \rho(h_i^{*,0}) * \rho(y_i^{*,0})\right) \end{cases} \tag{3.27}$$

and the biases:

$$\begin{cases} \Delta b_i^h = -\frac{\partial \mathcal{L}}{\partial b_i^h} = \frac{1}{\beta}\left(\rho(h_i^{*,\beta}) - \rho(h_i^{*,0})\right) \\ \Delta b_i^y = -\frac{\partial \mathcal{L}}{\partial b_i^y} = \frac{1}{\beta}\left(\rho(y_j^{*,\beta}) - \rho(y_i^{*,0})\right) \end{cases} \tag{3.28}$$

3.2.4 . EP has triggered great attention: benchmark

EP is an architecture-agnostic learning framework as long as the architecture and the operations involved can be described in terms of an energy function. It has been applied to different standard deep learning architectures such as Fully-connected Neural Networks or Convolutional Neural Networks. EP has been also demonstrated on different tasks: MNIST [151] - gray-scale dataset with 10 classes, 60k training images and 10k testing images, CIFAR-10 [158] - RGB dataset with 10 classes, 60k training images and 10k testing images and recently ImageNet 32x32 [159] - RGD dataset which has been down-sampled from ImageNet with 1.2M training images, 50k testing images distriubted in 1000 classes. EP has also been derived for spiking neurons that suit most of existing neuromorphic platforms. We detail in the following paragraphs some work around these implementations to

illustrate the flexibility of EP.

Fully-connected Neural Networks. EP has been initially applied to fully-connected architectures on MNIST with 1,2 and 3 hidden layers [14]. Training such architecture is quite straightforward with EP. All operations required for the convergent recurrent neural network are basically the same as for pure-feedforward fully-connected neural networks. The only difference is that hidden layers get inputs coming from the next layer, which makes the network recurrent:

$$input_{s_i^l} = \sum_{j \in l-1} W_{ij}^{l,l-1} \rho(s_j) + \sum_{j \in l+1} W_{ij}^{l,l-1^T} \rho(s_j) \tag{3.29}$$

This architecture have been extensively used to demonstrate the relevance of using EP to train fully-connected neural networks: [17], [150], [155], [160]–[164].

Convolutional Neural Networks. Convolutional Neural Networks perform more efficiently than Fully-connected Neural Networks on computer vision tasks.

EP has also been applied to train convolutional architectures on MNIST [157], on CIFAR-10 [17], [165] and more recently [155] trained a large convolutional neural network on ImageNet 32x22 [159].

The difficulty for applying convolutional architectures in EP was to describe the symmetric operations of the forward convolution and pooling operations that are extensively used in Convolutional Neural Networks. Ernoult et al. [157] came up with Transpose Convolution and Inverse Pooling operations. Given a feature map tensor y and the weights for convolution w such as $y = x \star w$ - where $\star$ denotes the convolution operation - then the transpose convolution operation is $\star^{-1}$ and $x = y \star^{-1} w$. In a similar way we can do the inverse pooling operation. The forward max pooling operation loose spatial information because of the max operation. For doing the inverse operation, one has to store the index of the maximal element of the pooling window used for the forward operation.

With these methods Ernoult at al. successfully trained the first convolutional neural network on MNIST. They have shown the best test accuracy reported with EP with this architecture with $\approx 1\%$ test error. Laborieux et al. [165] scaled up to CIFAR-10 by using a symmetric nudge. In fact, they have shown that nudging the network with $\beta > 0$ only was giving false estimate of the gradient. Scellier & Bengio initially proposed to use a random-sign β [14] to better estimate the gradient. Despite the fact that this method works well for fully-connected architecture, it is not accurate enough to reduce the bias in the gradient computation for training convolutional neural networks. Laborieux et al. nudge the system with first $+\beta$ and then with $-\beta$. Then the gradient is computed given the two resulting equilibrium states.

The backlash of training such architecture is the non-locality of the learning rule. Indeed the learning rule for the convolutional weights between two features maps s_l and s_{l+1} is given by:

$$\Delta w_{l,l+1} = \frac{1}{\beta} \left(\mathcal{P}^{-1}(s_{l+1}^{*,\beta}) \star s_l^{*,\beta} - \mathcal{P}^{-1}(s_{l+1}^{*}) \star s_l^{*} \right) \tag{3.30}$$

where $\mathcal{P}^{-1}$ denotes the inverse pooling operation and $\star$ the convolution operation. We conjecture that even if the learning rule is non-local: it requires a convolution operation between two feature maps that is a highly non-local operation, it still relies on the states of two adjacent layers and thus with clever hardware circuitry we could compute the updates quasi-locally.

In [155], the authors propose to use complex neurons so a sinusoidal nudge can be applied. They show that the gradient thus computed is even more accurate compared to BP. They show that this new dynamics improves the accuracy on CIFAR-10. They also succeed in training a convolutional neural network on an unprecedented large dataset with EP which is ImageNet 32x32 [159]. They report an accuracy as high as that reached with backpropagation.

Spiking Neural Network. Most of the work about EP has been done with neurons that have a continuous non-linear activation function: using mostly the hardsigmoid activation ($f(x) = max(0, min(1, x))$) function that saturates bellow 0 and above 1, which is necessary for having a convergent system in simulations. But a few works have focused on spiking neural networks (SNN) trained with EP. Spiking neural networks are neural network where the activation is no longer continuous but rather neurons emit spikes depending on whether the "membrane potential" of the neurons crosses a threshold. They are more bio-realistic. One great advantage of such networks is that the events (spikes) are quite sparse and make these networks very energy-efficient. One major drawback was the lack of gradient-based training algorithm suitable for hardware SNNs.Existing gradient-based training algorithms for SNNs rely on external circuits that compute "surrogate gradients" that require complex peripherical circuit with low energy-efficiency. Thus SNN implemented on-chips have thus been mainly trained with STDP-like or heuristic learning rules which are empirical and do not optimize an objective function [74], [77], [166], or have been used for very low-power inference platform only [167].

The works with EP have shown that SNNs can be trained with the same dynamics for both the inference and the error-backpropagation phases, which makes the system SNN+EP very appealing for hardware implementations. The work of Mesnard *et al*. [168] is interesting because instead of storing the first equilibrium state - or firing rate - to compute the gradient after the nudge phase, they update the parameters in two steps. They do the negative update given the first equilibrium state just after the free phase. Then they nudge the system and do a second update based on the second equilibrium state. This kind of parameters update is very memory-efficient but requires very low learning rate to not modify too much the dynamics of the nudge phase compared to the free phase.

Recently, Martin *et al*. developed EqSpike [169] (of which I am co-author), which is a version of EP designed to train SNNs. In EqSpike, the idea is to leverage the possible overlap of spikes that go forward and backward through the same synaptic device to trigger a weight update during the nudging phase. This results in an *intrinsic learning* where no external signal has to be applied at the synapse level in order to update the weight. Then no memory is required to store the state of neurons reached at the end of the free-phase. The learning rule is that of Continual EP [170] where the update is done at each event during the nudging phase (spikes that overlap and trigger a weight update). EqSpike has been developed with the goal to use memristors for a hardware implementation [171]. Indeed, the update rule of memristor is highly non-linear with a threshold that the superposition of spikes can reach in order to trigger the update of the memristor in-situ with no global update signal. EqSpike reaches state of the art accuracy on MNIST with a shallow architecture. EqSpike is very interesting for hardware applications as we only need to have access to the input and the output nodes in order to train the system to do image recognition.

3.3 . Binary Neural Networks

Usually, neural networks are parametrized with full precision variables (mostly 32 bits variable for which GPUs are most optimized for[5]). Full-precision synaptic weights enable to discriminate inputs to which a neuron is more or less sensitive covering all shades in between. Also, neural activations are mostly full precision. This allows a graded non-linear response of the neuron with regard to its input: ex: sigmoid, ReLU, tanh, ... However, such neural networks require a huge amount of memory for both the inference and the error-backpropagation phases as described in Section 1.1.3. This has precluded for a long time the training but also the deployment of large models on edge devices where the memory and computational budgets are very limited. One solution, that is the extreme case of quantization of the variables (see Section 1.3.2), has been to reduce the precision of those variables to the case where they are binary. We cannot go beyond binary for synapses: we need to know if a synapse is excitatory $(+1)$ or inhibitive (-1). Similarly, we cannot go beyond binarization for neural activations: we need to know if a neuron "fires" $(+1)$ or not (-1). Such neural networks with binary synaptic weights and/ or binary activations are called Binary Neural Networks. BNNs have been a great milestone for embedding large deep learning models onto edge devices. However, as we will see, the binarization of those model holds only for the inference phase. The optimization of BNNs is not straightforward and still requires a full-precision variable per binary synapse. This point has precluded until now the training of BNNs on the edge. Thus, the training of BNNs is done on energy-consuming machines and only the trained BNN model is transferred to edge devices. Nevertheless, one can leverage elementary operations such as XNOR, Popcount, ... implemented in basic language to perform the operations of a BNN which allows very efficient and fast hardware implementations for inference.

3.3.1 . A historical introduction to Binary Neural Networks and definitions

Motivated by low-latency and energy-saving neural networks, Courbariaux *et al*. published the first BNN in 2015 [18]. They initially binarized only the weights at inference (or run) time and introduced the binarization as:

$$w_b = sign(w) \tag{3.31}$$

where w stands for a set of real-value weights (32 bits floating points weights) that are used for the optimization as we will see in section 3.3.2. w_b are the binary weights that are used for the inference. It is the first work that trains from scratch a BNN with binary weights at inference time. This work reduces the memory required to store the model by 32 due to the binarization of the weights but only reduces by 2 the computational power required for the inference (Fig. 3.11). The vector-matrix multiplication between the inputs and the weights is now down to changing (or not) the sign of the input and then do the addition of a vector of full-precision data.

Pushing further the binarization, Hubara et al came up in 2016 with Binary Neural Network [19] where the neural activations are also binarized along with the synapses. This work has been a major breakthrough in the BNN field as they were the first work to use backpropation to train BNNs despite the binary activation that is non-derivable. We will see in Section 3.3.2 how they succeeded in using backpropagation with the help of the Straight Through Estimator (STE) [172].

XNOR-Net [173] was published little after and is a successful tentative to achieve near-sota accuracy on large scale dataset such as ImageNet that previous approaches failed at. Due to their very limited precision on the parameters and activations, BNNs where thought to be uncompetitive

[5] NVIDIA blog

to complex benchmark tasks until this paper. BNNs having both binary weights and activations drastically reduce the memory and the computational power required for the inference (Fig. 3.11).

In binary neural networks (weights and activations), all operations required for doing the multiply and accumulate (MAC) operation, which is the costliest operation during inference, can be reduced to elementary operations between binary vectors/ matrices. In Fig. 3.12 we detail the elementary operations involved in the binary MAC operation: the bitwise $XNOR$ product and the $popcount$ operation[6]. Until now we depicted the binary variables (weight or activation) with ± 1 but to be correct, they are stored with bits in CMOS hardware thus they are $0/1$. That is why we use the $XNOR$ product instead of the multiplication we do with binary variables $-1/+1$ (Table 3.2, Table 3.3).

Table 3.2: $XNOR$ truth table when inputs are ± 1 is equal to the dot product between the inputs

$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_1 * x_2}$
-1	-1	1
-1	1	-1
1	-1	-1
1	1	1

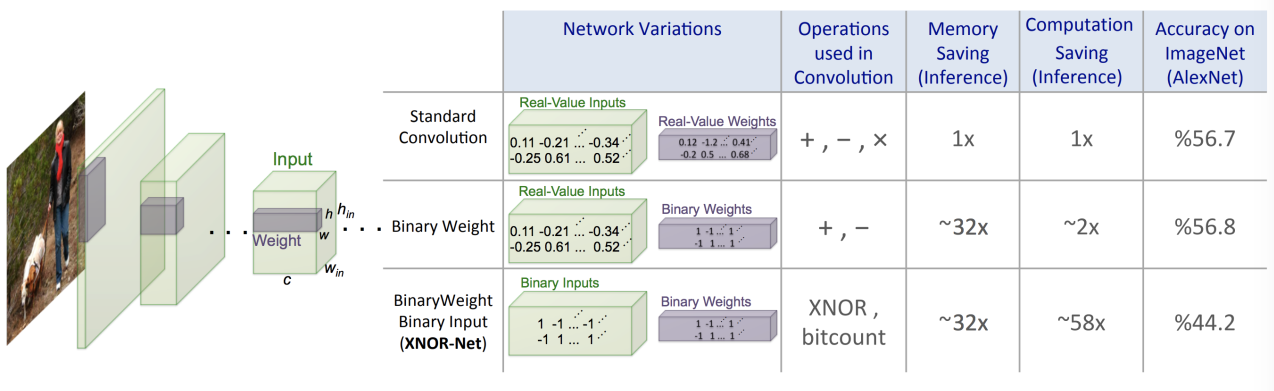
Table 3.3: $XNOR$ truth table when inputs are $0/1$ - as are stored binary variable in CMOS. The correspondence is immediate between the two methods when we assert the following translation: $-1 = 0$ and $1 = 1$

$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{XNOR(x_1, x_2)}$
0	0	1
0	1	0
1	0	0
1	1	1

The MAC operation between a binary vector and a binary weight matrix is done in a three steps procedure (Fig. 3.12): first we compute the $XNOR$ product between the input binary vector and each column of the binary weight matrix. Then we apply the $popcount$ elementary function that computes the number of bits in a binary vector. Finally we offset the popcount operation by the size of the output vector to get the results of the MAC operation. All these operations are very time and energy efficient to realize with CMOS hardware and they are already implemented at a very low level. The theoretical speed-up is 32 (Fig. 3.11) because we can now cast binary activations and binary weights into 32 bits vectors usually used to represent a single real-value parameter or activation. The ability to leverage low-level basic operations ($XNOR$, $popcount$) is the main reason that made BNNs so popular for building efficient and fast inference engines.

The input of BNNs is often still real-valued. The product of real-value inputs and binary weights produces a real-value pre-activation for the first layer which is then binarized after the sign activation is applied. In practice this is not a huge problem as the input is fixed and can be treated as a fixed point variable and not floating point as are the real-value weights because they are updated during the optimization process. However, if one really wants to process binary inputs, there have been some work about binarizing the input in a stochastic way [174] - the probability of a pixel being 1

[6]In practice, the original popcount operation is a bit modified to take into account possible negative outputs: $modified_popcount = 2 * popcount(XNOR(x, W) - size(popcount(XNOR(x, W))$ - source: Apple developper website



	Network Variations	Operations used in Convolution	Memory Saving (Inference)	Computation Saving (Inference)	Accuracy on ImageNet (AlexNet)
Standard Convolution	Real-Value Inputs 0.11 -0.21 ... -0.34 -0.25 0.61 ... 0.52; Real-Value Weights 0.12 -1.2 ... 0.41 -0.2 0.5 ... 0.68	+ , − , ×	1x	1x	%56.7
Binary Weight	Real-Value Inputs 0.11 -0.21 ... -0.34 -0.25 0.61 ... 0.52; Binary Weights 1 -1 ... 1 -1 1 ... 1	+ , −	~32x	~2x	%56.8
BinaryWeight Binary Input (**XNOR-Net**)	Binary Inputs 1 -1 ... -1 -1 1 ... 1; Binary Weights 1 -1 ... 1 -1 1 ... 1	XNOR , bitcount	~32x	~58x	%44.2

Figure 3.11: from XNOR-Net [173] - This table emphasizes the differences between a standard neural network where both parameters and activations are real-valued, a binary-weights only BNN and a fully binarized BNN. It also highlights the memory and computational resources savings due to the binarization of both the weights and the activations. Despite extreme quantization, BNNs still perform well at ImageNet.

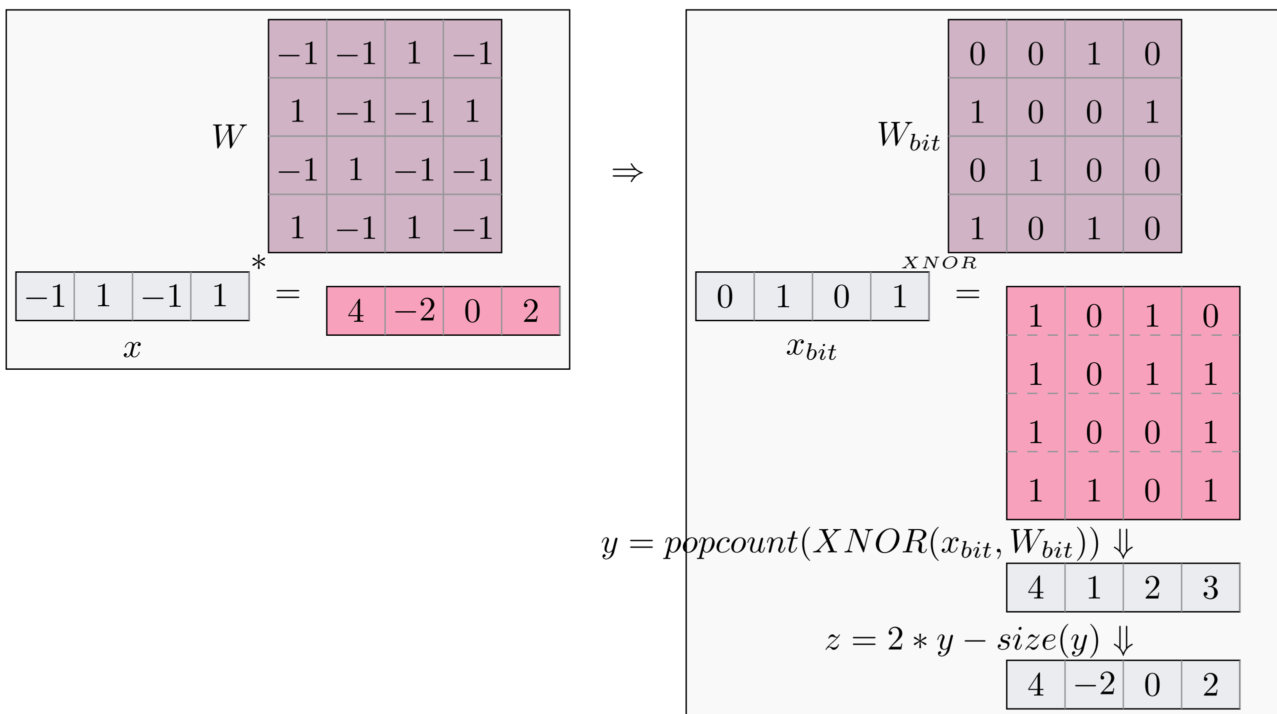


Figure 3.12: Left : multiply and accumulate operation between a binary input vector and a binary weight matrix where binary variables are denoted as +/-1. Right: with CMOS hardware, binary variables are stored with bits that can take 0/1 values.

depending on its intensity. This has shown to perform as well as with real-value inputs on MNIST and CIFAR-10. Recent work [175] on a new kind of stochastic binarization with multi-channels input images (such as RGB images from CIFAR-10) has shown to perform even better for CIFAR-10.

3.3.2 . Challenges with BNNs: saturation and optimization

In this section we first describe how we can optimize neural networks when the synapses and the activations are binarized. Binary synapses prevent the direct use of stochastic gradient descent (see Section 1.1.3) as we can not do small steps in the gradient direction to update the weights as they only have two possible states. We will highlight two current optimization methods that succeed in optimizing binary synapses. Also, the binary activation function has a zero gradient almost everywhere (except for the step where it is infinite). So methods have been developed to overcome this issue that prevents gradient-based optimization from working well. Finally, we will examine the impacts of having binary weights in a neural network. We will see two methods that aim at reducing this impact that can cause neural saturation and also prevent the gradient-based optimization method to function.

Optimization with latent weights

The supervised optimization step of the parameters of a neural network, Stochastic Gradient Descent optimization method, relies on smooth small steps in the direction of the gradient of the cost function. In practice it works because the real-value parameters can take all the values in a given range. This kind of optimization has proven to perfom very well for solving complex tasks.

But when the parameters are binarized, such optimization techniques do not hold. Or at least not directly. Indeed, the smooth small steps cannot be realized as the parameters can only take the values -1 and $+1$. The only possible steps are ± 2 steps. Such large steps could cause instability in the dynamic of the network and thus alter or even make impossible the training with such an approach.

The approach that has been initially chosen for optimizing BNNs with gradient-based approach is to use two sets of weights [18]: one set of real-value weights that can hold the optimization step and one set of binary weights that are the sign of the real-value weights. The binary weights are used for inference and for computing the gradient. The real-value weights are called "latent weights" because they are somehow hidden to the model.

The second challenge is to compute derivatives despite the fact that the activation function of the binary neurons - the sign function - is non-derivable. The solution that has been found is to use the "straight-through-estimator" [172] - STE - to recover a kind of derivative for the activation function of the neurons. If we assume $a_i = sign(x_i)$ is the activation function of neuron i and x_i its pre-activation and that we can compute $g_{a_i} = \frac{\partial \mathcal{L}}{\partial a_i}$, then the Straight-Through-Estimator gives the following gradient for the pre-activation: $g_{x_i} = g_{a_i} * 1_{|x_i|<1}$. The gradient computed with STE is the gradient of the hard-tanh activation function that cancels the gradient when the pre-activations are too large, which can be an issue for training.

Optimization without latent weights: BOP

Although latent weights in BNNs accumulate weight updates, Helwegen *et al*. [160] suggested that they were not weights in the strictest sense (they are not used at run time) but were only meant to convey inertia for the optimization of the binary weights. Based on this insight, Helwegen *et al*. [176] proposed a Binary Optimizer (BOP) which flips the binary weights solely based on the value of their associated momentum (without latent weights *per se*): if the momentum is large enough and crosses a threshold from below, the binary weight is switched. By using one full precision variable (the momentum) instead of two per synapse (the latent weight and the gradient), BOP is of definite interest to reduce the memory footprint of BNN training, which is why our work heavily relies on this technique (see Section 4.2). BOP has two hyperparameters: the value of the flipping decision

Algorithm 4 Training algorithm of a BNN with latent weights - we emphasize the binary weights in bold pink police and the latent (real-value) weights in bold blue

1: **Inputs:** $\theta = \{\boldsymbol{W}, b\}$, x, $\hat{y}$
2: **Outputs:** $\boldsymbol{W}^b$
3: $\boldsymbol{W}^b \leftarrow sign(\boldsymbol{W})$ ▷ 1. Forward Propagation
4: $A_0 = X$
5: **for** $layer = 1$ to L **do**
6: $x_{layer} \leftarrow A^b_{layer-1} * \boldsymbol{W}^b_{layer}$
7: $A_{layer} \leftarrow BatchNorm(x_{layer}, \alpha_{layer}, \beta_{layer})$
8: **if** $layer < L$ **then**
9: $A^b_{layer} = sign(x_{layer})$
10: **end if**
11: **end for**
12: Compute $\ell = \ell(A_L, \hat{y})$ ▷ 2. Cost Function
13: Compute $g_{A_L} = \frac{\partial \ell}{\partial A_L}$ ▷ 3. Backpropagation
14: **for** $layer = L$ to 1 **do**
15: **if** $layer < L$ **then**
16: $g_{A_{layer}} = g_{A^b_{layer}} \circ 1_{|A^b_{layer}| \leq 1}$ ▷ Straight Through Estimator
17: **end if**
18: $(g_{x_{layer}}, g_{\alpha_{layer}}, g_{\beta_{layer}}) = BackBatchNorm(g_{A_{layer}}, x_{layer}, \alpha_{layer}, \beta_{layer})$
19: $g_{\boldsymbol{W}^b_{layer}} = g^\top_{x_{layer}} A^b_{layer-1}$
20: $g_{A^b_{layer-1}} = g_{x_{layer}} \boldsymbol{W}^b_{layer}$
21: **end for**
22: $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta * g_{\boldsymbol{W}^b}$ ▷ 4. Update real-value parameters

threshold τ, and the adaptativity rate γ. The larger τ, the less frequently the binary weight flips and the slower the learning. On the other hand the larger γ, the more sensitive the momentum to a new gradient signal, the more likely will be a flip of a binary weight. The BOP algorithm is summarized in Alg. 5.

Binary weights cause neural saturation: use batch-normalization and/ or scaling factors as counter-balance

Standard neural networks have weights that are often evenly or normally distributed in a limited range around 0. Thus, a slight imbalance between the absolute magnitude of positive weights and the absolute magnitude of the negative weights does not have a dramatic influence on the activation function of a neuron. It is because the average value of a weight weighting an input of a neuron is very small. However, when weights are binary, their impact on the activation function of a neuron is much more significant, especially when the activation they weight is binary (see Table 3.2 and Table 3.3). That could be a problem because it can cause neurons to saturate and thus slow down the training of the network because a specific saturated neuron will be less likely to feel a difference at its input after few updates of its synaptic weights when it is saturated. Indeed, whether the activation is continous or binary, if the neuron saturates in the forward pass, then the gradient of its activation with respect to its pre-activation is zero: $\frac{a_i}{x_i} = 0$ which results is a zero weight gradient. As emphasized above, for BNNs it is not straightforward what is saturation because the activation

Algorithm 5 Training algorithm of a BNN with BOP [176] - we emphasize the binary weights in bold pink police and the momentum (real-value) in blue

1: **Input**: g, m, $\boldsymbol{W^b}$, γ, τ.
2: **Output**: m, $\boldsymbol{W^b}$.
3: $m \leftarrow \gamma g + (1-\gamma)m$
4: **for** $layer \in [1, L]$ **do**
5: **if** $|m_i| > \tau$ and sign(m_i) = sign$(\boldsymbol{W_i^b})$ **then**
6: $\boldsymbol{W^b_{layer}} \leftarrow -\boldsymbol{W^b_{layer}}$
7: **end if**
8: **end for**

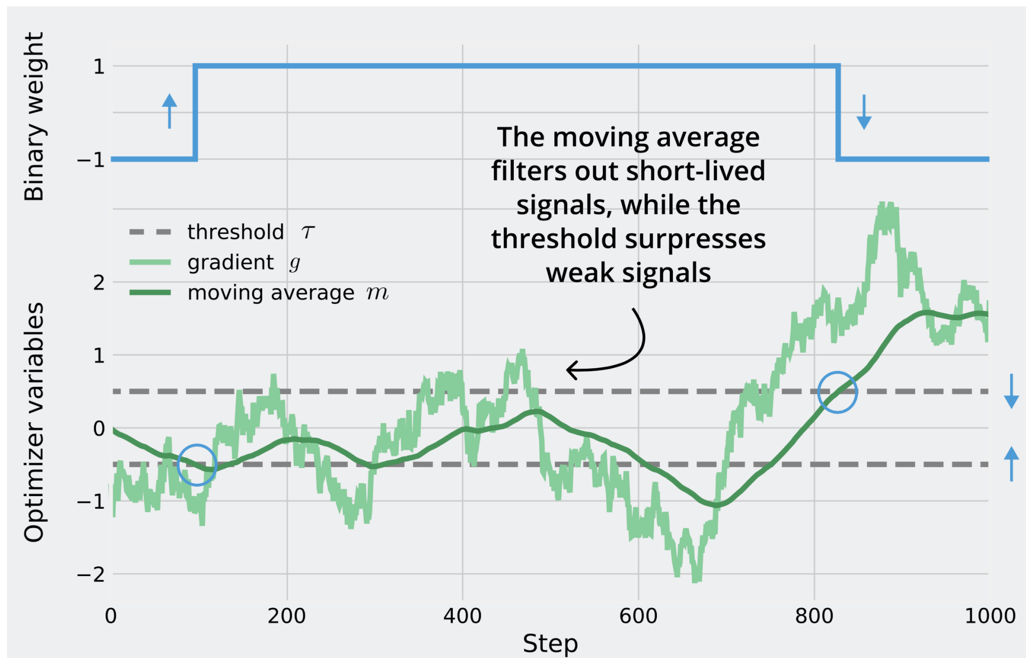


Figure 3.13: BOP algorithm - Taken from [176]

has only two states. In fact, the saturation applies to the STE [172] that we use to compute the pseudo-gradient for the binary activation. So methods that reduce the risk of saturation are intensively sought after.

Batch-Normalization. Since early work on BNNs, batch-normalization (BN) [177] has been used as the critical tool to mitigate the impact of binary weights on the activation function of neurons in BNNs - whether the activation is analog (sigmoid, ReLU, tanh, ...) or binary. BN works in the following way: let's consider a vector x_i, where the index i denotes the $i-th$ sample of a mini-batch of size m, that describes the pre-activations or the activations of a layer in a neural network (the choice of applying BN before or after the activation depends on the authors), and y_i the output of the BN operation, then the operations involved in computing are:

$$y_i = BN_{\gamma,\beta}(x_i) \tag{3.32}$$

$$\mu_{batch} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{3.33}$$

$$\sigma^2_{batch} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{batch})^2 \tag{3.34}$$

$$\hat{x_i} = \frac{x_i - \mu_{batch}}{\sqrt{\sigma^2_{batch} + \epsilon}} \tag{3.35}$$

$$y_i = \alpha * \hat{x_i} + \beta \tag{3.36}$$

By centering and normalizing x_i, the impact on the activation of binary weights is drastically reduced. BN is a good technique to ensure units are not saturated.

Scaling Factors. We now introduce a second method to mitigate the effect of the binary weights on the neural saturation. This method as been introduced in XNOR-Net [173] and boils down to scaling down the binary weights to same values α that are called "scaling factors". In XNOR-Net, the scaling factors have been intially derived as scalars that minimize the loss of information when the real-value weights $\mathbf{M}$ are binarized into $\mathbf{B}$.

To estimate the value of α, we restate the goal of α in term of an objective function to minimize:

$$J(\mathbf{B}, \alpha) = ||\mathbf{W} - \alpha\mathbf{B}||^2 \tag{3.37}$$

We now want to minimize $J(\mathbf{B}, \alpha)$ with the condition that $\mathbf{B} \in \{-1; +1\}$ and $\alpha \in \mathbb{R}^+$.

To find $\mathbf{B}^*$ and α^* we expand Eq.3.37:

$$J(\mathbf{B}, \alpha) = \mathbf{W}^\top\mathbf{W} + \alpha^2\mathbf{B}^\top\mathbf{B} - 2\alpha\mathbf{M}^\top\mathbf{B} \tag{3.38}$$

Here $\mathbf{B}^\top\mathbf{B}$ is a constant as well as $\mathbf{W}^\top\mathbf{W}$. The optimal binary weights matrix $\mathbf{B}^*$ is given by $\mathbf{B}^* = argmax_{\mathcal{B}}(\mathbf{M}^\top\mathbf{B})$ and the solution of that equation is $\mathbf{B}^* = sign(\mathcal{M})$[7]

To get the optimal value for α, we derive Eq. 3.38 with respect to α and set it to zero to get:

$$\alpha^* = \frac{\mathbf{M}^\top\mathbf{B}^*}{n} \tag{3.39}$$

where n is the number of synaptic elements in both $\mathbf{B}$ or $\mathbf{M}$. But as $\mathbf{B}^* = sign(\mathbf{M})$, it follows:

$$\alpha^* = \frac{\mathbf{M}^\top sign(\mathbf{M})}{n} = \frac{\sum_i |W_i|}{n} = \frac{1}{n}||\mathbf{W}||_{\ell 1} \tag{3.40}$$

Eq. 3.40 explicitly states that α^* acts as a scaling factor that downscales the binary weights and that the scaling depends on the amplitude of all the weights in the weight matrix.

Also, as in practice $0 < \alpha < 1$, scaling factors scale down the binary weight to $\pm\alpha$ and act as a kind of regularization that avoids neural saturation.

With the use of both BN and scaling factors in BNNs, we have tools to mitigate the impact of the large scale of binary weights on the activation of the neurons.

[7]See [173] for detailed derivations.

3.3.3 . BNNs have triggered great attention for custom hardware

The simplicity of the operations supporting the inference of BNNs triggered great attention for custom hardware implementations. As described in introduction (Section 1.3.3), large efforts are currently made in order to design efficient hardware to support the operations involved in neural networks. As the extreme case of quantization, BNNs rely on the elementary quantities used to perform computation on CMOS hardware, thus such hardware is expected to speed up the computations of BNNs. We will describe the elementary operations involved in BNNs and see a few custom hardware implementations with CMOS substrates. Then, back to the central topic of the thesis, we will discuss non-conventional approaches which leverage the behavior of non-volatile components for doing in-memory computing.

Standard approaches.

BNNs were intially motivated by the potential speed-up and low-memory requirements that could offer optimized software implementations with low-level code. Courbariaux et al. already detailed a BNN-optimized kernel running on GPU that exhibits a x7 speed up compared to a baseline kernel to run a multi-layer perceptron on MNIST. Their optimized kernel uses intensively the basic operations of $XNOR$ and $popcount$ to reach this level of accuracy but still runs on general-purpose hardware that can slow down the operations.

Alternatively, there has been lot of works [178]–[181] around building custom CMOS-based hardware for the inference of BNNs. These approaches focus on a specific BNN architecture so every operation can be highly optimized.

Both alternatives have triggered great attention from companies for the gains BNNs exhibit at doing fast computation on the edge where computational and memory budgets are limited. We can cite XNOR.AI, a startup that was founded by authors of XNOR-Net paper and was acquired by Apple in early 2020 for $200 Billions[8]. More recently there is PlumerAI[9] that aims at developing edge solutions for person detection or speech recognition with the help of BNNs.

Unconventional approaches.

In the same way as presented in the introduction (Section 2.1), approaches using emerging components, mostly relying on non-volatile memory (NVM) such as RRAMs or MRAMs, have emerged for implementing BNNs in hardware [174], [182]–[185].

Instead of using the continuum of states between the low resistance state (LRS) and the high resistance state (HRS), as typically done with memristors, hardware implementations of BNNs with NVMs use only the two extreme states to encode the binary states (0/1). When used in binary mode, NVMs are extremely attractive approaches due to their low operating power, their ability to perform in-memory computing and a reduced intra-device variability compared to the full-analog mode.

Emerging non-volatile memories such as MRAMs or RRAMs have been intensively looked out for the operating low-power required to change their state (from 0 to 1 going from Low Resistive State (LRS) to High Resistive State (HRS) = SET or conversely = RESET). Hardware implementations of BNNs with such non-volatile memories could allow to run inference engines with very little power

[8] https://techcrunch.com/2020/01/15/apple-buys-edge-based-ai-startup-xnor-ai-for-a-reported-200m
[9] https://plumerai.com/

energy like on edge devices. However, such devices are still mainly experimental[10] and are prone to intra-device and device-to-device variability (see Section 2.1). This undesirable behavior can alter the performance of the BNN that is optimized for a given set of binary parameters and could not endure such failure at programming time. We can measure how much a BNN is sensitive to programming-errors with the bit-error rate tolerance where after training we voluntarily flip the parameters according to some (low) probability and measure the accuracy on the testing dataset to see how the bit-error rate affects the BNN. It has been demonstrated [183] that surprisingly BNNs are particularly tolerant to bit errors confirming the relevance of the binary NVMs approach. Moreover, there has been intensive work to mitigate device-to-device variability. The most common option is to increase the current of the SET pulse but in return it increases the power required to operate the hardware BNN. It turns out that running BNNs with unconventional NVMs seems to be about finding the sweet spot between tolerated bit error rate and the power required to reach this error rate. However, binary NVMs show much less variability than continuous valued emerging components and that make them good candidates for large-scale hardware implementations.

Conclusion

BNNs have triggered great attention because they rely on computations that leverage elementary operations on binary vectors. Early works on dedicated computing kernels have shown huge speed ups [18]. Additionally, there have been multiple works to build faster and lower-power inference engines with custom hardware implementations. The use of NVMs for this application allows even better performances as no data movement is required for the inference. However, the training of such systems still relies on backpropagation and the energy and memory required to train those system can be overwhelming compared to the energy used for doing inference.

Nevertheless, BNNs are a promising approach to design low-power chips based on emerging nano-devices utilized in a binary regime which drastically reduces the undesirable effects of their variability. In the next chapter we will see how we can bypass backpropagation by using EP to compute the parameters updates of BNNs, thus making hardware trained by EP much easier to implement.

3.4 . Summary

We have introduced in Chapter 2.1 the concept of computing with the property of energy minimization that physical system naturally have. This concept is very appealing as the physical system naturally performs the mapping input $\leftrightarrow$ output by dynamically evolving toward the states of higher probabilities or equivalently the states of lower energy, thus the expression computing through energy minimization.

However it was not clear how one could have such an energy-based hardware in which we can tune the parameters on-chip in order to perform learning thanks to its dynamics. In this section we introduced the Ising Machines that are pure energy-based hardware as they are hardware designed to find the ground state of an Ising Hamiltonian (introduce in Section 2.2.2). Ising Machines are hardware that realize the intuition of Hopfield [9] that was to compute with coupled spins. However, we have also seen that, due to the lack of good learning algorithms, Ising Machines are confined to solve combinatorial optimization problems. The rise of fast reconfigurable Ising Machines such as

[10]Despite the fact that STT-MTJ are already commercialized, their small OFF/ON ratio prevents using them in crossbar arrays. However they can still be used for performing in-memory computing.

the D-Wave Ising Machine makes it possible to demonstrate that we can embed and train such a neural network on the Ising Machine. We will use EP to supervisely train the D-Wave Ising Machine in Chapter 5.

Energy-based models have also been overshadowed by the deep learning models. Indeed, the latter are trained with backpropagation that allows the gradient-based optimization (stochastic gradient descent) of a global cost function conversely to energy based-models that have been mostly trained with local learning rules that optimize local-only objectives which limits the potential of such networks. We have introduced Equilibrium Propagation as an alternative to backpropagation to compute the gradient of a global cost function for such energy-based models that only relies on the property of these systems to evolve toward a minimum of the energy function. Still EP proposes local learning rules that are very attractive for hardware implementations. EP has been applied to different tasks, datasets and neural network architectures but no large-scale hardware implementation has been reported in the literature yet.

One main reason, is that the devices that are envisioned for a hardware implementation are still experimental and noisy which prevent the training (see Section 2.1). Reducing the precision required for both the synapses and the neural activations is crucial for a demonstration of a training by EP of such unconventional hardware. For this purpose we have introduced Binary Neural Networks that are artificial neural networks which synapses and/or neural activation have been binarized. We described the gains in terms of memory and computing resources required for standard digital hardware but also described unconventional approaches to build hardware BNNs. We also described methods to train BNNs despite the fact that the synapses and the activations are binary. Furthermore, we introduced two methods that aim at avoiding neural saturation and allow the optimization of BNNs. In the next chapter (Chapter 4.3), we will combine both BNNs and EP to demonstrate that we can train BNNs with local learning rules in a supervised way.

Chapter 4

Training Dynamical Binary Neural Networks with Equilibrium Propagation

As seen in the previous sections, Equilibrium Propagation (EP) [14] is an algorithm intrinsically adapted to the training of physical networks, thanks to the local updates of weights given by the internal dynamics of the system. However, the construction of such a hardware requires to make the algorithm compatible with either existing neuromorphic CMOS technologies, which generally exploit digital communication between neurons and offer a limited amount of local memory or with the emerging nano-devices that are promising as they operate at low-power but are still experimental and exhibit a lot of device-to-device and intra-device variability. In this chapter, we demonstrate that EP can train dynamical networks with binary activations and weights. We first train systems with binary weights and full-precision activations, achieving an accuracy equivalent to that of full-precision models trained by standard EP on MNIST, and losing only 1.9% accuracy on CIFAR-10 with equal architecture. We then extend our method to the training of models with binary activations and weights on MNIST, achieving an accuracy within 1% of the full-precision reference for fully connected architectures and reaching the full-precision accuracy for convolutional architectures. Our extension of EP to binary networks opens new solutions for on-chip learning and provides a compact framework for training BNNs end-to-end with the same circuitry as for inference.

4.1 . Introduction

As we have seen, Equilibrium Propagation (EP) [14] is a learning framework that leverages the dynamics of energy-based physical systems fed by static inputs to compute weight updates with a learning rule local in space [157] which can also be made local in time [161], and in addition scales to CIFAR-10 [186]. Today, EP is developed on standard hardware (GPU) that 1) does not provide for the low power and the computing efficiency a dedicated hardware implementation might exhibit [103], [187] and 2) makes it complicated to scale EP to large scale datasets such as ImageNet due to the duration of simulations though recent results on ImageNet 32x32 [155] suggest it can be feasible. An EP-dedicated hardware would reduce the energy consumption of training by two orders of magnitude compared to GPUs and accelerate training by several orders of magnitude [169], while being competitive on large scale benchmarks in terms of accuracy since the gradients estimates prescribed by EP are equivalent to those given by BPTT [157].

The main asset of EP is the ability to perform on-chip learning, especially when the memory and the computational budgets dedicated to training and inference are constrained (e.g. embedded environments). EP is also naturally suited for training physical systems whose dynamics are unknown and hardly derivable [150], [163], [169]. EP therefore appears as a solution for on-chip training for embedded systems and dynamical hardware, two cases with which BP is not compatible without major adaptation. EP could be used to train neuromorphic hardware whether it is based on digital CMOS chips such as the famous neuromorphic platforms TrueNorth [74] or Loihi [75] or based on emerging nano-devices [187].

EP is however based on full-precision (64 bits floating point) weights and activations that do not match the current requirements of such hardware systems. Full-precision weights overload the memory capacity of chips when they are stored digitally [103], [188], and are prone to noise and hard to read when stored with emerging synaptic nano-devices [184], [185], [189], [190]. Moreover, analog activation functions are not directly compatible with widely-used digital communications between neurons [103]. This study has been initially derived in the goal to be applied to hardware built with emerging devices as they promise larger gains of energy than digital neuromorphic platforms. However, we will see that it could also have implications for the design of custom digital chips.

In this chapter, we address the issue of on-chip learning via EP by training dynamical systems having binary activations and weights. We first leverage the recent progress made in Binary Neural Networks (BNNs) optimization [160], to binarize the synapses in energy-based models trained by EP. The optimization of weights is performed using the inertia of the gradient. This lowers the memory required for training such systems compared to real-valued ("latent") weights optimization, traditionally used for training BNNs. We then binarize the activation functions, yielding an easy way to compute the local gradient while supporting a digital communication between neurons.

4.2 . EP Learning of Recurrent Binary Weights with Full Precision Neural Activations

In this section, we show that we can train dynamical systems with binary weights and full precision activations by EP with a performance on MNIST and CIFAR-10 close to the one achieved by their full-precision counterparts trained by EP [157], [186]. Our technique relies on the combined use of BOP described in Alg. 5 and of a proper weight normalization to avoid vanishing gradients. Therefore, we first describe how EP can be embedded into BOP (Alg. 6). Then, we propose two weight normalization schemes: one with a fixed scaling factor taken from [191], another one with a dynamical scaling factor directly learned by EP. We show that the use of the learnt weight normalization, which naturally fits into the EP framework, considerably improves model fitting and training speed on MNIST and CIFAR-10.

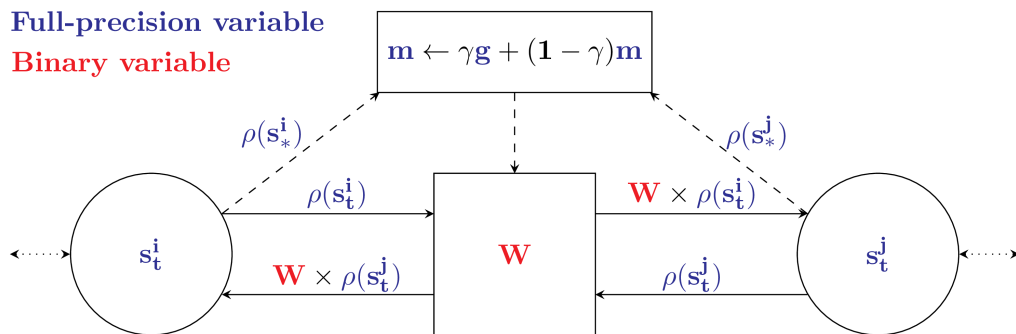


Figure 4.1: Building block of binarized EP: two neurons communicate bidirectionally through a binary synapse. The color code highlights the precision of each variable: the synaptic weight (bold red) is binary and the internal state of the neurons (s_t^k), as well as the momentum (m) averaging the gradient (bold blue), are full-precision. The activations ($\rho(s_t^k)$) and the equilibrium activations ($\rho(s_*^k)$) are full-precision variables (blue) in the section (Section 4.2). The gradient estimate (g) prescribed by EP (bold blue) is also a full-precision variable in this section (Section 4.2). Every neuron also has a bias which is full-precision and does not appear on the figure for clarity.

4.2.1 . Feeding EP weight updates into BOP

As said in Section 3.3.2, we cannot directly use stochastic gradient descent (SGD) to optimize binary weights. Using latent weights to support the SGD optimization step where the gradients are computed by backpropagation has long been the way to overcome the particularity of binary weights. However it requires to store with high precision the latent weights which has up to now prevent the training of BNN on the edge. Here we have chosen to use BOP instead of SGD+latent weights. BOP relies on the inertia of a momentum variable to dictate the state of the binary weights. This momentum variable is very similar in principle to the physical momentum, such as EP is very grounded in physics (as it relies on the property of physical systems to evolve toward the ground states with highest probabilities (see Section 2.1)). Thus merging both frameworks appeared to be a good match to train neural networks which have binary synapses.

Working principle. We now explain how we used BOP to optimize the binary synapses given the gradient computed with EP. At each training iteration, the first steps of our technique are the same as standard EP: the first phase and the second phase are performed as usual and the EP gradient estimate g is obtained from the steady states s_* and s_*^β for each synaptic weight. Thereafter, g is directly fed into the BOP algorithm (Alg. 5): for each synapse connecting neuron j to neuron i, the EP gradient estimate g_{ij} conveys inertia to the synaptic momentum m_{ij}, and the binary synaptic weight W_{ij} is flipped or not, depending on the value of m_{ij}. Finally, as usual in BNNs [60], the biases are full-precision and are updated with standard Stochastic Gradient Descent (SGD). We summarize all those steps in Alg. 6, where we have highlighted binarized variables in bold pink for clarity. With this procedure, we have a system in which the synapses are binarized at all time. In this section we use a full-precision activation function for the neurons, the hardsigmoid, and full-precision gradients. The binarization of activation functions and ternarization of gradients is addressed in Section 4.3.

Algorithm 6 EP learning of dynamical binary weights (with simplified notations). Binarized variables are in bold pink. When the neural activations are binarized (Section 4.3), the EP gradient estimate g is ternarized (in bold green), otherwise full precision (Section 4.2).

1: **Input**: x, $\hat{y}$, s, β, $\theta = \{\mathbf{W}, b\}$, η, m, γ, τ.
2: **Output**: $\theta = \{\mathbf{W}, b\}$, m.
3: *I. Inference & Error-backpropagation phases*
4: **for** $t = 1$ to T **do** ▷ 1. Free phase
5: $\quad s \leftarrow s - dt \times \frac{\partial E(x,s,\mathbf{W},b)}{\partial s}$
6: **end for**
7: $s_* \leftarrow s$
8: $\ell \leftarrow \ell(y^{*,0}, \hat{y})$ ▷ 2. Compute Loss function
9: **for** $t = T$ to K **do** ▷ 3. Nudge phase
10: $\quad s \leftarrow s - dt \times \frac{\partial E(x,s,\mathbf{W},b)}{\partial s} - \beta \times \frac{\partial \ell(\mathrm{y},y)}{\partial s}$
11: **end for**
12: $s_*^\beta \leftarrow s$
13: *II. Gradient computation & Optimization step*
14: $\mathbf{g}_{\theta=\{\mathbf{W},b\}} \leftarrow -\frac{1}{\beta}\left(\frac{\partial F}{\partial \theta}(x, s_*^\beta, \beta, \mathbf{W}, b) - \frac{\partial F}{\partial \theta}(x, s_*, 0, \mathbf{W}, b)\right)$
15: ▷ 4. Compute EP gradient from s_*^β and s_*
16: m,$\mathbf{W}$ = $\mathrm{BOP}(\mathbf{g}_\mathbf{W}, m, \mathbf{W}, \gamma, \tau)$ ▷ 5. Apply BOP (Alg. (5))
17: $b \leftarrow b + \eta \times \mathbf{g}_\mathbf{b}$ ▷ 6. Update biases with SGD:

Hyperparameter tuning. Similarly to [160], we monitor the number of weight flips per epoch and layer-wise in order to tune the hyperparameters of BOP, using the metric:

$$\pi_{\text{epoch}}^{layer} = \log\left(\frac{\text{Number of flipped weights}}{\text{Total number of weights}} + e^{-9}\right) \tag{4.1}$$

Heuristically, $\pi_{\text{epoch}}^{layer}$ reflects a trade-off between learning speed (high $\pi_{\text{epoch}}^{layer}$) and stability (low $\pi_{\text{epoch}}^{layer}$). We measure $\pi_{\text{epoch}}^{layer}$ in the regions of γ and τ where learning performs well, and use this value of $\pi_{\text{epoch}}^{layer}$ in return as a criterion to tune γ and τ on new models.

4.2.2 . Normalizing the Binary Weights with a fixed scaling factor

When binarizing synaptic weights to ± 1, neural activities may easily saturate to regions of flat activation function, resulting in vanishing gradients. It is especially true with the hardsigmoid activation function often used with EP. Batch-Normalization [192] (see Section 3.3.2) used by Courbariaux *et al*. [60] and Hubara *et al*. [193] helps with this issue by recentering and renormalizing activations by computing the batch statistics. Batch-Normalization has been introduced in recurrent neural networks such as LSTMs to process sequence tasks [194] but it does not translate directly to energy-based models. The normalization scheme should indeed itself derive from an energy function in order to be learnable, which restricts the choice of candidate normalizations. However, the goal in convergent dynamical systems processing static inputs is not to center neural activations at every time step, but rather at their steady state. Moreover, using batch-based weight normalization schemes is far from straightforward from a hardware prospective. For this purpose, we have first studied how we can normalize the binary weights with a fixed scaling factor.

Static XNOR-Net weight scaling factor. In the design of their XNOR-Net model, Rastegari *et al*. [191] introduced a scaling factor (see Section 3.3.1) to minimize the difference between the binary synapses and the corresponding set of full-precision "latent" weights at each layer. This scaling factor is updated at each training iteration as the 'latent" weights are being optimized. The scaling factor also depends on the size of two adjacent layers and on the magnitude of these latent weights. The scaling factor reads in our context:

$$\alpha_{n,n+1} = \frac{||w_{n,n+1}^{\text{init}}||_1}{\dim(w_{n,n+1}^{\text{init}})} \tag{4.2}$$

where n is the index of a layer, $w_{n,n+1}^{\text{init}}$ are the full-precision random weights used to initialize the binary weights. Using this scaling factor, we initialize each binary weight, layer by layer, as $W_{n,n+1} = \pm\alpha_{n,n+1}$. Contrarily to XNOR-Net where the scaling factor is updated at each forward pass, we first keep the scaling factor fixed to its initial value throughout training.

We describe in Alg. 7 how we initialize the binary weights and the corresponding scaling factors layer-wise:

Algorithm 7 Layer-wise initialization of the scaling factors

1: **Input**: Architecture: $\{\mathrm{N_{layers}}, \mathrm{N_{neurons_per_layer}}\}$.
2: **Output**: Scaled binary weights ($W^b \in \{\pm\alpha\}$)
3: **for** $l = 1$ to N_{layers} **do**
4: $\quad w^{init}_{l,l+1} = Kaiming(N_l, N_{l+1})$ ▷ Kaiming initialization [195]
5: $\quad \alpha = \frac{||w^{init}_{l,l+1}||_{l1}}{\dim(w^{init}_{l,l+1})}$
6: $\quad W^b_{l,l+1} = \alpha * \mathrm{Sign}(w^{init}_{l,l+1})$
7: **end for**

We found that the use of scaling factors α as initialized with Alg. 7 is crucial to ensure successful training. In fact, if the synapses are initialized too low or too large, we face the vanishing gradient issue as the activation saturate at both 0 or 1. In order to show this effect we trained a system with a fully connected architecture comprising 1 hidden layer of 4096 neurons, with binary weights and full-precision activations, for 50 epochs on MNIST. For this experiment we set the scaling factors between the input and the hidden layer equal to the scaling factor between the hidden and the output layer. We plot in Fig. 4.2 the test error obtained with Alg. 7 for different values of the fixed scaling factor. The blue arrow indicate the value corresponding to an initialization of α with Alg. 7 (we took the averaged value of both scaling factors initialized layer-wisely to obtain a point on the plot).

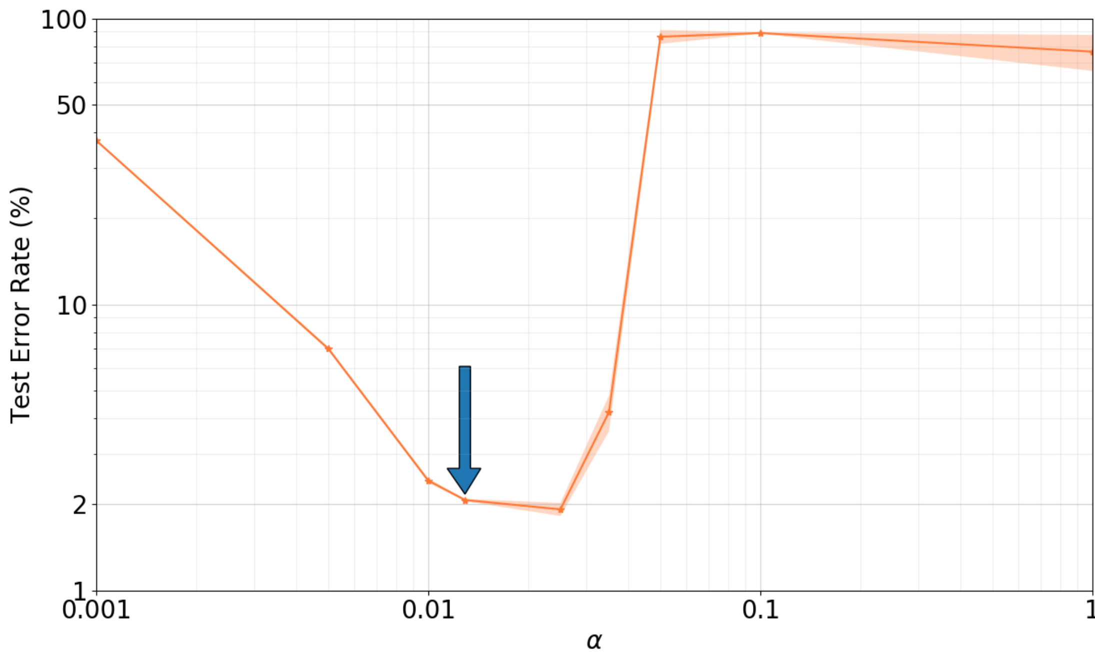


Figure 4.2: Mean test error $\pm$ standard deviation (computed with 3 trials each) of a 1 hidden layer fully connected neural network (784-4096-10) on MNIST as a function of the scaling factor α - The dots represent the test error of networks with a single fixed α - α initialized by the method described in 4.2.2 with Alg. 7 is indicated by the blue arrow.

Fig. 4.2 shows that there is a sweet spot for α between about 0.012 and 0.025 where the network performs at EP literature level. But even in this range it is not obvious to find the ideal value for α. Moreover, we found that fixing arbitrarily a single value for α in deep architectures (fully connected architecture with 2 hidden layers or convolutional architectures) fails at ensuring successful training. We finally chose to initialize the scaling factors layer-wise with the method of Alg. 7 as this method is architecture-agnostic and reduces the number of hyperparameters as we already have many to tune.

Table 4.1: Error of EP and BPTT on networks having binary synapses with a fixed or a dynamical scaling factor - Results are reported as the mean over 5 trials $\pm$ 1 standard deviation - Our results are highlighted with a light plum background - Benchmark performances are taken from [157], [186]

		EP - Binary Synapses				**EP**	**BPTT**
		Fixed α		Dynamical α		Benchmark	Binary Synapses
Dataset	Model	Test	Train	Test	Train	Test	Test
MNIST	(1fc)	2.07 (0.02)	0.77	**1.7 (0.04)**	**0**	2.00	2.14 (0.06)
MNIST	(2fc)	2.48 (0.08)	0.29	**2.28 (0.13)**	**0**	1.95	2.38 (0.07)
MNIST	(conv)	$0.85 (0.11)$	0.46	$\mathbf{0.88 (0.06)}$	$\mathbf{0.05}$	1.05	0.97 (0.03)
CIFAR-10	(conv)	$16.8 (0.3)$	6.9	$\mathbf{15.66 (0.28)}$	$\mathbf{5.54}$	13.78	14.45 (0.12)

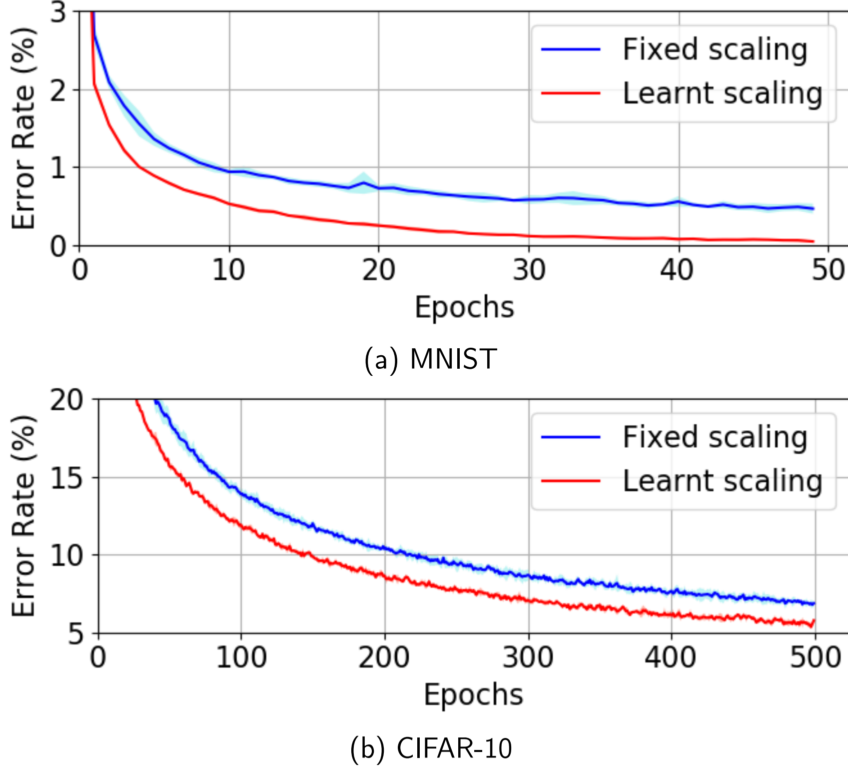
Results. We investigate fully connected architectures (with one and two hidden layers) on MNIST and convolutional architectures (with two and four convolutional layers) on the MNIST and CIFAR-10 datasets. We employ prototypical models to speed up training as in [157]. Our results (Table 4.1 - "EP - Binary Synapses") are benchmarked against those of full precision models (Table 4.1 - "EP - Benchmark") and those obtained by BPTT+BOP (Table 4.1 - "BPTT - Binary Synapses"). Note that for a given architecture, the number of neurons we used per layer may not be the same as in reference architectures – see Appendix C.2 for details. Also, the code to run the simulations has been made available online on Github[1] with all the command lines to easily reproduce the simulations.

Overall, Table 4.1 shows that the normalization of weights with a fixed scaling factor allows EP with binary synapses to perform comparably to full-precision models trained across different fully connected and convolutional architectures, on MNIST and CIFAR-10. The fully connected architecture which has one hidden layer trained on MNIST shows no statistically significant loss of performance compared to full-precision counterpart trained by EP. This architecture with binary synapses reaches the same accuracy if trained by (EP+BOP) or by (BPTT+BOP). The fully connected architecture having two hidden layers trained on MNIST with fixed scaling factors shows 0.5% performance degradation compared to full-precision models trained by EP. For the convolutional architecture trained on MNIST, we can even observe a slightly better training and testing (-0.2%) errors on model with binary synapses compared to full precision models trained by EP. We explain this improvement by the cumulative use of the randomization of β and of the regularization effect induced by the binarized architecture itself [60]. Furthemore, the training framework (EP+BOP) achieves a similar accuracy as the framework (BPTT+BOP). Finally, the performance of our convolutional model trained on CIFAR-10 is $\sim 3\%$ less than the one of Laborieux *et al*. [186], using the same architecture. Also, the network trained by (EP+BOP) shows only 2.5% degradation of the accuracy compared to the same network trained by (BPTT+BOP).

[1] Link to the Github repository for this project

In the next subsection (Section 4.2.3), we address the difficulty to select the best value of α in order to get the best testing accuracy by directly learning the scaling factor with the help of EP.

4.2.3 . Normalizing the Binary Weights with a learnt scaling factor



(a) MNIST

(b) CIFAR-10

Figure 4.3: Average training error as a function of the number of epochs for a convolutional architecture with binary synapses trained on MNIST with a static (blue curve) or a dynamical (red curve) scaling factor. Curves averaged over 5 trials ± 1 standard deviation.

Dynamical weight scaling factor learned by EP. Using fixed scaling factors gives high, yet sub-optimal accuracies (see Fig. 4.2 in the Appendix). Bulat & Tzimiropoulos [196] show that the scaling factor can be learnt by backpropagation to extend XNOR-Nets. Here we derive a learning rule for the scaling factor with the help of the theorem of Scellier & Bengio [14] to ensure that it provides a gradient estimate of the loss $\mathcal{L}_*$ defined in Eq. (3.21). The reasoning to derive this learning rule is the following. We split the binary weights in two parts: $W_{n,n+1} \leftarrow \alpha_{n,n+1} \times w^{bin}_{n,n+1}$ where $w^{bin}_{n,n+1}$ are the binary weights scaled to ± 1 and the scaling factors $\alpha_{n,n+1}$ are initialized as when they are fixed. The resulting dynamics still derives from an energy function, and one can derive a learning rule for $\alpha_{n,n+1}$ which reads as:

$$\Delta\alpha_{n,n+1}(\beta) = -\frac{1}{\beta}\left(\frac{\partial E}{\partial \alpha_{n,n+1}}(s^\beta_*) - \frac{\partial E}{\partial \alpha_{n,n+1}}(s_*)\right), \tag{4.3}$$

so that $\alpha_{n,n+1}$ is learned like any other network parameter, and $\lim_{\beta\to 0}\Delta\alpha_{n,n+1}(\beta) = -\frac{\partial \mathcal{L}_*}{\partial \alpha_{n,n+1}}$. In Section C the learning rules for fully connected and convolutional architectures are derived in all the settings of EP.

Results. Fig. 4.3 illustrates on a convolutional architecture the gain in performance obtained by learning the scaling factor. Globally, this technique systematically results in faster learning and better model fitting across all the models, and almost always in better generalization as observed in Table 4.1. Learning the scaling factor by EP is thus a powerful alternative to Batch-normalization in convergent dynamical systems as highlighted by Fig. 4.3.

In the next tables 4.2, 4.3 and 4.4, we report the training and test errors obtained with fixed and dynamical scaling factors on MNIST and CIFAR-10 with different architectures, at mid-training and at the end of the training.

Table 4.2: Mean Train and Test errors on MNIST (over 5 trials each) computed after 25 and 50 epochs for two shallow networks with one and two hidden layers with binary synapses trained with EqProp - We denote in the *Learn* α column if the scaling factor is learnt or not

		25 Epochs	50 Epochs
Architecture	Learn α	Test (Train)	Test (Train)
784-4096-10	✗	2.14 (0.92)	2.07 (0.77)
784-4096-10	✓	1.66 (0.03)	**1.7 (0)**
784-512-10	✗	- (-)	**4 (3.5)**
784-512-10	✓	2.45 (1.24)	**2.2 (0.7)**
784-4096(2)-10	✗	2.47 (0.4)	2.48 (0.15)
784-4096(2)-10	✓	2.27 (0.02)	**2.28 (0)**

Learning the scaling factor accelerates the training. In tables 4.2, 4.3 and 4.4,, we show that the training times are accelerated when the scaling factor is learnt instead of being fixed after initialization. In particular for MNIST, the training is accelerated by a factor over two compared to the fixed scaling, for both the fully connected and convolutional architectures.

For CIFAR-10 the acceleration is not as large as for MNIST but we struggled to fine-tune the learning rate for the scaling factors and thus better combinations could result in a larger acceleration.

Table 4.3: Mean Train and Test errors on MNIST (over 5 trials each) with a convolution network with binary synapses trained with EqProp - We denote in the *Learn* α column if the scaling factor is learnt or not

		25 Epochs	50 Epochs
Architecture	Learn α	Test (Train)	Test (Train)
1-32-64-(fc)	✗	0.92 (0.63)	0.84 (0.46)
1-32-64-(fc)	✓	0.79 (0.16)	**0.88 (0.047)**

Networks that learn the scaling factors better fit the training set. Also in tables 4.2, 4.3 and 4.4, we see that every trainings done with dynamical scaling factors always better fit the training set than trainings done with fixed scaling factors. Training errors on MNIST are improved by 0.8% and 0.15% for fully connected layers architectures having 1 and 2 hidden layers. The convolutional architecture trained on MNIST also gains 0.45% in terms of training error. Whereas

the fully connected architectures sees the test error also improved alongside the training error, the convolutional architecture sees a slight degradation of the test error due to over-fitting.

The convolutional architecture trained on CIFAR-10 gains 1.1% training error.

Table 4.4: Mean Train and Test errors on CIFAR-10 (over 5 trials each) with a convolution network with binary synapses trained with EqProp - We denote in the *Learn* α column if the scaling factor is learnt or not

		100 Epochs	500 Epochs
Architecture	Learn α	Test (Train)	Test (Train)
3-68-128-256-256-(fc)	✗	18.4 (13.4)	16.8 (6.9)
3-68-128-256-256-(fc)	✓	17.6 (11.86)	**15.54 (5.54)**

Learning the scaling factor can reduce the memory requirements of the network Finally, learning the scaling allows to train a fully connected architecture with 1 hidden layer of only 512 hidden neurons (the standard architecture being trained by EP on MNIST in the literature) with very little loss of accuracy compared to the architecture with 4096 hidden neurons. This little architecture with learnt scaling factor looses only +0.2% testing error and +0.6% training error (see Table 4.2 and Fig. 4.1) whereas with a fixed scaling factor we have +2% testing error and +3% training error (see Table 4.2 and Fig. C.1). Despite the fact that we did not make the same simulations with deeper neural networks or with the convolutional architectures, we expect the effects of the learnt scaling factor to be same, thus drastically reducing the size of the model for hardware implementations.

4.3 . EP Learning of Recurrent Binary Weights with Binary Neural Activations

The techniques presented in the previous section use full-precision neural activations. However, it is highly preferable to rely on binary activation values in hardware. Binary read and write errors can indeed be accommodated without too much circuit overhead in neuromorphic systems [184] and binary values are easier to pass between spatially distant hardware neurons [103]. In this section, we show that we can train dynamical systems with binary weights and binary activations by EP, resulting in a performance on MNIST again approaching full-precision models on fully connected and convolutional architectures. Our implementation relies on two main components: the choice of a proper activation function to binarize neural pre-activations, and output layer augmentation. Combining these two techniques, we can design dynamical systems which are sensitive to error signals despite threshold effects and can compute ternary gradients in return. The corresponding pseudo-algorithm is the same as Alg. 6, except that the gradient estimate g is now ternarized.

4.3.1 . Convergent neural networks with binary activations.

Ternarizing EP gradients. We can note from Eq. (3.27) and Eq. (3.28) that the precision of the gradient g estimate provided by EP is typically determined by the choice of the activation function ρ. For instance, if ρ outputs binary values $\{0, 1\}$, we immediately see from Eq. (3.27) that, for the parameters $\theta = \{W, b\}$, the gradients has values:

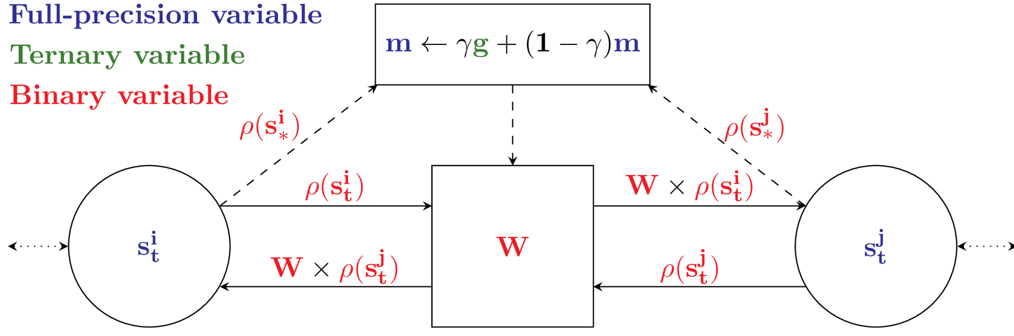


Figure 4.4: Building blocks of fully-binarized EP: the synaptic weight (bold red) is binary and the internal state of the neurons (s_t^k), as well as the momentum (m) averaging the gradient (bold blue), are full-precision. The activations ($\rho(s_t^k)$) and the equilibrium activations ($\rho(s_*^k)$) are now binary (red) in this section (Section 4.3). The gradient estimate (g) prescribed by EP (bold blue) is here ternary (bold green) in this section (Section 4.3).

$$\mathbf{g}_\theta \in \left\{-\frac{2}{\beta}, 0, \frac{2}{\beta}\right\}. \tag{4.4}$$

In practice $\beta = 2$ works well, resulting in $\times 40$ gradient compression compared to 64-floating point resolution.

However, binarizing neural activations comes at several costs for the dynamics of the neurons. The energy function of the system subsequently outputs a semi-discrete variable which affects the dynamics of each neuron non trivially: if the updates of neuron activations are simultaneous, the dynamics of the system may not converge [197]. In particular, this precludes the use of prototypical models [157], that can be employed to speed up training as we did in the previous section. Therefore, we must use standard energy-based models as in [14] so that binary activations are updated only when the full-precision pre-activations reach the threshold of the activation function, thus non simultaneously. We next detail some empirical properties the binary activation of the neurons should have in order to define convergent dynamics.

Binarizing neural activations into $\{0, 1\}$. While we binarize weights to opposite signs, we found that using the sign activation function to binarize neural activations into $\{-1, 1\}$, as usually done with BNNs to implement MAC operations with XNOR gates, entails non-convergent dynamics. This confirms previous findings on EP which emphasized the importance of bounding the neural activations between 0 and 1 to help dynamics convergence using the hardsigmoid activation function $\rho(s) = \max(0, \min(s, 1))$ [14]. Therefore, our proposal here is to use the Heavyside step function shifted by 0.5:

$$\rho(s) = \mathrm{H}\left(s - \frac{1}{2}\right) \tag{4.5}$$

where $H(x) = 1$ if $x \geq 0$, 0 otherwise. However, the energy-based dynamics of our models requires to gate $\frac{\partial E}{\partial \rho}$ by the derivative of ρ as Eq. 3.16 rewrites as:

$$\frac{ds}{dt} = -\frac{\partial E}{\partial s}(x, s, \theta) - \frac{\partial \rho(s)}{\partial s}\frac{\partial E}{\partial \rho(s)}(x, \rho(s), \theta). \tag{4.6}$$

Noting that Eq. 4.5 is obtained by asymptotically sharpening the narrowed hardsigmoid around $\frac{1}{2}$ within $[\frac{1}{2} - \sigma, \frac{1}{2} + \sigma]$ denoted $\hat{\rho}$, namely:

$$\rho(s) = \lim_{\sigma \to 0} \hat{\rho}(s, \sigma) = \mathrm{H}\left(s - \frac{1}{2}\right) \tag{4.7}$$

we propose to substitute the derivative of ρ as:

$$\frac{\partial \hat{\rho}(s, \sigma)}{\partial s} \approx \frac{\partial \boldsymbol{\rho}(s)}{\partial s} = \begin{cases} \frac{1}{2\sigma} & \text{if } \left|s - \frac{1}{2}\right| \leq \sigma \\ 0 & \text{else} \end{cases} \tag{4.8}$$

where σ is a parameter discussed in Appendix C.2. Therefore, denoting $\hat{\rho}' = \partial_s \hat{\rho}$ for simplicity, the free dynamics of s can be approximated as:

$$\frac{ds}{dt} \approx -\frac{\partial E}{\partial s} - \hat{\rho}'(s) \frac{\partial E}{\partial \rho} \tag{4.9}$$

4.3.2 . Augmenting the Error Signal to Nudge Neurons with Binary Activations

Binarization of activations can block the proper propagation of errors. As the system sits at rest at the end of the first phase of EP, upon nudging the output layer by the prediction error, the motion of the system during the second phase of EP encodes error signals [157], [198]. Therefore during the second phase, a given neuron i needs to have its activation function change from $\rho(s_{*,i})$ to a distinct $\rho(s^{\beta}_{*,i})$ to compute the error gradient locally and transmit it backward to upstream layers. However, when using a discontinuous activation function like defined in Eq. (4.7), we may have $\rho(s_{*,i}) = \rho(s^{\beta}_{*,i})$ if the pre-activation s_i of the neuron moves less than the value of the activation threshold of ρ, thus zeroing the error signal, or equivalently vanishing gradients. Consequently, we need to ensure that for a sufficient number of neurons i:

$$\Delta s_i = |s^{\beta}_{*,i} - s_{*,i}| > \frac{1}{2} \tag{4.10}$$

In order to satisfy Eq. (4.10) for a sufficient number of neurons, we propose to increase the error signal by augmenting the output layer so that each prediction neuron is replaced by N_{perclass} neurons per class, inflating the output layer from N_{classes} to $N_{\text{classes}} \times N_{\text{perclass}}$. We choose N_{perclass} in such a way that the number of output neurons matches approximately the number of neurons in the penultimate hidden layer: $N_{\text{perclasses}} \approx \frac{N_{\text{penultimate}}}{N_{\text{classes}}}$. In this way, the output layer delivers a large and redundant initial error signal that can push neurons beyond the activation threshold of ρ and propagate across the whole architecture. Our solution is reminiscent of the the use of auxiliary output neurons already used for spiking neurons trained with STDP [15] were the enlarged output layer is used for the diversity of the neurons it offers so different neurons can specify to a specific input and that also inspired [199]. Here this solution is used with a very different motivation.

Why increasing only the size of the output layer? A layered architecture trained with EP makes the system sensitive to the error signal if and only if neurons between the output layer - where the error signal is applied - and a neuron of interest, are sensitive to the target. The first hidden layer is in this sense a bottleneck for the error signal. It often receives more forward signal from the other hidden layers than backward signal from the output layer which only has 10 neurons for MNIST and CIFAR-10. During the nudging phase of EP, only one neuron in the output layer

is nudged to be 1, the others being nudged to 0. And this little change in the output layer is not sufficient for the first hidden layer to reach the criteria of good error signal Eq. 4.10. Therefore, the binary activations of the neurons in the first hidden layer do not change and the error signal is blocked.

Once the first hidden layer changes its binary activations, the error signal can flow through the network. In fact, it has more impact on the others hidden layers because it is often larger than the output layer and matches approximately the size of the others layers thus it is more likely to impact the binary activation of the next hidden layer. Augmenting the error signal is crucial to train systems with binary synapses and activations with EP.

4.3.3 . Results

We investigate here fully connected (1 and 2 hidden layers) and convolutional architectures on MNIST. The first layer receives full-precision inputs from the input layer and binary inputs from the next layer. For a given architecture, the number of neurons used per layer is different for both situations: for the fully connected architectures we use 8192 neurons per hidden layer and the two convolutional layers of the convolutional architecture have respectively 256 and 512 channels - see Appendix C.2 for more details. We use a randomized sign for β as prescribed by Laborieux *et al*. [186] to improve the gradient estimate given by EP for all simulations except for the fully connected architecture with two hidden layers where we only use $\beta > 0$.

Fig 4.6 shows for the convolutional architecture a trend observed for all models: when using 10 neurons in the output layer, training fails (blue curve) while it succeeds upon augmenting the output layer. It is here augmented by a factor 70 (red curve) which is required for the number of neurons in the output layer to match the number of input neurons that the last convolutional layer receives from the penultimate convolutional layer: we multiply the number of channels in the penultimate convolutional layer (256) by the kernel size (5×5) and divide the result by the max pooling kernel size (3×3) which gives $\sim$ 700 output neurons).

Table 4.5: Error achieved by EP with binary synapses & activations, and fixed scaling factors α - Results are reported as the mean over 5 trials $\pm$ 1 standard deviation.

		Fully binarized EP	
Dataset	Model	Test	Train
MNIST	(1fc)	2.83 (0.06)	0.2
MNIST	(2fc)	$3.03(0.03)$	$0.84(0.17)$
MNIST	conv	$1.14(0.08)$	$0.67(0.04)$

Performance. The results obtained on MNIST with fixed scaling factors are summarized in Table 4.5. On the fully connected architectures, the accuracy approaches those obtained with binary synapses and full-precision activations, with a slight degradation of 0.8% for one hidden layer and 0.6% for two hidden layers (see Table 4.1). The degradation is slightly enhanced when we compare with the full-precision counterpart trained by EP where the performance is degraded by 0.8% for one hidden layer but 1% for two hidden layers. We account the degraded performance of the architecture which has two hidden layers by the fact that we use $\beta > 0$ which makes the estimation of the gradient less accurate than when estimated with the sign of β random. Using a random sign for β is a good practice [14], [157], [165] to ensure the gradient estimate is not biased. [14], [157] use a random sign for β while [165] prescribe to use both sign for β and to do two nudging

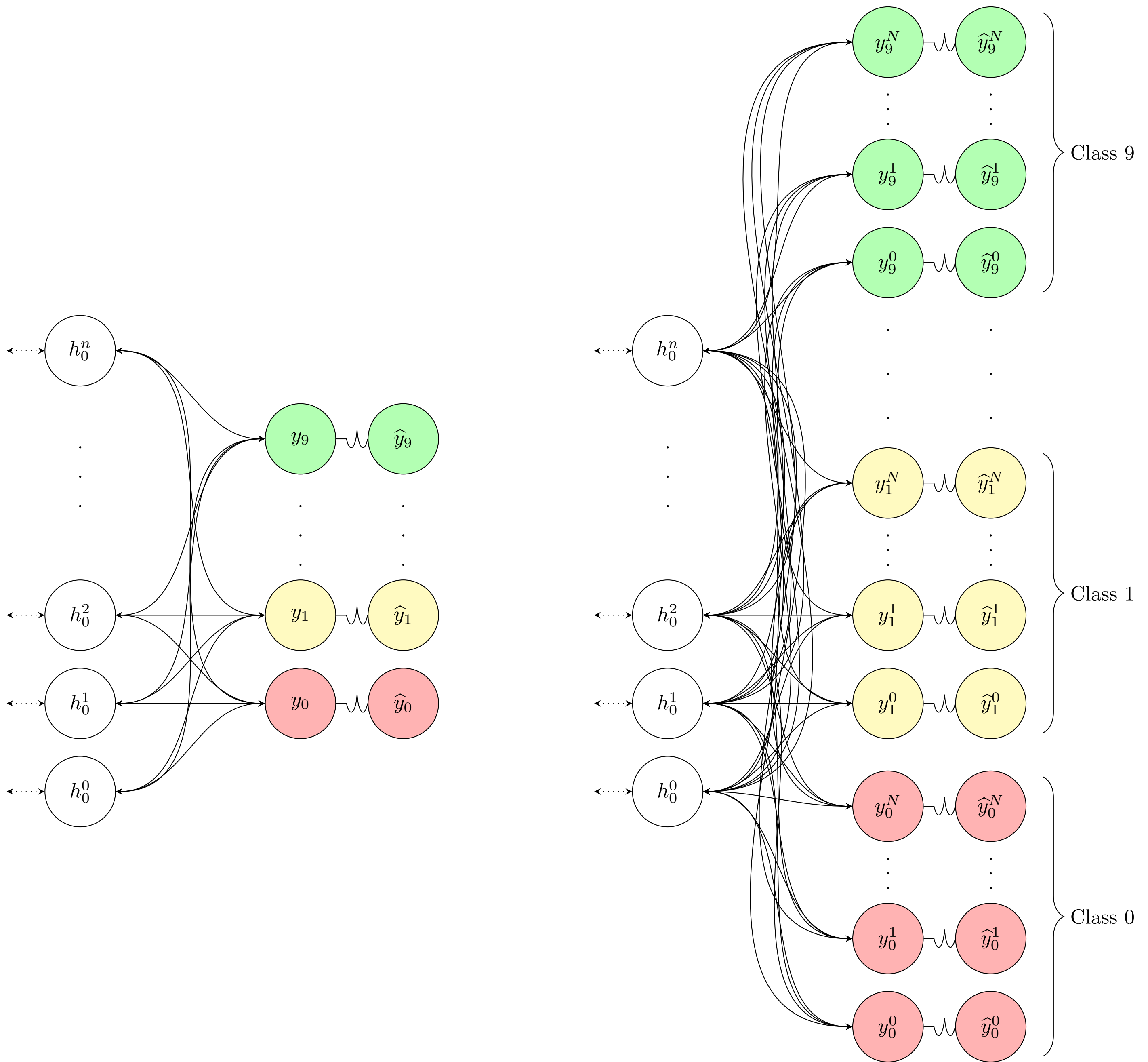


Figure 4.5: Left: Schematic of the classic output layer with one output neuron per class - compared to Right: the enlarged output layer where we have N_{perclass} output neurons per class - Hidden units are denoted by h, output units by y_x and the target units by $\widehat{y}_x$ - We represent the nudging of the output neurons by the corresponding target units with the small springs on the schematic - For simplicity we drew this schematic for datasets having 10 output classes but it can be applied to any dataset - Dashed arrows on the left hand of both networks indicate the bidirectional connections with the rest of the network

phases to cancel the 0 order bias of the gradient estimation. We used $\beta > 0$ because we found that reaching the second equilibrium point with $\beta < 0$ is possible but very long to get in practice with

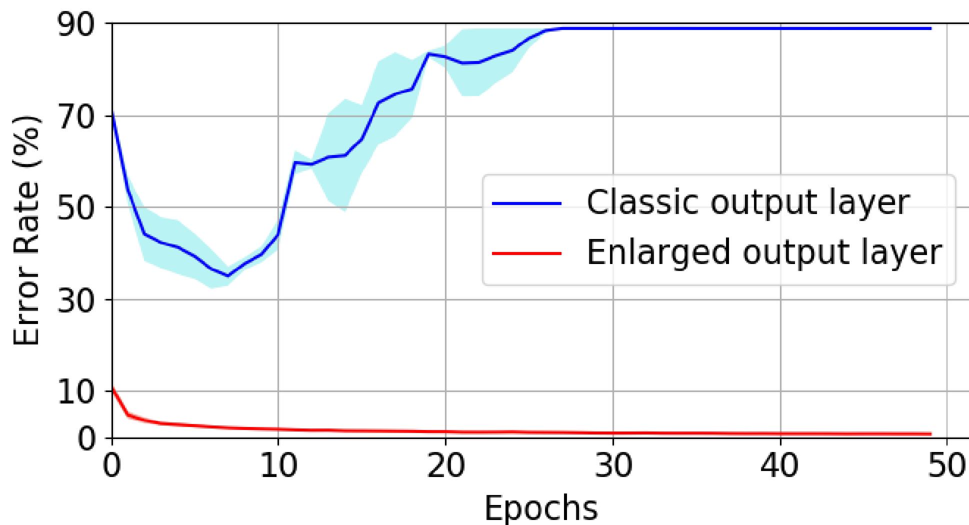


Figure 4.6: Average training error as a function of the number of epochs for a convolutional architecture with binary synapses and binary activations trained on MNIST with a classic output layer (10 output neurons - blue curve) or an enlarged output layer (700 output neurons - red curve). Blue curves are averaged over 2 trials ± 1 standard deviation - Red curves are averaged over 5 trials ± 1 standard deviation.

the standard nudge due to the large variation of the dynamics caused by a lot of output activations changing after being nudged. For the convolutional architecture trained on MNIST, we also report a performance only 0.2% below the system which has binary synapses and full-precision activations but within the error bars of the one achieved by full-precision models as reported by [157]. We think that optimizing the nudging strategy - or adding skip connections such as in ResNet that avoid vanishing gradients [34] - could improve the error obtained with two or more hidden layers and will be key in the future for scaling to CIFAR-10.

Efficient inference with an enlarged output layer Usually an output layer has as many neurons as the number of classes in the dataset and the prediction is the *argmax* of the output layer.

But when we train systems having binary activations the output layer is augmented (Section 4.3.2) and it is not straightforward to make a prediction taking the *argmax* of the output layer. Doing inference with this enlarged output layer can be a major computational overhead.

We describe next two methods we used to make a prediction with the enlarged output layer. We used both methods in our simulations and show they give similar accuracy in the end.

Making predictions by averaging each sub-class. This first method allow us to retrieve a situation similar to the classic output layer having one neuron per class. In fact we first average the internal state - or pre-activation - of each neuron belonging to a class which gives 10 averaged values and the prediction is taken as the *argmax* of these averaged values. This method is reminiscent to that developed in [200] where the method was used to combine different classifiers.

Making predictions with one neuron per sub-class. The first method we describe above to make the prediction could reveal to be computationally and time expensive and costly to realize

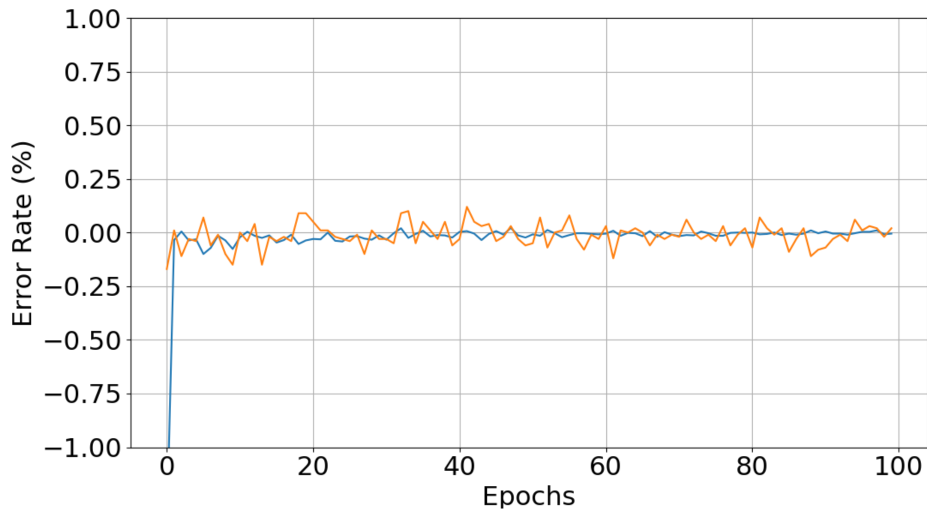


Figure 4.7: Difference of the train (blue) and test (orange) errors computed with the averaging method and the method where only one neuron per subclass is used for getting the infered class. We used a neural network with one hidden layer of 8192 neurons and 100 output neurons

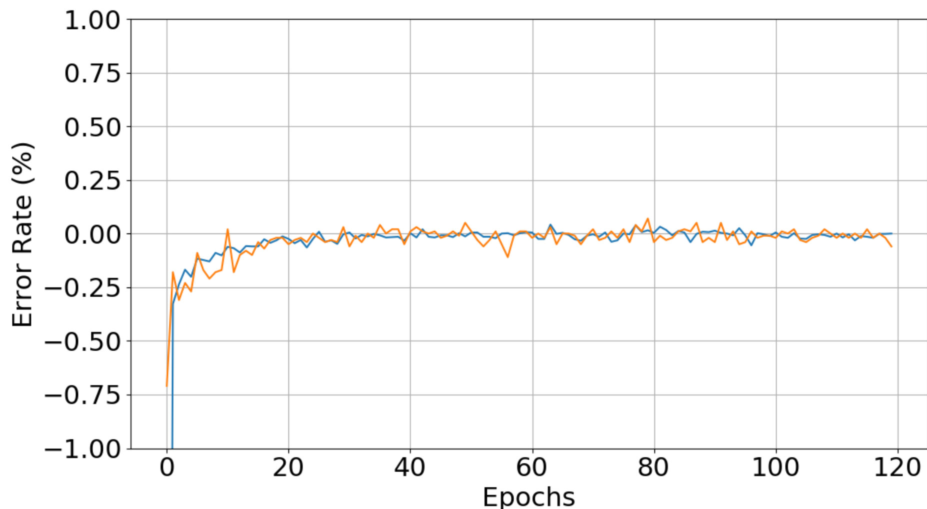


Figure 4.8: Difference of the train (blue) and test (orange) errors computed with the averaging method and the method with only one neuron per subclass is used for getting the infered class. We used a neural network with two hidden layers of 8192 neurons and 8000 output neurons

on digital hardware. A second, more hardware-friendly, method is to look at the state of only one output neuron per class and take the argmax of these "single-neurons".

Comparison of the two methods. We wanted to see if the second method, which constitutes a great simplification of the prediction process, would perform as well as the first method. For this purpose, we plot in Fig. 4.7 and Fig. 4.8 the difference of the training and the testing errors of

two fully connected architectures with binary synapses and activations and 1 and 2 hidden layers trained on MNIST computed with the two methods described above. We see that for the network with 1 hidden layer, the difference between the two methods does not exceed 0.1% for both training and testing errors. For the network with 2 hidden layers, despite the fact that the difference starts at a high level with more than 0.7% of difference for the testing error and more than 1% for the training error, in the end of the training process the differences have decreased to almost 0%. Both figures show the effectiveness of the second method at making the predictions at a lower cost both computationally and in time than the first method.

4.4 . Related works

Ji & Gross [164] have studied the effect of weight and gradient quantization of an energy-based model trained by EP, showing that at least 12 or 14 bits are required to achieve less than 10% test error on MNIST. Here we show that the weights (at all time) and the neural activations (at read time) can be compressed down to 1 bit only, yielding ternary gradients (at read time) and binary communication between neurons in the system. We discuss how the full-precision pre-activations and accumulated weight momentum can be handled in a neuromorphic chip. Finally, our work is the first to demonstrate energy-based model compression with EP on CIFAR-10.

4.5 . Impact of Binary Equilibrium Propagation on hardware implementations

The work around binary EP has been mainly motivated by a hardware perspective since binary elements (synapses or activations) can foster simpler hardware implementations of EP, both for neuromorphic hardware platform with standard digital electronics (TrueNorth or Loihi) or for unconventional hardware, digital or analog, made of emerging nano-devices. Despite the fact that the former technology is mature compared to the emerging analog nano-devices envisioned for low-power implementations, our work about reducing precision required for both synapses and neuron can accelerate the emergence large-scale implementations made of such nano-devices trained with EP.

For this purpose, we identified a few perspectives building on the results we got.

The first perspective is about the implementation of the binary synapses along with the optimization circuit that allows to program them during the training process. In the following first subsection we discuss the challenge of using non-volatile memories to implement the binary weights on-chip. Using such devices to encode the binary states drastically reduce the memory requirement of the implementation but more than that, it allows new computing architectures such as a cross-bar array that bypasses the Von Neumann bottleneck and promises large gains in energy consumption. The design of a custom circuit that performs the optimization step of BOP [176] is also a challenge, but the physical roots of the optimizer make it, combined with emerging devices, possible to implement, in principle.

The second perspective concerns the neurons having a binary activation function. In the second subsection we examine how the dynamics inherited from both the binary synapses and neurons can be rewritten in terms of elementary functions implemented at a low-level on CMOS hardware. This makes it possible to design digital neurons that, independently of how the hardware synapses are implemented, have simple dynamics. Finally we imagine to use stochastic hardware binary neurons

that can switch from a state to the other with more or less probability, such as stochastic magnetic tunnel junctions. This make the hardware implementation even compacter and energy-saving. The stochasticity of the neurons could be a way to add regularization to the dynamics [201], to allow a better propagation of the error signal but above all is to be put into perspective with the results we will show in the next section (Section 5) where initial stochastic binary neurons are rendered deterministic during learning with EP.

4.5.1 . Implementing in hardware the binary synapses and the optimization stage

Depending on the kind of hardware implementation, standard digital or emerging nano-devices based neuromorphic platform, binary synaptic weights can be encoded by different means. For the first, we can simply encode a binary weight as a bit whose two possible states encode the binary weight. Then, as we have already seen in Section 3.3.1, one can leverage the elementary XNOR function to compute the matrix product with vectors of binary activations or if the activations are spikes such as on TrueNorth and Loihi, the bit can drive a voltage inverter that modulates the sign of the spike.

Binary NVMs as hardware binary synapses? We focus on the second possibility for implementing binary synapses in hardware, as emerging nano-devices such as the Non-Volatile Memories (NVMs) could drastically reduce the energy required to train and use hardware neural networks. Refs. [174], [182], [184], [202] have already highlighted the interest of using NVMs to encode the binary states of binary synapses in hardware. Those NVMs can be either RRAM (Resistive memory such as a memristors (see Section 2.1)) or MRAM (Magnetic memory such as a magnetic tunnel junction).

Contrarily to standard digital bit storage, those approaches allow us to possibly arrange the devices in cross-bar array architectures. Thus, as we have already described in Sections 2.1 and 3.3.3, the Kirchhoff laws can be leveraged to perform both the synaptic stage of a neural network and the summation of the input of each neuron.

Memory gain at run time. Conceptually, having binary weights results in a drastic reduction of the memory requirements for training a neural network, especially with EP. But this still has to be put into perspective with the need of larger architectures. As often observed in binarized architectures [60], [193], we achieve accuracy similar to the one of full-precision models at the price of having 8 times more hidden neurons in fully connected architectures (see Fig. C.1 of Appendix C.2.1 for a more detailed analysis). In convolutional architectures with binary synapses only, we have used at most the same number of output feature maps than their full precision counterparts for computational efficiency. After training, our models use 2 and 7.5 less memory for the synapses for fully connected architectures on MNIST (for two and one hidden layers respectively), 9 and 54 less for the convolutional architectures used on MNIST and CIFAR-10 respectively. Also, our preliminary results (Section 4.2.3) indicate that learning the scaling factors drastically reduce the size required for a model to reach sota accuracy which can also reduce the memory requirement of the binary neural networks trained with EP.

Another important point to investigate beyond the implementation of binary synapses, is whether we can perform the optimization step of BOP on hardware without memory and energy overhead.

Realize BOP in hardware: leveraging capacitors dynamics to store the momentum. The memory requirements for training should be subject to a more careful treatment. Inertia-based optimization (BOP) requires a single full-precision variable: the momentum, compared to the latent-weight counterpart often trained by elaborate optimization techniques such as (SGD + Momentum) which uses at least 2 full-precision variables: the latent weight and the momentum. Inertia-based optimization thus reduces by at least a factor 2 the required memory for training. Still we have to store this full-precision variable which could be a large memory overhead.

However, the particular nature of BOP which is based on a momentum makes it particularly attractive to be implemented with non-standard memories. First, let's recall the discrete dynamics of the momentum that rules the BOP optimization algorithm:

$$m(t+1) = (1-\gamma)\cdot m(t) + \gamma \cdot g(t) \tag{4.11}$$

where $t+1$ refer to the update of the momentum after the gradient $g(t)$ is computed either for a single input data or a batch of multiple inputs.

In fact, this discrete time update of the momentum can be rewritten into a continuous time update rule which reads:

$$\frac{dm}{dt} + \gamma \cdot m(t) = \gamma \cdot g(t) \tag{4.12}$$

which naturally appears as the differential equation describing the evolution of the voltage of a capacitor. γ would be the equivalent to the time-constant $\frac{1}{RC}$ of a RC circuit. Capacitors are CMOS-compatible, and highly linear which makes them well suited for storing full-precision variables as already proposed in [189] where they use capacitors to store the weights updates during a short period of time. They can thus be used to store the inertia, thereby lowering the memory requirement to one capacitor per binary weight and more globally lowering the memory required for training.

In Fig. 4.9 we propose a simple way to arrange the capacitors that would retain the momentum for every binary weight of the hardware neural network in a cross-point memory architecture. For this implementation, one capacitor per binary weight would be required. Capacitors would be fed sequentially with their corresponding gradient signal after each gradient computation step. The flipping signal to flip the binary weight W_{ij} is obtained by simply comparing the voltage of the capacitor that encodes the momentum m_{ij} with a tunable DC threshold voltage V_{th}.

Still, as we have described in Section 4.2.1, we need to implement the possibility to fine-tune the time constant $\gamma = \frac{1}{RC}$ of the low-pass filter. We propose to pair each capacitor with an adjustable resistor that could be a memristor for instance to implement this tunability. This will allow to design a task and architecture-agnostic optimization chip that is completely disconnected from the kind of neural network it optimizes and from the algorithm that computes the gradient (EP or backpropagation).

We admit this architecture is a step back from the goal of having a local learning rule with compact circuitry for the update of the parameters as the optimizer requires an entire hardware cross-bar memory to store all the momentum. However this way to store the momentum requires only one pair of devices per binary synapses compared to 8, 16 or 32 bits depending on the precision required if the momentum is stored in standard memories as a full-precision variable. Furthermore, this architecture reduces the number of signals to apply to select a single device on the optimization chip. Indeed, we address the individual capacitors with a specific address that is the couple $(\#row, \#column)$, which corresponds to the address of the corresponding synapse in the synaptic cross-bar array. Thus, as the update phase is essential for training the neural network, inevitably the

NVMs are individually and sequentially addressed and we can take advantage of this sequential addressing to address simultaneously the corresponding capacitor which reduces the number of signal to generate for the optimization stage.

Finally, this design to implement BOP in hardware is quite general-purpose as it does not depend on the gradient-computation algorithm (EP or backpropagation) nor on the kind of neural network it is optimizing. So we can imagine that this potential optimization chip could be added on top to existing custom implementations of BNNs (with FPGAs or ASICS) for instance to enable training on-chip (assuming one can easily flip the binary weights on the implementation).

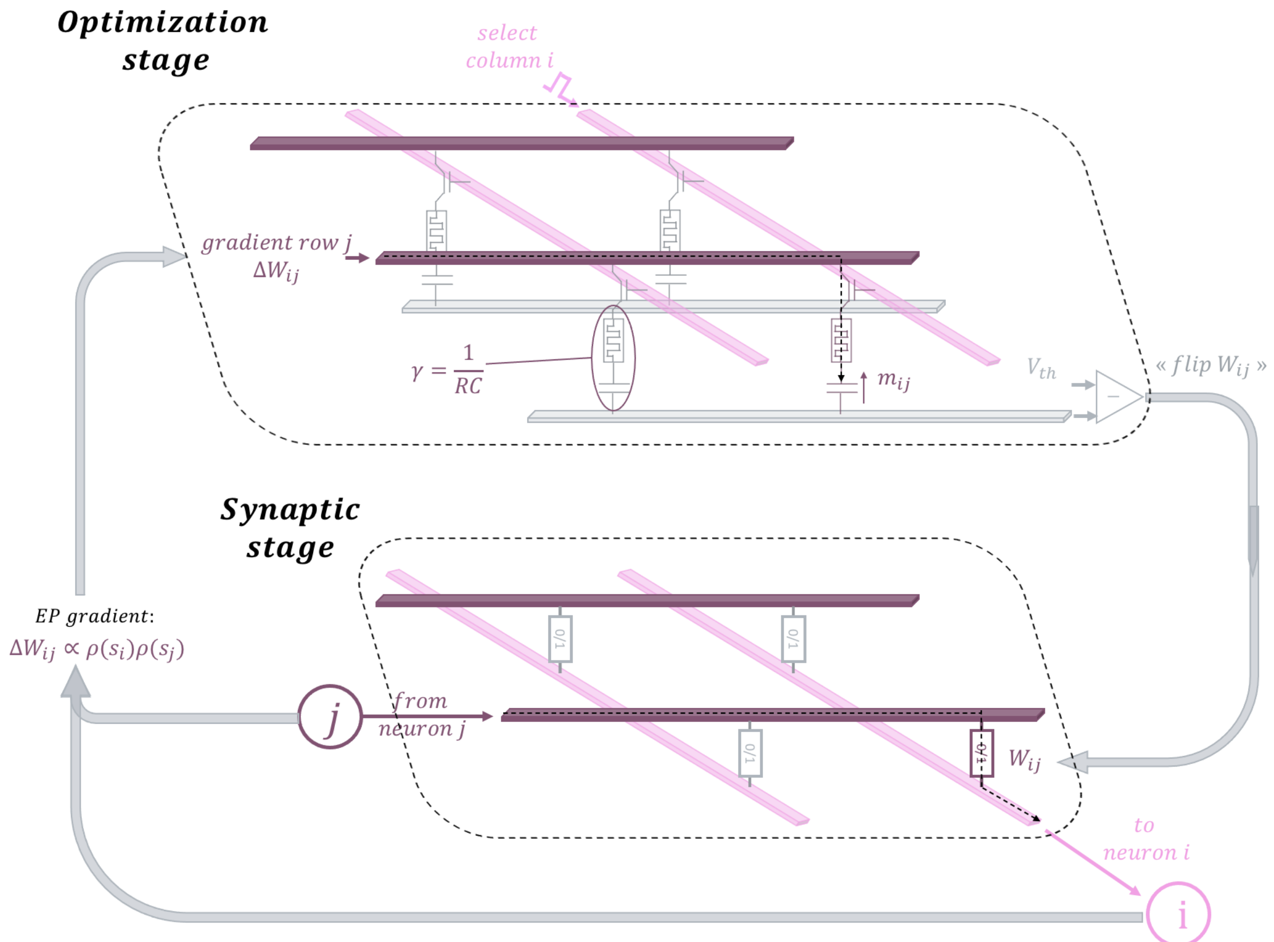


Figure 4.9: Proposition for a hardware implementation of a Binary Neural Network whose synapses are made of emerging nano-devices arranged in a cross-bar array. The synaptic array is topped by a cross-bar array which has the same dimensions that contains one couple (memristor, capacitor) per synapse to store the corresponding momentum m_{ij}. The charge of the capacitor, thus its voltage, is updated by pulses that correspond to the gradient computed by EP from the state of the neurons the synapse W_{ij} connects.

4.5.2 . Hardware implementation of fully binarized Equilibrium Propagation

System-level memory considerations. When binarizing the activation in addition to the synapses, we had to increase the number of neurons in each layer compared to full-precision models:

by 16 for the fully connected layers resulting in 8192 neurons per hidden layer and by 8 for the convolutional architecture which has 256 and 512 channels per respective layer, to get accuracy approaching reported results with full-precision architectures. But considerable gains in terms of memory and computing are achieved due to the way the gradient is computed and because of the binary activations.

In terms of memory, neurons only have to store 1 bit as the first equilibrium state. That way, the communication between neurons is not only binarized, but in addition, compared to previous works on EP achieving binary communication through spikes [168], [169], [203], our method drastically reduces the memory to compute the gradient. Indeed, spikes need to be stored for several time steps to get an estimation of the firing rate of each neuron, resulting in heavy memory requirements. EqSpike [169] estimate the firing rate with a local low-pass filter which can be done without storing the averaged activity with 32 or 64 bits. Furthermore, as the weights are updated in an event-based fashion, no updating signal needs to be applied to every synaptic weight thus reducing the complexity of the hardware required. It could be interesting to merge both frameworks: EqSpike + Binary EP for a hardware implementation to leverage the benefits of both frameworks.

Despite the fact that we need to enlarge the output layer depending on the architecture, we show in Appendix 4.3.2 that probing the state of a single neuron per class in the output layer is sufficient to obtain almost the same accuracy than when averaging the states over all the output neurons, which is beneficial for lowering the energy consumption of hardware (Figs. 4.7-4.8).

In addition, contrarily to BNNs trained by BP where we need to store every intermediate operation, including the integer pre-activation of each layer between the forward and the backward passes in order to compute the full-precision gradient, here we only need to store the 1 bit activation after each phase in order to compute the gradient, which drastically reduces the memory requirements of the model for training by a factor 40. The current implementation of binary EP on GPUs, however, still relies on full-precision neuronal state $s(t)$ variables. In future implementations of binarized EP on dedicated hardware, these neuron states can be implicitly encoded through the dynamics of nano-devices, thus solving this issue [189].

Now that we had an overview of the potential gains in terms of memory that a hardware binary implementation of EP could offer, we next discuss how we could implement digital neurons with standard CMOS hardware and more precisely with low-level functions that are emulated by elementary logic gates.

Gradient computation with low-level operations. The gradient estimate $\mathbf{g_{ij}}$ is indeed now ternary (Eq. 4.4), and can be easily computed with the subtraction of 2 AND operations. With the notations of Eq. 3.27 it decomposes as:

$$\Delta W_{ij} = \frac{1}{\beta}\left(AND(\rho(s^{\beta}_{i,*}), \rho(s^{\beta}_{j,*})) - AND(\rho(s^{0}_{i,*}), \rho(s^{0}_{j,*}))\right) \tag{4.13}$$

which amounts to only 5 elementary operations including the subtraction and involve little CMOS circuitry (4 transistors). Furthermore, this computation stage can be share among many couples of neurons so it has little area impact.

Binary dynamics with low-level operations. We now discuss how the dynamics of the neurons that have a binary activation function along with binary synapses - whether they are implemented

with standard memory or with NVMs arranged in cross-bar arrays - can be restated in terms of elementary operations on binary vectors. This quick overview can help visualizing what a dedicated CMOS implementation could look like.

- *Binary Synapses:* Previously we have focused on the case where hardware binary synapses are implemented with NVMs. In this configuration, the MAC operation between an input vector and the synaptic elements is directly computed by Kirchhoff laws (see Section 3.3.3). But we can also imagine that the synaptic elements are stored as bits in binary vectors with standard CMOS memory. In that case, as the binary activation is really 0 or 1 (and not -1 or 1 as in XNOR-Net [173]), we can not simply use the XNOR operation between the input vector and the synaptic weights, which are really -1 or 1 but encoded as 0 and 1 in binary vectors. But we can overcome this issue by performing two AND operations. The first between the binary input vector and the vector of binary weights, the second between the binary input vector and 1-vector of binary weights:

$$x_i \propto -\frac{\partial E}{\partial s_i} \propto \sum_j W_{ij} \rho s_i = popcount\left(AND(\rho(s_j), W_{ij}) - AND(\rho(s_j), 1 - W_{ij})\right) \quad (4.14)$$

  This is easily done in hardware and does not exhibit overhead in area and energy as an AND gate is half the number of transistor than a XNOR gate.

- *Binary Neurons:* The main characteristic of the binary neurons in binary EP is their internal variable that filters out temporal inputs that come from the adjacent layers. Despite the fact that this variable has been treated as a full-precision variable in our software simulations of binary EP, we believe that it can be reduced in precision to being an integer (Int32 or lower) despite being initially a floating point (Float32)[2]. Thibaut Loiseau, a previous intern in the team, has carried out simulations on a fully-connected architecture where both neurons and synapses were binary and where he reduced the precision of the internal state of the neuron. He successfully achieved the training of a shallow network that has 2048 hidden neurons when the internal variable is a 6 bits integer variable with little loss of accuracy (-1% of test accuracy). In this context, this reduces to only 6 bits the precision required to store the internal variable of the binary neurons in the case where we store that variable with standard memory. As said before, we can also consider to use unconventional devices to store that variable such as a capacitor [204] whose time constant would be large.

  Now we describe with more details how starting from the gradient dynamics:

$$s_{t+1} = s_t + dt \cdot \left(-\frac{\partial E}{\partial s}\right) \quad (4.15)$$

  we can derive a dynamics that is very CMOS-compatible.

  We now distinguish two contributions from $-\frac{\partial E}{\partial s}$: $-\frac{\partial E}{\partial s}^{s} = -s$ and $-\frac{\partial E}{\partial s}^{W,b} = \rho'(s_i)\left[\sum_j W_{ij}\rho(s_j) + b_i\right]$ such as $-\frac{\partial E}{\partial s} = -\left(\frac{\partial E}{\partial s}^{s}\right) - \left(\frac{\partial E}{\partial s}\right)^{W,b}$. As said in the previous paragraph, $-\frac{\partial E}{\partial s}^{W,b}$ is either computed naturally by the Kirchhoff law if the synapses are implemented with NVMs or with simple

[2]Converting a Floating point variable that is between 0 and 1 to an Integer requires to scale-up the Floating point variables and only then one can operate a rounding operation to get an integer variable. This scale-up necessitates small transformations regarding the MAC operation but still can be performed. Moreover, one can shift the bias into the activation despite being added to the internal state at each time step (the low-pass filtering of a constant outputs the constant) . It reduces a lot the computation and allows keeping a high precision bias.

AND operations as described in Eq. 4.14 when the binary weights are stored with standard digital memory. Then, Eq. 4.15 combined with the special case of the gradient dynamics (Eq. 3.16) derived from the Hopfield Energy (Eq. 2.11) that has guided our derivation of EP where $-\frac{\partial E}{\partial s_i} = -s_i + \rho'(s_i)\left[\sum_j W_{ij}\rho(s_j) + b_i\right]$ allow us to rewrite Eq. 4.15 as the dynamics of a leaky integrator neuron[3]:

$$s_{t+1} = (1 - dt) \cdot s_t + dt \cdot \left(-\frac{\partial E}{\partial s}^{W,b}\right) \tag{4.16}$$

where $-\frac{\partial E}{\partial s}^{W,b}$ depends on the actual hardware implementation but is easily computed as stated above.

Now, when we use $dt = 0.5$ Eq. 4.16 simplifies as:

$$s_{t+1} = 0.5 \cdot s_t + 0.5 \cdot \left(-\frac{\partial E}{\partial s}^{W,b}\right) \tag{4.17}$$

Starting from here, a natural way to store the internal variable of the binary neurons, that can be reduced in precision to an integer, would be to use a shift register (of 6 bits for instance). Indeed, a division by 2 is simply a right shift for such hardware element. So we rewrite Eq. 4.17:

$$s_{t+1} = RightShift(s_t) + RightShift(\left(-\frac{\partial E}{\partial s}^{W,b}\right)) \tag{4.18}$$

We now have the building bricks for considering to build an actual hardware implementation of EP. The implementation can be either based on standard CMOS devices or based on a mix of standard CMOS devices with emerging nano-devices that are CMOS compatible. We sum the whole training procedure in the case where the internal variable in stored with shift registers that are standard CMOS memories in Alg. 8.

Aggregating those possibilities in a digital neuron. Such a neuron could be summed up as in Fig. 4.10.

Such an approach is promising as it requires very little memory per neuron: 6 bits to store the internal variable and 6 bits to store the input. Depending on how the input-synaptic weights vector-matrix product is computed for computing the input, a time step of the dynamics could requires very little clock tops: both $RightShift$ can be synchronized and the bias can be integrated in the threshold of the activation, thus eliminating one addition, making the dynamics of such a system very fast.

The implementation that relies on the use of a cross-bar array easily solves the required dense connectivity between layers problem that we still face if we store the binary weights in a standard digital memory.

[3]the term s_i^2 in the Hopfield Energy function (Eq. 2.11) encodes the leaky term in the dynamics)

Algorithm 8 CMOS hardware driven binary EP algo: synapses and neural pre-activations are binarized, the EP gradient estimate g is ternarized, we extensively use elementary operations on binary vectors

1: **Input**: x, y, $s = Int6$, β, $\theta = \{\mathbf{W}, b\}$, η, m, γ, τ.
2: **Output**: $\theta = \{\mathbf{W}, b\}$, m.
3: **for** $t \in [1, T]$ **do** ▷ 1. Free phase
4: $\quad s \leftarrow RightShift(s) + RightShift\left(-\frac{\partial E}{\partial s}^{W,b}\right)$
5: **end for**
6: $s_* \leftarrow s$
7: **for** $t \in [1, K]$ **do** ▷ 2. Nudge phase
8: $\quad s \leftarrow RightShift(s) + RightShift\left(-\frac{\partial E}{\partial s}^{W,b} - \beta \times \frac{\partial \ell(\mathrm{y},\mathrm{y})}{\partial s}\right)$
9: **end for**
10: $s_*^\beta \leftarrow s$
11: $g^W \leftarrow \frac{1}{\beta}\left(AND(\rho(s_i^{*,\beta}, s_j^{*,\beta})) - AND(\rho(s_i^{*,0}, s_j^{*,0}))\right)$
12: $g^b \leftarrow \frac{1}{\beta}\left(\rho(s_i)^{*,\beta} - \rho(s_i)^{*,0}\right)$ ▷ 3. Compute EP gradient
13: i. Update capacitors charge: ▷ 4. Apply BOP (Alg. (5))
$m_{ij}^t = (1-\gamma) * m_{ij}^{t-1} + \gamma * g_{ij}^W$
14: ii. Get signal to flip the weights:
$flip = Comparator(m_{ij}^t, \tau)$
15: **if** $flip = 1$ **then** ▷ 5. $flip$ prescribes the weights flippings
16: $\quad \mathbf{W_{ij}} \leftarrow -\mathbf{W_{ij}}$
17: **end if**
18: $b \leftarrow b + \eta \times g^b$ ▷ 6. Update biases with SGD

4.5.3 . Stochastic binary neurons?

The implementation we have envisioned just above can be made with 100% standard digital CMOS components or with a mix of emerging nano-devices and standard digital CMOS components. However, we have also seen that hardware implementations only based on such unconventional devices exhibit higher gain in energy consumption than mix CMOS-unconventional nano-devices [8]. Thus, in order to open the discussion about the hardware implementation and mainly to consider hardware implementations only built with such emerging nano-devices, we imagine to substitute the deterministic binary neurons that we have used in all our simulations and to discuss a potential hardware implementation.

There exist some binary emerging nano-devices that are binary and stochastic. The stochastic magnetic tunnel junction is a good example [205].

At first sight, adding stochasticity in the dynamics seems to be counter-intuitive. Indeed, EP functions by shaping minimum in the energy function that is defined by the state of all the neurons in the network. So turning stochastically on and off some neurons in the system depending on the input seems like it could alter the performance of the entire system.

But adding stochasticity in neural networks has been demonstrated to be helpful for training. The idea is old as stochastic neurons have been already used in Boltzmann machines [11]. Stochastic neurons can be used to improve the generalization property of the networks. Dropout [30], [206], [207] is a good method to add stochasticity in the network: for each input

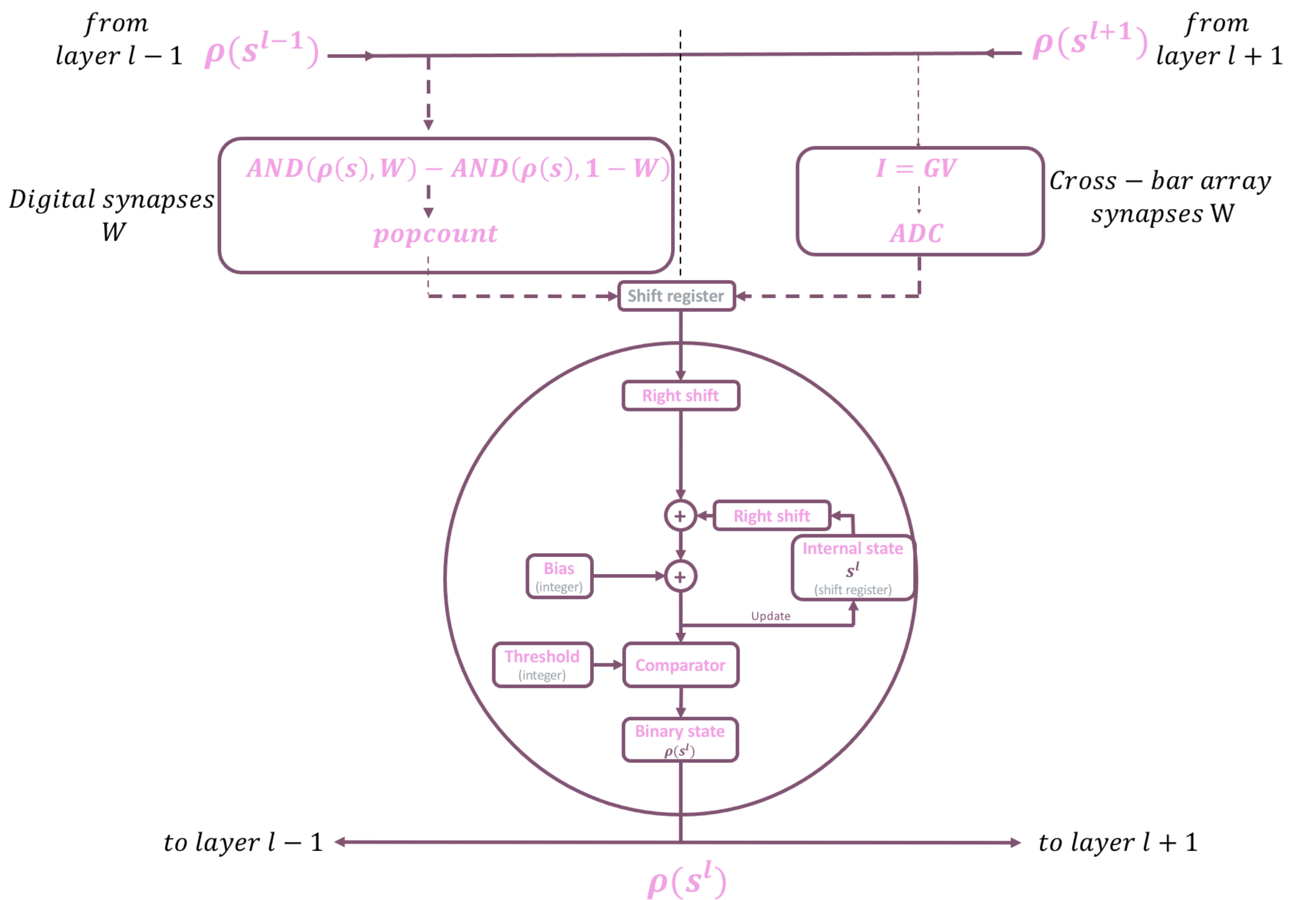


Figure 4.10: Proposition for a hardware implementation where the binary neuron is implemented with standard digital CMOS technology. The neuron receives inputs from the neighboring layers that are modulated by the synaptic weights in a way that depends on how the binary synapses are really implemented: stored as digital bits with standard digital memory or emerging devices arranged in a cross-bar array architecture. Locally, the neuron stores its internal state on a shift register, an individual bias and a threshold to binarize its internal state, threshold that can be shared among many neurons.

data, some neurons in the network are silenced with a certain probability. Drop-out allows the neurons not to specialize too much on a given feature of the training data and thus allow to improve the performance of the network on unseen data after training. It has been also demonstrated that stochastic binary neurons better compute the gradient [208].

From a hardware point of view, stochastic neurons are very appealing as they offer sparse representations, thus less power is dissipated in a cross-bar if the synapses are encoded in the conductance of resistors/ memristors but also because the stochastic binary devices that exist such as stochastic magnetic tunnel junctions are very low-power [205] and avoid the use of standard CMOS technology to implement a hardware neuron as envisioned in Fig. 4.10. Thus making the hardware implementation very compact.

Finally, we will show in the next chapter (Section 5.3) that the training procedure of EP allows to shape a network whose binary units (spins) are initially stochastic to a deterministic

behavior that always predicts the same output for a given input. This observation strongly encourages future research about using stochastic binary units for hardware implementations.

4.6 . Conclusion

As a conclusion, we provide here a binarized version of EP that exhibits only a slight degradation of accuracy compared to full-precision models.

In this chapter, we have addressed the issue of on-chip learning via EP by training dynamical systems having binary activations and weights. We first leverage the recent progress made in Binary Neural Networks (BNNs) optimization [160], to binarize the synapses in energy-based models trained by EP. The optimization of weights is performed using the inertia of the gradient. This lowers the memory required for training such systems compared to real-valued ("latent") weights optimization, traditionally used for training BNNs. We then binarize the activation functions, yielding an easy way to compute the local gradient while supporting a digital communication between neurons.

More precisely, our contributions are:

- We introduced a version of EP that can learn recurrent binary weights assuming full-precision activations (Fig. 4.1). For simplicity, we call this version of EP "binarized EP". Our implementation uses a novel weight normalization scheme directly learnable by EP. We are able to maintain an accuracy similar to full-precision models on fully connected and convolutional architectures on MNIST. We extend these results to the CIFAR-10 task, with performance only degraded by 1.9 % from that achieved with the full-precision counterpart trained by EP [186].

- We have extended our technique to fully binarized dynamical networks where both weights and neuron activations are binarized (Fig. 4.4). We demonstrate successful training on fully connected and convolutional architectures on MNIST with a slight degradation (between 0.2 and 1%) with respect to standard EP. This "fully binarized" version of EP achieves binary communication between neurons while reducing the memory required to compute the local gradient to 1 bit, making the gradient ternary.

- Our code is available at: `https://github.com/jlaydevant/Binary-Equilibrium-Propagation`.

This version of EP offers the possibility of training on-chip BNNs with compact circuitry because the hardware required for training is the same as for inference, whereas current BNNs are trained on conventional hardware, before being transferred to compact, low-energy chips. Finally, the version of EP with binary synapses and full precision activations is of major interest for future fast, low power hardware built on emerging devices. Joint development of EP and hardware will be critical for adapting EP to larger data sets.

Chapter 5

Supervised learning in an Ising Machine with Equilibrium Propagation

As we have seen in the previous chapters, the algorithm of Equilibrium Propagation is highly promising for training physical systems to perform machine learning tasks with high accuracy, as it leverages the property of energy minimization to compute the gradients in a supervised training framework.

Despite the generated interest [120], [150], [155], [157], [163], [165], [169], there are still no large scale physical implementations of this algorithm. Indeed, on one side, the emerging nano-devices that are the most promising for future ultra-low-power in-memory-computing implementations are still experimental, which means that their dynamical properties suffer from large intra and inter-device variability that constitute a major hurdle for learning (see Section 2.1). On the other side, available large-scale digital neuromorphic platforms such as TrueNorth or Loihi do not straightforwardly derive from an energy function which could be minimized for learning as desired in EP. Finally, the only existing large-scale systems natively computing through energy minimization, Ising machines (see Section 3.1), did not seem a-priori compatible with EP. They indeed are based on two-level dynamical spins, whereas the original EP algorithm exploits the smooth and gradual dynamics of analog neuron states to compute the real-valued gradients to be then applied as parameter changes in the system.

In this chapter, we will show that we can use the fully-binary version of EP that we have developed (see Chapter 4.3) using state-of-the-art machine learning methods, to train an Ising machine through gradients that we measure directly from the spins states of the machine at equilibrium.

For this purpose, we have chosen as platform an Ising Machine developed by D-Wave [134] that allows the multiple round-trips between the system and the optimization stage needed for learning, with fast parameters updates. The D-Wave Ising Machine is based on superconducting qubits that encode the state of spins in the circulation of a super-current and the system eventually reaches the ground state of the Ising Hamiltonian with the quantum annealing procedure.

We will describe how we mapped EP to the Ising system, and how we leveraged quantum annealing to reach equilibrium. We will show that we can train the D-Wave chip to solve MNIST with a fully-connected architecture, with sota result for a software neural network of the same size. These results were obtained by using an embedding procedure to circumvent the local connectivity of D-Wave between spins, that limits the number of spins on the chip that can be effectively used for computing. We finally show that the actual layout of the chip can instead be taken advantage of to perform convolutions efficiently, and solve a small task with this highly hardware-compatible architecture.

5.1 . Introduction: an Ising Machine as a parametrizable energy-based model for supervised learning: the D-Wave QPU

5.1.1 . Spin-based hardware are parametrizable energy-based models

As we have seen, physical systems of coupled spins and artificial neural networks have been intimately linked since the beginning of artificial intelligence. Thus, as pointed out in Section 2.1, the property of collective evolution of the system towards a state of equilibrium which is defined by the value of the couplings between spins motivated the first works on artificial neural networks [9]–[11], [110].

An Ising machine can thus implement a neural network, where the spins of the IM emulate hardware binary neurons and the tunable couplings between spins are the equivalent of the symmetric bi-directional synapses of Hopfield networks [9], [112] and Boltzmann Machines [11].

However, the use of a real or simulated coupled spin system to emulate an artificial neural network has since developed very little, and only for solving toy tasks [135], [209]–[212]. The main use of Ising machines today is indeed the solution of combinatorial optimization problems which cannot be treated in a reasonable time with conventional computers [122].

The first explanation is related to the limited number of spins available in Ising machines. Hopfield networks and Boltzmann machines require hardware spins to encode all the inputs, which are extremely numerous for an image classification problem: 784 for MNIST, whereas the Ising machines currently available only have a few thousand spins at most (see Section 3.1).

Nevertheless, a few works have implemented and trained Restricted Boltzmann Machines in Ising machines, in particular with the D-Wave computer [135], [137], [139], [141]. The D-Wave IM indeed perfectly fits the framework of RBMs because the stochasticity of the hardware is sought to train the network. [135] train a RBM on a coarse-grained version of MNIST in order to fit the chip that was available at the time of publication. [139] only train some couplings in between hidden nodes (80 nodes) of the Boltzmann Machine. The couplings between the visible and the hidden layer are computed in a side computer. Only [137] train on the standard dataset for machine learning tasks that is MNIST. However it is still trained layer-wise and show poor performances on MNIST/200[1] (maximum 67% test accuracy with 479 hidden nodes).

The D-Wave IM has also been used to train neural networks, and more specifically to train parts of neural networks. For example, [213]–[215] leverage the probabilistic nature of this hardware to generate a sparse latent representation of an auto-encoder. However, again, the authors do not train the whole auto-encoder on the D-Wave IM. The IM is in fact used to generate a sparse latent representation given the latent representation that the auto-encoder computes from its training with backpropagation. The objective function that the IM minimizes is a function that describes that sparsity of the representation.

The second reason why few neural networks have been implemented on IM is related to the algorithms that such systems can train. 1) Backpropagation can not be used to train such a system as the forward pass, *i.e.* the dynamics of the spins from the initial state to the ground state, is either hardly or non-derivable and 2) as we have seen, the algorithms initially used to train Hopfield networks and Boltzmann machines proposed before EP do not optimize a global cost function, which limits their performances. The fully-binarized EP algorithm that we have developed in this thesis opens new perspectives to train Ising machines in a supervised way.

When Ising machines are used to solve combinatorial problems via the minimization of their energy, the solution, the ground state of the Ising energy function, depends on the coupling parameters which are themselves fixed by the problem of interest to be solved. Such a use of the Ising machine can be called "problem-driven". For a use as an artificial neural network, the set of parameters that allows to have the minimum of the global cost function, is "found" by observing a large number of

[1]We refer to MNIST/X a subset of the MNIST dataset that contains only X training images from the original training dataset that contains 60k training images.

data - which is a "data-driven" approach. In this case, the Ising machine is used both to find the ground state of the energy function in order to perform the inference step but also to determine how to update the parameters to successively minimize the cost function.

These two approaches are not watertight and one can imagine using an Ising machine as an energy-based model to solve combinatorial problems for example, and to determine the problem-dependent parameters in a supervised way, by looking at a database. Thus this approach would be much more flexible since it does not require to know the problem to be optimized and facilitates the solution since one does not need to find the Ising energy to minimize specific to the problem.

Contrarily to the classical use of "problem-driven" Ising machines where the coupling parameters are fixed and given by the problem to be solved, the ease with which the parameters can be updated is of crucial importance in the context of supervised learning, where the back and forth motion between data and the state of the neural network is frequent. This property is fundamental to make an Ising machine an ideal substrate to emulate and allow the training of a large-scale artificial neural network in a supervised way. The D-Wave IM evolves all spins in a parallel fashion and, as such, is truly similar to convergent dynamic energy-based models such as the ones we have used to perform EP-trained simulations so far (Section 3.2 et Section 4.3).

5.1.2 . A specific case: the D-Wave IM

To carry out the work presented in this chapter, we have chosen to use the Ising Machine proposed by D-Wave. It is interfaced with a Python API that allows to apply the desired parameters to the chip so that it returns the state of the spins that minimizes the corresponding energy. As we will see later, this IM is based on superconducting materials and therefore requires to be operated at very low temperature (20mK), which is far from the initial low power objectives of our devices of interest. Moreover, this machine works as a cloud service accessed via the API developed by D-Wave and distances us even more from our initial object of study. However, the main interest of our work on this hardware is to demonstrate that we can perform supervised learning by introducing the parameterization in the energy function of a system that naturally evolves towards a state that minimizes this energy, thus opening the way to other implementations based on miniaturized, low-power hardware. The results we are going to present were made possible by the relative speed and accuracy of the parameter update phase allowed by the chip with a very decentralized architecture to operate the updates [216].

The Ising machine built by D-Wave encodes the $-1/+1$ state of spins in the circulation of a super-current in a flux qubit. The circulation of the super-current can take two possible classical states: the clockwise $|\circlearrowright\rangle$ or the counterclockwise $|\circlearrowleft\rangle$ [217]. These individual qubits are coupled to other qubits by inductive couplings. D-wave uses a quantum annealing procedure, based on the quantum tunneling effect between the circulations of the superconducting currents, to help the spin system thus constituted to escape from undesirable local minima and reach the ground state of the corresponding Ising energy function.

This procedure for guiding the system towards the equilibrium state is inspired by the procedure of simulated annealing or thermal annealing. We will now describe in more detail the mechanisms involved in the QA procedure that allow the system to evolve towards its fundamental state. We will then see which hardware components are used in the chip designed by D-Wave to get a more concrete idea of how the Quantum Annealing procedure is physically implemented.

Raw description of QA. Although physical systems evolve globally so as to minimize their energy (see Section 2.1 for discussion), it happens that, when the landscape of this energy is complex and

highly non-convex, the system gets stuck in a configuration that corresponds to a local and not a global minimum. While we use this property to encode patterns into local minima in a Hopfield network, this becomes a major problem for the operation of an Ising machine which has to find the ground state of a given energy function.

We have already described the simulated annealing or thermal annealing procedure in the Section 3.1 where the controlled temperature applied to the physical system gives the spins a stochasticity that allows them, sometimes, to escape from such undesirable local minima.

Instead of using temperature as an annealing parameter, the D-Wave machine uses the tunneling effect combined with a control parameter of the energy gap between the ground state and the first excited state to reach the ground state of the global system. The quantum annealing procedure is, above all, allowed by the quantum nature of the circulation of the super-current that can be either in classical state or in a superposition state. This quantum annealing procedure is in fact derived from the adiabatic theorem [218] which is a fundamental theorem in quantum physics. This theorem states that: "A physical system is maintained in its instantaneous eigenstate if a given perturbation acts on it slowly enough and if there is a significant interval between the eigenvalue and the rest of the spectrum of the Hamiltonian" [218].

This theorem is of major interest in the case of Ising machines and theoretically guarantees the convergence of the quantum annealing process to the ground state if the system is perturbed slowly. Indeed, if we imagine that we can prepare a quantum system in the ground state of an initial Hamiltonian $\mathcal{H}_0$ and that we perturb it in such a way that it is finally subjected to the Ising Hamiltonian of interest $\mathcal{H}_{Ising}$ to be minimized, that is the quantum annealing procedure, then we can expect that the system remains at any time of its evolution in the ground state of the time-dependent Hamiltonian. Thus, if we measure the state of the spins at the end of the annealing procedure, there is a high probability that it is the $\mathcal{H}_{Ising}$ ground state.

This procedure can be rewritten in the following way:

$$\mathcal{H}(t) = A(t) \cdot \mathcal{H}_0 + B(t) \cdot \mathcal{H}_{Ising} \tag{5.1}$$

where $\mathcal{H}(t)$ is the time-dependent Hamiltonian to which the physical system is subject at all times and $A(t)$ and $B(t)$ are the two annealing parameters that drive the system to the ground state of the Ising energy to be minimized.

To understand how A and B act on the system where the spin state is encoded in the super-current flow, one can imagine to do the correspondence between A and a transverse magnetic field B_x applied on the transverse component x of the spins, and between B and a longitudinal magnetic field B_z applied on the longitudinal component z of the spins. Thus, by decreasing A and increasing B during the annealing, the spins pass from one base to the other according to the parameters of the Ising Hamiltonian which applies along the longitudinal axis. By adjusting these two fictitious magnetic fields, one progressively digs wells in the initially smooth energy landscape, that encode minima of the Ising energy function to be minimized. The tunneling effect allows spins trapped in local minima to join the global minimum of the system during annealing.

Intuition of how QA is implemented on hardware. More concretely, the super-currents of the D-Wave qubits can be described in two possible bases: σ_z is the rotational basis of the $|\circlearrowleft\rangle, |\circlearrowright\rangle$ current and σ_x is the rotational basis of the symmetric and anti-symmetric combinations of the super-current flow : $\frac{1}{\sqrt{2}}(|\circlearrowleft\rangle + |\circlearrowright\rangle)$, $\frac{1}{\sqrt{2}}(|\circlearrowleft\rangle - |\circlearrowright\rangle)$. The spin encoding is done according to the σ_z basis with a ± 1 spin depending on whether the super-current flows clockwise or counterclockwise

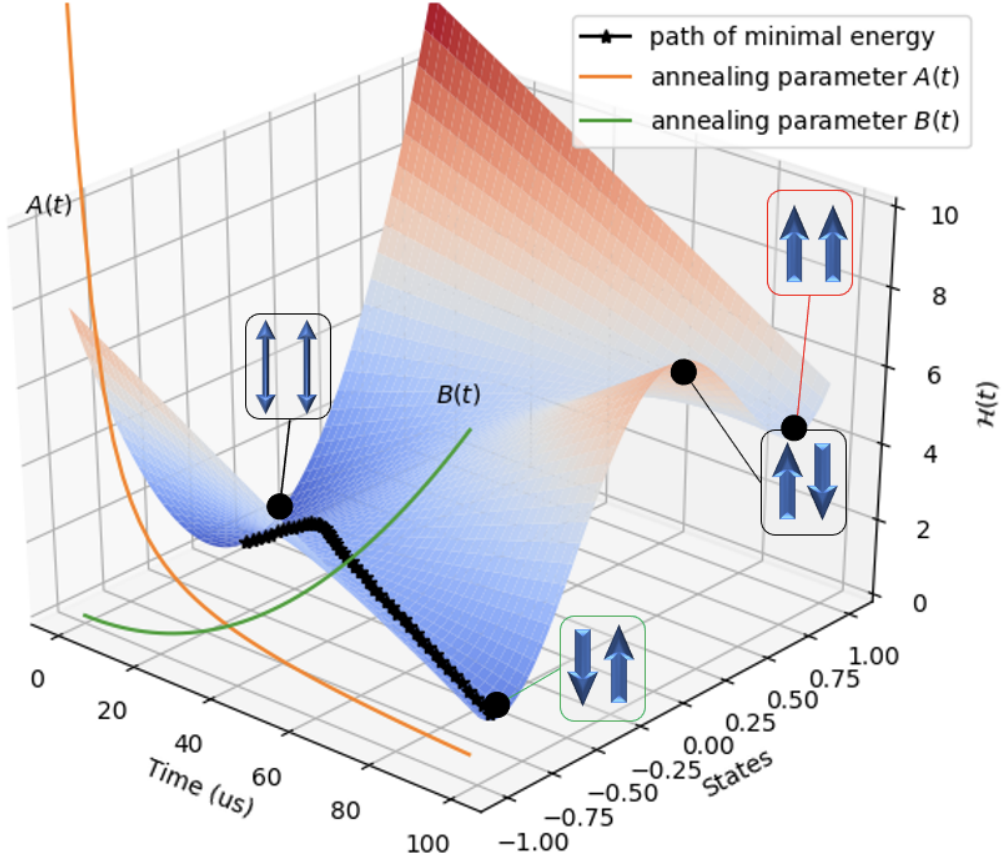


Figure 5.1: Evolution of the energy landscape of a quantum system made of two coupled qubits whose Hamiltonian is $H_{Ising} = J_{12}\sigma_z^1\sigma_z^2 + h_1\sigma_z^1 + h_2\sigma_z^2$ (with $J = 1$, $h_1 = +1$ and $h_2 = -1$) being annealed under the procedure of Quantum Annealing. The system is slowly driven by the Annealing parameter toward the ground state of the Ising Hamiltonian of interest. Given this Hamiltonian, the anti-parallel state with σ_z^1 headed down and σ_z^2 headed up is favored as it is the state with the lowest energy.

according to the σ_z basis. Thus, the time-dependent Hamiltonian (Eq. 5.1) can be rewritten as follows:

$$\mathcal{H}(t) = A(t) \cdot \mathcal{H}_0(\sigma_x) + B(t) \cdot \mathcal{H}_{Ising}(\sigma_z) \tag{5.2}$$

We can already see that by appropriately projecting the current from the σ_z basis to the σ_x basis and vice versa, we will be able to allow the circulation to reverse by passing through a superposed state where the projection on an eigenstate of the σ_z basis will depend on the external influences applied to a qubit - i.e. on the couplings we apply to this qubit. However, the contribution to the superposed state must decrease with time since we want the system to end up in a spin configuration on the σ_z basis.

We see in the equation 5.1 how by varying the annealing parameters $A(t)$ and $B(t)$ we can influence the pure current circulation projected on the σ_z basis by adding state superposition with the σ_z basis. If $A(t)$ is a decreasing function with time, then the time-dependent Hamiltonian of

the system of interest will initially tend to make the super-currents evolve in a superposed state and then progressively let them "project" onto the eigenstates of the σ_z basis according to the couplings applied between qubits.

In Fig. 5.1 we draw the time-dependent energy landscape of system made of two coupled spins. The system, initially prepared in the degenerated superposed state, is slowly driven toward the state the is the ground state of the Ising Hamiltonian to minimize. We see the time evolution of the annealing parameters $A(t)$ and $B(t)$ that either decrease or increase over time and therefore shape the energy landscape by digging holes for the minima (local or global). We expect the system to follow the trajectory highlighted by the black stars. It is especially true when the energy landscape is as simple as in Fig. 5.1 as we deal with only two coupled spins. However, in practice and when the number of spins is much larger, the system can follow other trajectories where, through tunneling effects, it can go from one minimum to another and reciprocally, which allow it to explore the energy landscape similarly to the thermal effects introduced in Simulated Annealing, until it reaches the ground state of the system.

Description of the hardware Now that we have a device, the flux qubit, that can physically encode a spin state, and a procedure, quantum annealing, that allows us to guide the system of coupled spins towards the ground state of the Ising Hamiltonian, we still have to see how the qubits are integrated and physically coupled on a chip. This is of paramount importance since the size of the problem that we will be able to solve will strongly depend on the configuration of the couplings.

Physically, the persistent current qubits used to encode the spins are RF-SQUIDs (RF-Superconducting Quantum Interference Device) with a so-called "compound-compound Josephson-junction" structure. This type of device is a superconducting loop in which Josephson junctions are arranged.

These qubits are coupled together by tunable DC-SQUIDs which interact with two neighboring qubits.

These SQUID devices are interesting because we can bias them and thus change their physical properties with magnetic fields. Thus, we can easily change the coupling between two qubits by applying a different magnetic field to the DC-SQUID governing the coupling between two RF-SQUIDs. Similarly, the annealing parameters can be provided via current lines that are inductively coupled to the RF-SQUIDs. These current lines can be common to all the qubits so that they effectively evolve at the same time.

More concretely, by applying different appropriate magnetic fluxes to the different components, one can:

1. Apply the annealing signal that acts on the tunneling energy between the two states of the σ_z basis via $I^x_{ccjj}(t)$. This annealing flux is provided by a current line common to all qubits and coupled by an inductance to each qubit to produce the annealing flux. The quantum annealing scheme thus presented allows to have only one annealing signal to send to the chip and reduces the hardware complexity.

2. Apply a fixed coupling between two qubits to implement in hardware the coupling between two spins of the Ising Hamiltonian via $\Phi^x_{co,jj}$ applied to the DC-SQUID (Coupler ij).

3. Apply an individual constant bias to each qubit, that partly defines the Ising Hamiltonian $\Phi^x_{Ip,i}$.

4. Apply a time-varying current $I_g(t)$ that compensates the drift of the constant bias due the annealing signal applied.

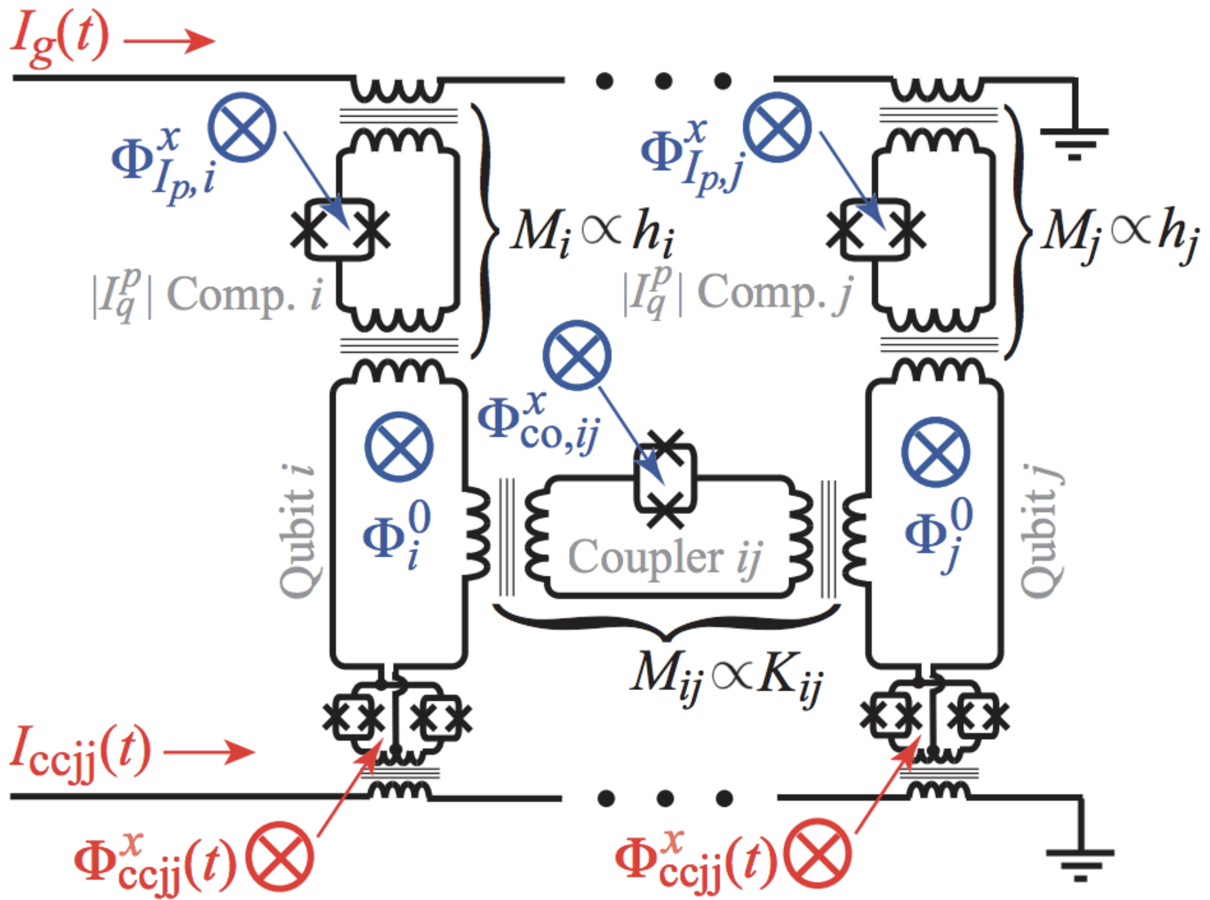


Figure 5.2: Electrical schematic of the hardware circuit for having two qubits that represent two spins of the Ising Hamiltonian to minimize and the coupler (M_{ij}) between them. Bottom current line provides the annealing schedule via the annealing flux $(\Phi^x_{ccjj}(t))$ applied to each qubit. Top current line provide a bias current that aims at keeping the bias flux - that emulates the bias field (h_i) of each spins - constant during the annealing procedure for each qubit. Figure from [134]

We now have ways to implement the quantum annealing procedure on hardware. However, as the couplers that allow the coupling between two spins necessitates that the qubits are physically close, this poses a problem when we want to couple more than a few qubits between them.

In fact, the choice made by D-Wave to use RF-SQUIDs devices for the implementation of quantum annealing, makes the coupling beyond the nearest neighbors impossible. D-Wave has proposed two types of architectures for its chips where each qubit is coupled to a greater or lesser number of nearest neighbors:

- The first architecture is the so-called "Chimera" architecture [134], [212], with clusters of 8 qubits made of 4 qubits mutually coupled to 4 other qubits. Moreover, each qubit is coupled to two other qubits belonging to two adjacent clusters. This architecture is summarized in Fig. 5.3.
- The second is the "Pegasus" architecture which allows a qubit to have up to 16 couplings with nearby qubits[219].

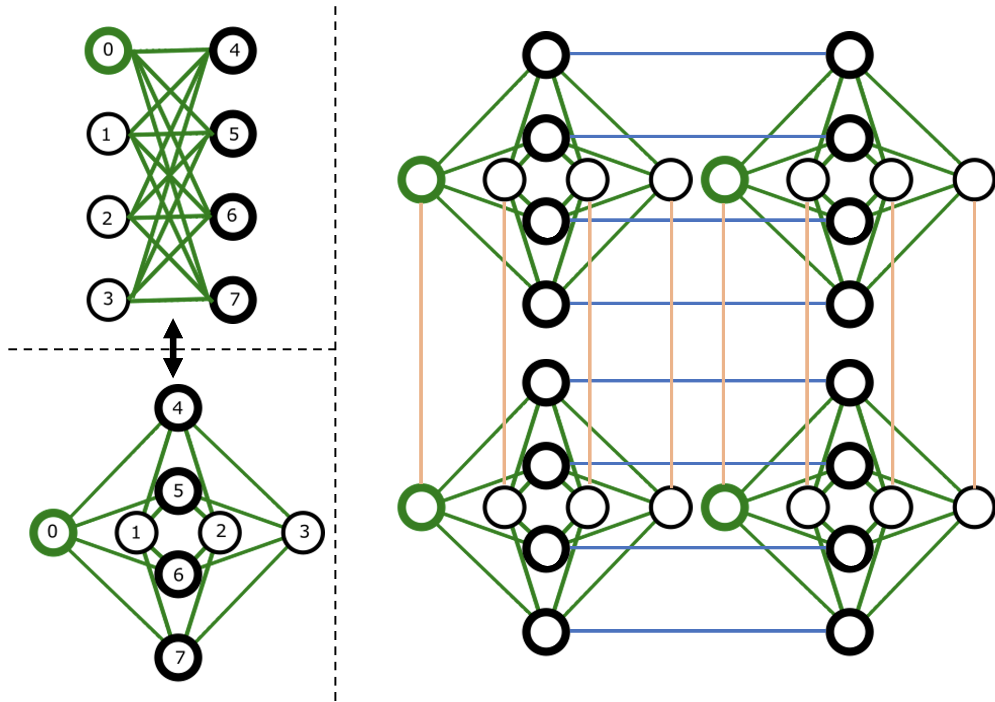


Figure 5.3: Chimera connectivity: cluster of 8 qubits where 4 qubits are fully-connected to 4 other qubits. Lateral connectivity is also available but in a 1-to-1 basis. Top left and bottom left are the representation of the same cluster of 8 qubits. Right schematic is the layout of 4 coupled clusters. Local full connectivity is highlighted in bold green and lateral connectivity is in bold blue and bold yellow.

Although the connectivity improves with the Pegasus layout compared the Chimera, this design still suffers from the lack of coupling to more than 16 neighbors and cannot directly be used to solve problems that have more densely coupled variables.

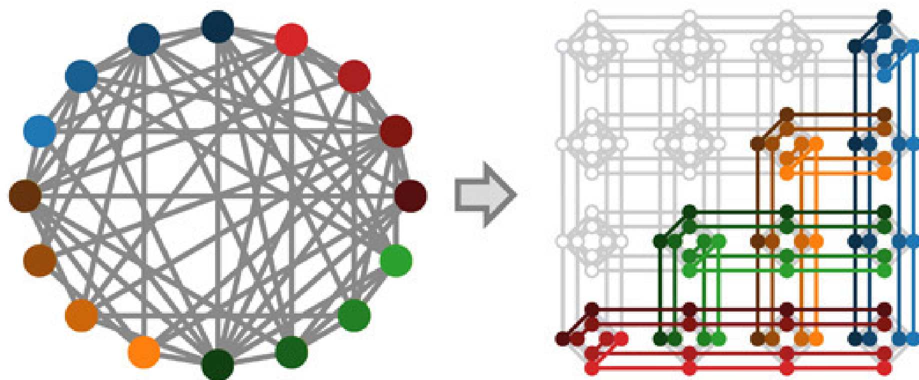


Figure 5.4: Embbeding a dense Ising problem onto the Chimera layout: each individual spin of the Ising problem is mapped to multiple qubits on the chip in order to be coupled to multiple other qubits. From [129]

To be able to solve Ising problems whose size exceeds that of the all-connected clusters available on the chip, we can use an embedding procedure to map our problem to the chip architecture [220], [221]. This procedure will use several qubits strongly coupled together to represent the same spin on

the chip. This set of different qubits supposed to represent the same spin value is called a "chain" for which we can set the value of the strong coupling parameter. Thus, with this procedure, we can implement problems with dense connectivity on the chip despite couplings that are only local on the chip (see Fig. 5.4).

However, this procedure has limitations since the size of the problem to be solved on the chip decreases strongly when using several qubits for a single spin. Moreover the time to find the solution is also longer since the signal from one spin on one side of the array will take longer to propagate through all the strongly coupled qubits. The time to solution for such a system with N spins scales as N^2 [123].

We will concentrate on the Chimera architecture (D-Wave chip DW_2000) since for the following results it will be the most interesting to look at. However, the Pegasus architecture (D-Wave chip Advantage 4.1) has also been used for getting the results we will show with the fully-connected architecture as the chip contains more qubits and more couplings so larger problems can be embeded.

5.2 . Equilibrium Propagation allows supervised learning in a Ising Machine

In this section we first integrate the energy-based model that is the IM of D-Wave in the EP training loop.

We show that the IM can achieve the two phases required by EP in a simple way thanks to the symmetric couplings and by adjusting the annealing procedure. From the steady state of the IM, a very simple gradient is computed, for now on an off-chip computer, but local circuits can be imagined for future dedicated designs. Finally we summarize the EP training procedure with IM-in-the-loop that we will use later to demonstrate that EP can supervisely train a physical system by computing the gradients of the global cost function only through the dynamics of the system.

5.2.1 . D-Wave IM can perform both the inference and the error-backpropagation through its intrinsic dynamics alone

As we have already seen, EP requires 2 successive phases of the same system dynamics to realize the inference phase and the error backpropagation phase.

If the equivalent between spins, couplings and neurons and synaptic weights is obvious: spins play the role of binary neurons and couplings between spins play the role of synaptic weights, to achieve the two phases of EP on an Ising machine driven by the Quantum Annealing procedure is less immediate.

For a given problem, typically the Ising function to be minimized is:

$$E_{Ising} = \sum_{i \neq j} J_{ij}\sigma_i\sigma_j + \sum_i h_i\sigma_i \tag{5.3}$$

where the couplings J_{ij} are equivalent to the synaptic weights W_{ij} and the local biases h_i are equivalent to the neuron biases b_i.

The Ising machine evolves towards a state $\{\sigma_i\}$ which, in principle, is the fundamental state of the Ising energy and thus the solution state of the Ising problem posed.

However, for different inputs we want the equilibrium state to be different, since we want to be able to predict different classes. We must then be able to influence the Ising energy function with the different inputs.

Encoding the input data. A solution implemented by [135], [137], [139], [141], [213], [214] was to apply a strong bias on some spins in order to represent the inputs. This has the advantage of having the inputs explicitly in the energy function, but it considerably reduces the number of qubits available on the chip to emulate neurons, thus reducing the performance of the network to solve the task. Moreover, the inputs are often quite large (MNIST has images of 784 pixels, compared to the DW2000 chip which has 2048 qubits, so MNIST is about 1/3 of the accessible qubits, before embedding!). Previous works using this technique had to resort to a method of sub-sampling the inputs to fit the available chip size.

The method we followed is to perform outside the Ising machine a first vector-matrix product between the input vector and the feedforward weight matrix linking the inputs to the first hidden layer, similar to what was done in [139]. Indeed, this product results in a static vector during the evolution of the system, since the inputs are static over time and, in the EP framework, the synapses of the input layer are unidirectional, unlike the following ones. This vector has the same dimension as the bias vector applied to the neurons of the first hidden layer. Thus, the inputs are imposed on the neural network by the bias applied to the neurons of the first hidden layer. This also allows us to have inputs of a rather large dimension that would not fit on the chip directly.

Thus the new Ising problem to solve is the following:

$$E_{Ising}^{Inference} = \sum_{i \neq j} J_{ij}\sigma_i\sigma_j + \sum_i h_i\sigma_i + \sum_{i \in 1^{st}Hidden} \left(\sum_{j \in Input} W_{ij}X_j \right) \sigma_i \quad (5.4)$$

where X is the input vector, external to the chip. We compute the input bias applied to each neuron of the first hidden layer with $\sum_{j \in Input} W_{ij}X_j$. Although the input is external to the chip, the weight matrix $W^{input,1^{st}hidden}$ will still be locally optimized thanks to the state of the spins of the first hidden layer and the value of the inputs.

Then, having a way to control the inputs presented to the chip, we can realize the first phase of EP or inference phase. The Ising machine performs the minimization of the Ising energy function corresponding to the inference phase of Eq. 5.4.

The state measured on the Ising machine at the end of the annealing procedure is thus:

$$\{\{\sigma_i^{*,0}\}|J_{ij}, b_i\} = arg\,min\ E_{Ising}^{Inference}(\{\{\sigma_i^{z,*,0}\}, J_{ij}, b_i\}) \quad (5.5)$$

Performing the nudge phase. Now the question of the nudge phase, corresponding to the retro-propagation of the error, arises. Indeed, this phase requires two features:

1. The possibility of nudging the output neurons towards their target state
2. The neurons of the system need, at the start the nudge phase, to be in the state they have reached at the end of the inference phase.

The first one is easily solved. Indeed, as in standard EP, we can simply add the Mean Squared Error cost function to the Ising energy function of the inference phase to know how to nudge the system. Thus, if we omit the explicit dependence of the biases of the first hidden layer neurons on the inputs, the Ising energy function of the error back-propagation phase is written :

$$E_{nudge} = \sum_{i \neq j} J_{ij}\sigma_i\sigma_j + \sum_i h_i\sigma_i + \beta \cdot \frac{1}{2}\sum_i (\sigma_i^o - \hat{\sigma_i^o})^2 \quad (5.6)$$

where the spins of the output layers are noted σ^o and their target state $\hat{\sigma^o}$. We can develop this energy function by developing the square of the cost function:

$$E_{nudge} = \sum_{i \neq j} J_{ij}\sigma_i\sigma_j + \sum_i h_i\sigma_i + \beta \cdot \frac{1}{2} \sum_{i \in Output} \sigma_i^{o^2} + \hat{\sigma_i^o}^2 - 2 * \sigma_i^o \hat{\sigma_i^o} \tag{5.7}$$

But as spins can only take the values ± 1, we can simplify $\sigma_i^{o^2}$ and $\hat{\sigma_i^o}^2$ to 1:

$$E_{nudge} = \sum_{i \neq j} J_{ij}\sigma_i\sigma_j + \sum_i h_i\sigma_i + \beta * \frac{1}{2} \sum_{i \in Output} 2 - 2 * \sigma_i^o \hat{\sigma_i^o} \tag{5.8}$$

and, in a simpler way:

$$E_{nudge} = \sum_{i \neq j} J_{ij}\sigma_i\sigma_j + \sum_i h_i\sigma_i - \beta \cdot \sum_{i \in Output} \sigma_i^o \hat{\sigma_i^o} + \beta * \sum_{i \in Output} 1 \tag{5.9}$$

The sum of a constant (1 here) is a constant which only shifts the global Ising energy: solving the Ising problem with or without this constant is the same, so we can remove it. This leaves the term $\sigma_i^o \hat{\sigma_i^o}$ which amounts to applying a nudge bias $-\beta * \hat{\sigma_i^o}$ to output neurons. Thus, simplifying the Ising energy function of the nudge phase, we obtain:

$$E_{nudge} = \sum_{i \neq j} J_{ij}\sigma_i\sigma_j + \sum_i h_i\sigma_i - \beta \cdot \sum_{i \in Output} \sigma_i^o \hat{\sigma_i^o} \tag{5.10}$$

This energy function corresponds to the augmented energy function of EP (see Section 3.2)and is therefore the Ising Hamiltonian to be minimized during the error backpropagation phase. However, it remains to be shown that this phase can be realized on an Ising machine driven by the quantum annealing procedure.

Nudging phase with reverse quantum annealing. QA is a forward algorithm that always starts from the superposed state of the qubits on the x basis and that drives the system toward the z basis. It theoretically prevents to do the nudge phase as EP requires the neurons to start the nudge phase from the state they have reached at the end of the free phase.

We overcame this issue by using the reverse annealing procedure.

Reverse Quantum Annealing (RQA) is a modified version of the quantum annealing algorithm initially designed to refine a first rough solution given by a first standard QA run.

We recall here the equation that drives the quantum annealing procedure:

$$\mathcal{H}(t) = A(t) \cdot \mathcal{H}_0(\sigma_x) + B(t) \cdot \mathcal{H}_{Ising}(\sigma_z) \tag{5.11}$$

where for the forward QA $A(0) >> B(0)$ which results in superposed spin states in the beginning of the annealing and $A(T) << B(T)$ with T that denotes the time at the end of the annealing procedure and results in spins that are collapsed on the $z - basis$ (see Fig. 5.5).

For performing RQA, we reverse for some time the trajectory of the annealing parameters (see Fig. 5.5): $A(t)$ is progressively increased from 0 to a given value and $B(t)$ is slowly decreased to a given value. After that, the process is reversed is again reversed: $A(t)$ is slowly decreased to 0 and $B(t)$ increased to the final value. That way, we add the possibility for the spins to acquire a little superposition on the x basis which can help the spins to escape from the state that they have

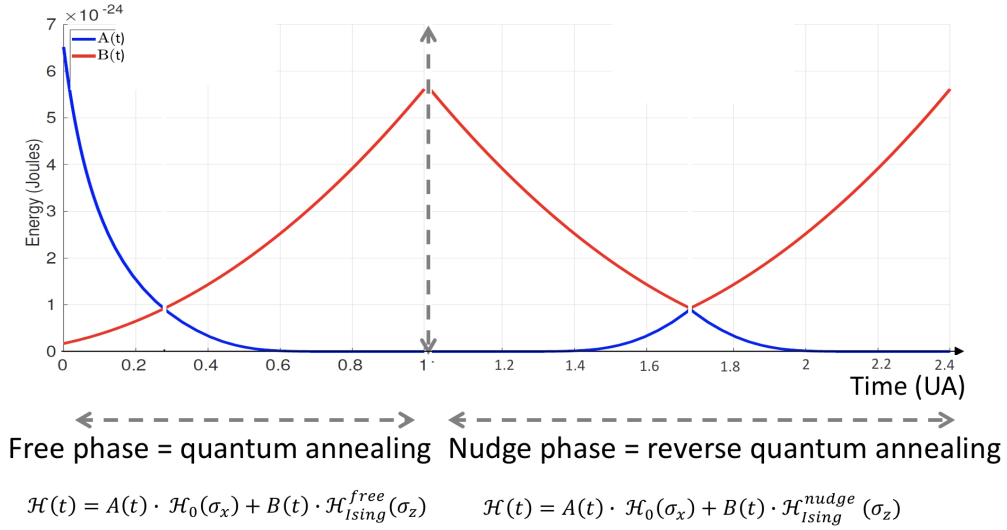


Figure 5.5: Evolution of the annealing parameters $A(t)$ and $B(t)$ for the forward Quantum Annealing with which we realize the free phase of EP, and for the Reverse Quantum Annealing with which we realize the nudge phase of EP.

reached after the free phase, according to the error signal applied through the nudging biases and the new states of the nudged spins.

To perform the nudge phase with D-Wave, we modified a bit this RQA algorithm as follows. After the forward QA of the first phase, we want to drive the output spins closer to the target state. For this, we perform a reverse annealing but we change the parameters applied to the chip compared to the first QA phase, in order to add the nudging bias to the output neurons.

To sum up, the system of coupled spins, starting from the first equilibrium state, eventually settles to a second equilibrium state thanks to the reverse annealing procedure:

$$arg\,min\ E_{nudge} = \{\{\sigma_i^{z,*,\beta}\}|J_{ij}, b_i, \hat{\sigma}\} \tag{5.12}$$

that minimizes the augmented energy function (Eq. 3.19).

So with RQA we found a way to perform the error-backpropagation phase and thus a way to perform training with EP.

5.2.2 . EP training algorithm with IM in the loop

Now that we have described how to use the D-Wave Ising machine as a parametrizable energy-based model to emulate an EP supervised trained neural network, we can write the algorithm describing the full the training loop with the Ising machine in Alg. 9.

5.3 . Training a fully-connected architecture on the D-Wave Ising Machine

We now apply the learning procedure that we have just described to train a neural network that has a fully-connected architecture on the Ising Machine of D-Wave. We describe how one can embed and train the network on that specific Ising Machine and then we report the results we obtained at training the network on a subset of the MNIST dataset (MNIST/100).

Algorithm 9 Supervised learning of a NN on the D-Wave Ising Machine with Equilibrium Propagation. Operations made on a side computer are highlighted in **violet**. Operations made on the Ising Machine via the D-Wave python API are highlighted in **pink**.

1: **Inputs:** Dataset, Architecture, Nudging Factor β, Learning Rate η, Number of free phases N, Number of nudge phases M
2: **Output:** Set of optimized weights $\{W_{ij}\}$
3: Initialise $W \leftarrow \mathcal{U}\{-0.01, 0.01\}$
4: **while** epoch $<$ N_epochs **do**
5:     **for** Input in Dataset **do**
6:         **Soft**: Instantiate the Ising problem given the input and the parameters
7:         **D-Wave**: Set $\{J_{ij}, h_i\}$ on the chip given the Ising problem and the embedding procedure
8:         **D-Wave**: Perform **N** free phases with QA - take the sample which has the minimal free energy
9:         **Soft**: Compute Train Error
10:         **D-Wave**: Apply nudging biaises on the output spins
11:         **D-Wave**: Perform **M** Nudge Phases with RQA
12:         **Soft**: Compute gradient: $\frac{\partial \mathcal{L}}{\partial W_{ij}} \leftarrow \frac{1}{\beta}(\sigma_i^{z,*,\beta}\sigma_j^{z,*,\beta} - \sigma_i^{z,*,0}\sigma_j^{z,*,0})$
13:         **for** weights **do**
14:             **Soft**: $W_{ij} \leftarrow W_{ij} - \eta * \frac{\partial \mathcal{L}}{\partial W_{ij}}$ ▷ Stochastic Gradient Descent (see Eq. 3.23)
15:         **end for**
16:     **end for**
17: **end while**

5.3.1 . Embedding the Fully-Connected neural network on the Ising Machine and updating the parameters

As described in Section 1.1.2, a fully-connected neural network is an architecture where all neurons of a layer are connected to all others neurons of the next layer. So ideally we would like to have the correspondence 1 spin on chip = 1 neuron. But, as described in Section 5.1.2, this is a kind of dense connectivity that the D-Wave Ising Machine can not afford. Thus we need to use the embedding procedure described in Section 5.1.2 and Fig. 5.4 to embed a fully-connected neural network on the D-Wave IM.

The next other major challenge we faced was to appropriately scale the weights of the network we want to train on the Ising Machine. Indeed, when they are pure software variables, the weights scale does impact the dynamics of the system but not as much as physical properties of a hardware physical system. Also, on the physical system, the parameters can be set in a given range that is not extensible contrarily to software weights. But we eventually succeeded in finding a good range of the parameters through many iterations on the IM investigating when the qubits saturate and when they don't and also looking at whether the network behaves differently when an update of the parameters is made (what we want) or not (what we do not want).

The RQA procedure was also challenging to fine-tune. As described in Section 5.2.1, RQA allows the spins to change from one equilibrium configuration, reached after the free phase for instance, to another, for performing the nudge phase. The spins change from one state to another because the reverse trajectory of the annealing parameters add the possibility for the spins to be again in a superposed state. However, the amount of superposition we can add has to be fine-tuned. Indeed,

depending on the RQA parameters we use, we can have a nudged state that is either the same as the first equilibrium state (the system is frozen) or to a completely random state (too much superposition). We will show in the next section (see Fig. 5.10) that we searched for the best RQA parameters to reach the highest accuracy on the task we train the network. We will see that the RQA parameters, mostly the amount of superposition we add to the spins, depend on the amount of spins in the system. Indeed, the larger the network is, the more we need to add superposition during the reverse annealing as spins will be more likely to stay in their initial state. This has already been highlighted with the freezing point were the larger the problem submitted to the IM is, the sooner, during the forward annealing, it will be stuck in a given configuration [222].

Finally, the crucial step was to be able to update the parameters of the Ising Machine in a fast and reliable way. The nice D-Wave API[2] allowed us to use the Ising Machine very easily and in a fast way such that the Ising Machine was easily integrated in the training loop as a hardware neural network to be trained. Parameters are sent to the Ising Machine via a Python dictionary that contains the biases to apply to each individual qubits and the couplings to apply between the relevant pairs of qubits.

5.3.2 . Results for a fully-connected architecture on MNIST

We now demonstrate the relevance of our approach by training a shallow fully-connected architecture on the D-Wave IM with the training algorithm we depict in Alg. 9 on a subset of the MNIST dataset [151].

For this section we used the D-Wave ADVANTAGE 4.0 chip which offers the highest connectivity between qubits (16 inter-qubit couplings for each qubit on the chip) so we can embed the largest fully-connected architecture possible on a D-Wave IM

The largest architecture we have been able to embed is $784 - 120 - 40$ where the input units 784 and the synaptic weights between the input and the hidden units are not physically on the chip as discussed earlier (Eq. 5.4) but rather are used to compute a vector of input biases that are applied to the hidden spins on the actual chip.

We used the standard embedding algorithm provided by D-Wave that is LazyFixedEmbeddingComposite. This algorithm finds the embedding *i.e.* the mapping architecture $\leftrightarrow$ chip layout. This procedure can be lengthy as the number of possible embeddings can be quite large. LazyFixedEmbeddingComposite is convenient as it searches for a good embedding only once for the first exemple and then re-uses the same embedding for all other exemples. Although this algorithm is faster, it also uses the same qubits on chip: so if one of them is faulty then the training process could take it into account.

The EP learning rule requires to be able to measure the state of each spin in the equilibrium states after the free and nudging phases. In practice, we sample the system multiple times per energy to be minimized (for both the free and nudged phases) as the system does not run at 0K as required for ideal adiabatic computing but rather at finite low temperature, which adds a little thermal noise and allows the spins to escape from the ground state [134], [223]. We sample the system 10 consecutive times and we retain the sample that has the minimal energy from this set of samples. This has allowed us to successfully train the IM. This also allows to sample the system only once after training to do fast-inference that a training based on an average of the samples would slow down.

Our initial goal was to train this architecture on MNIST. But due to the limited available access time to the chip (the computing access time is quite expensive), we only trained the architecture

[2] D-Wave API

on a subset of MNIST with 1000 training images and 100 testing images. Following the notation of [224], we denote this dataset as MNIST/100 where 100 corresponds to the number of training images per class. The subset was sampled in order to have an equidistribution of the images in all classes: 100 images per class in the training dataset. Between each epoch we shuffle the training images so that the training process does not overfit on the successively presented images. This task is indeed harder to solve (the test accuracy is expected to be lower) than standard MNIST as less training example are available and thus the distribution of the testing dataset is more likely to differ from the learnt distribution [224]. This task is the most complicated task solved on a D-Wave IM to date. [137] implemented a small Boltzmann Machine on subsets of MNIST which are according to our notation MNIST/5, MNIST/10 or MNIST/20. [135] also trained a Boltzmann Machine on D-Wave but they down-sampled the MNIST dataset to a coarse-grained MNIST dataset which is made of (6x6) pixels images.

Successful training of the IM. We report the first successful training of a large-scale non-linear dynamical energy-based system with EP on the IM of D-Wave. With the fully-connected architecture $784 - 120 - 40$ we achieve on MNIST/100 a training accuracy of 100% and a testing accuracy of 87% (see Fig.5.6) .

As we can not benchmark this result with literature (no training with EP has been reported on MNIST/100 yet) we conducted our own experiments to establish a benchmark to which we can compare the results we got on the D-Wave IM. We performed two kinds of simulations that are both purely software-based and based on the same type of neural network that we have trained on the D-Wave IM:

1. With the standard scheme used for performing simulations with EP: a gradient dynamics (Eq. 3.16) that is solved with a Euler scheme is used for both the free phase and the nudge phase. We only need to perform once each phase as the dynamics is not stochastic. We also use the Heaviside step function as the binary activation function (see Section 4.3.1 for the reason why we use a binary function which states are 0/1 instead of -1/1 for the spins) and the full-precision weights are updated with SGD (Section 1.1.3).

2. With Simulated Annealing (see Section 3.1) that is used to perform both the free and the nudge phase. We performed these experiments as Simulated Annealing is the closest dynamics to that of the D-Wave IM as it simulates the dynamics of coupled spins where controlled thermal effects allows the system to reach the global minimum. Similarly to the QA procedure, SA is stochastic so we also had to sample the network multiple times to get accurate equilibrium states. We also use SGD to update the full-precision weights.

Surprisingly we found that the accuracy we got on the D-Wave IM agrees with those we obtained with both software-based simulations (see Fig. 5.7). It is very encouraging because these results show that we were able to train such a stochastic system at the same accuracy than a deterministic system (dashed lines on Fig. 5.7). The same accuracy than Simulated Annealing (Fig. 5.7) is also encouraging because it shows that despite the fact that we loose some precision on the parameters on the chip (see Table 3.1 for reference) compared to pure-software simulations, this does not alter the performance of the network. It is known that training with hardware-in-the-loop, thus taking into account the imperfections of the hardware in the training process, makes the system to perform at its best at inference[109]. But the agreement between the accuracy obtained with QA and that obtained with SA is a bit disappointing as we do not see any QA advantage over SA, which

is in line with previous works where no quantum advantage has been found [225], [226]. Many arguments have been made to explain this behavior but the main reasons are the non-zero operating temperature of the chip that induces too much thermal effects on the qubits and thus decrease the coherence time of the qubits plus make the system escape from the ground state to reach near local minima due to thermal activation.

The relatively low testing accuracy that we obtain is caused by two factors:

1. The small size of the network that we were able to train on-chip. First of all, as emphasized in Chapter 4, we have to increase the number of binary neurons of an architecture in order to obtain an accuracy similar to that of continuous neurons. So despite this architecture implemented with continuous neurons in the litterature shows a better performance: EqSpike [169] trains a network which architecture is 784-100-10 and obtains 96.8% test accuracy on MNIST, this accuracy is necessarily degraded with binary neurons (and a smallest training dataset as we will show next). As the results we got with the D-Wave IM and SA match, we made the hypothesis that the results we get with SA on a digital computer can be extrapolated to the D-Wave IM for larger networks. In Fig. 5.7 we plot the accuracy obtained with SA when the number of hidden neurons is increased (with the architecture $784 - N - 40$ where N is increased). We show that both the training error and the testing error decrease when we increase N the number of hidden neurons. This is encouraging as future generations of D-Wave IM with greater connectivity and much more qubits will make possible to train larger architectures on the IM and thus get higher accuracy. The large number of neurons required to reach sota accuracy is coherent with what we have already shown in Chapter 4: when we binarize the activation functions of the neurons, we had to increase their number to reach similar accuracy than with a continuous activation function.

2. The size of the dataset is also an important factor in order to determine the accuracy we can get on the test dataset. We show in Fig. 5.8 with Simulated Annealing that if we increase the number of training images (from MNIST/10 to MNIST/300 passing by MNIST/100 on which we have trained the network on the IM) the testing accuracy increases as well. This trend is expected as the network sees more different examples so it does not specialize on particular rare features in the small dataset and is more likely to generalize the performance it reaches to unseen data. We have only been able to train on MNIST/100 as it is expensive to access the D-Wave IM. These experiments make us confident that training the D-Wave IM with more data will increase in the same manner the accuracy on unseen (testing) data.

Hierarchy. We investigate whether some hierarchy emerges during the training process on the chip, thus validating our approach to do supervised learning on the D-Wave Ising Machine. This hierarchy was not present in previous works about training neural networks on the D-Wave IM as most of the works were focused on training RBMs layer-wise.

For this investigation, we first collect and transfer the weights that have been trained on D-Wave to a software-based neural network that has the same architecture, but is a pure-feedforward neural network. We use backpropagation to reconstruct inputs that maximize the activity of some neurons in the network [31]. This time, the tensor to optimize "is" the input, the weights are being frozen. Then with this technique we can visualize which part of the input each neuron in the network is sensitive to. However, in order to do this investigation in a tight time, we had to use a differentiable activation function because it is much easier to handle with Pytorch. To stick as close as possible

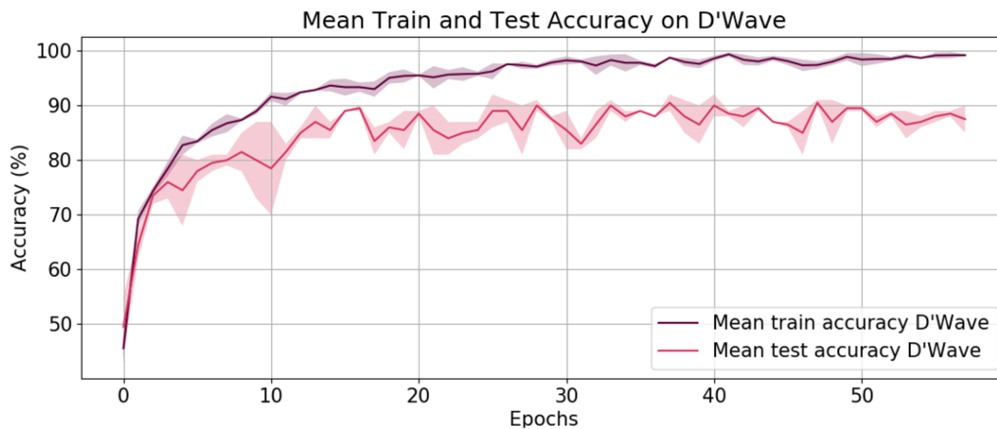


Figure 5.6: Train and test accuracy got with an fully-connected neural network (architecture $784 - 120 - 40$) trained with EP on the D-Wave Ising Machine on the MNIST/100 dataset.

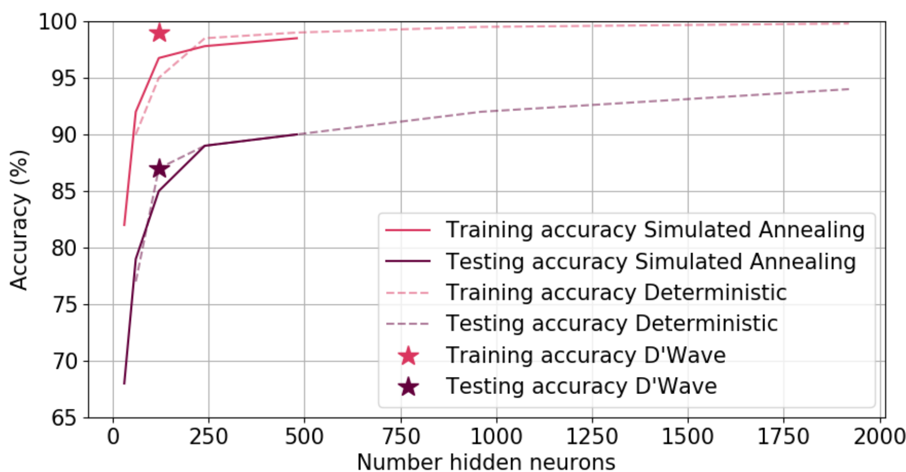


Figure 5.7: Train and test accuracy reached on the fully-connected architecture $784 - N - 40$ where N is varied for different methods to emulate the dynamics: Dashed line: accuracy obtained with the deterministic dynamics, Straight line: accuracy obtained with Simulated Annealing, Stars: accuracy obtained on the D-Wave IM.

to the binary activation of the IM we used the hard hyperbolic-tangent activation function which inputs were scaled to mimic the binary threshold and still have a non-zero gradient around the step: $\sigma(x) = HardTanh(5 * x)$.

In Fig. 5.9 we have reconstructed the input that maximizes the activity of a specific output neuron that encode the class "8" of MNIST. We see this reconstructed input is very similar to an image of a handwritten "8". It shows that this output neuron, throughout the hidden layer, is sensitive to pixels that globally form a 8 at the input, which validates our method.

We also want to visualize the pattern that activates the hidden neurons because if some hierarchy of features detected has been created during the training process, then the hidden neurons are sensitive to specific features in the input image that are combined for feeding the output neurons.

To investigate whether the hierarchy exists after training, we clamp an image of a "8" from MNIST as an input for the network. We then perform an inference on this image and we store

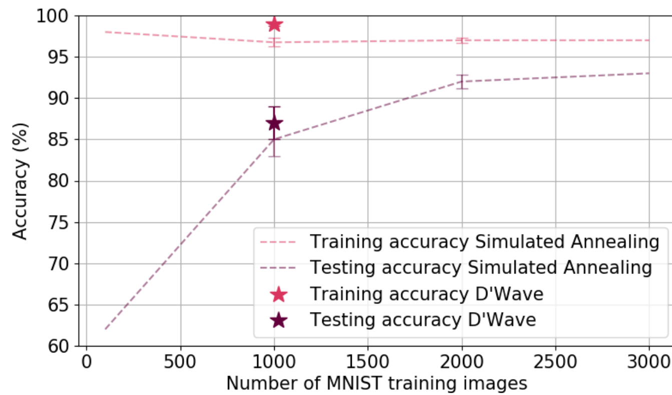


Figure 5.8: Accuracy obtained with the fully-connected architecture $784 - 120 - 40$, when we vary the size of the training dataset (still with en equidistribution of the number of images per class). Plain lines: train and test accuracy reached with Simulated Annealing. Stars: accuracies obtained when training on the D-Wave IM. Globally the testing accuracy increases when the size if the training dataset increases as the learnt distribution is wider and testing data are more likely to fall under the wider distribution learnt on the training dataset. Stars: accuracy got with QA on the D-Wave IM.

the state of the hidden neurons. We select the 10 neurons that have the highest activity during the inference on this image of a "8" . After that, we apply the same technique as before in order to find the inputs that individually maximize the activity of those 10 neurons. Then we plot the corresponding input pattern in Fig. 5.9. We see that hidden neurons activate for different input patterns that, once combined at the output layer allow the network to recognize the input.

We acknowledge that this method has limitations as we transfer the weights to a network that does not have the same activation function. But, the reconstructed inputs that we have found which maximize a specific output neuron correspond very much to what they should look like, hence telling us that the correspondence between the two networks is not tight but is not large neither. Additionally, we computed the error that this fictitious neural network (with $HardTanh$ and not re-trained) gets on MNIST/100 for both the training and testing dataset. We got 75.4% accuracy on the training dataset and 69% on the testing dataset which is not that bad considering the changes that have been made compared to the network trained on D-Wave where we got 99% of training accuracy and 87% of testing accuracy. The result we show with these methods are preliminary but are very encouraging. We expect to extend this input reconstruction method to other physical neural networks in order to better understand how physics computes.

Reverse annealing for the nudge phase. We now report in Fig. 5.10 the impact of the parameters of the reverse annealing procedure on the classification accuracy.

As explained above, we can specify which amount of "superposition" we can add during the nudging phase such as the spins evolve toward the second equilibrium state giving the error signal. We report in Fig. 5.10 the accuracy we obtained with Simulated annealing depending on the value of the RQA parameter we set for the nudge phase. In this plot, the more we are on the left, the more we add superposition to the system.

We see that in order to reach the accuracy we report in the previous paragraph, we have to add a certain amount of superposition to the system during the nudge phase. It seems that there is a threshold below which the system does not learn at all: we do not add enough superposition so the

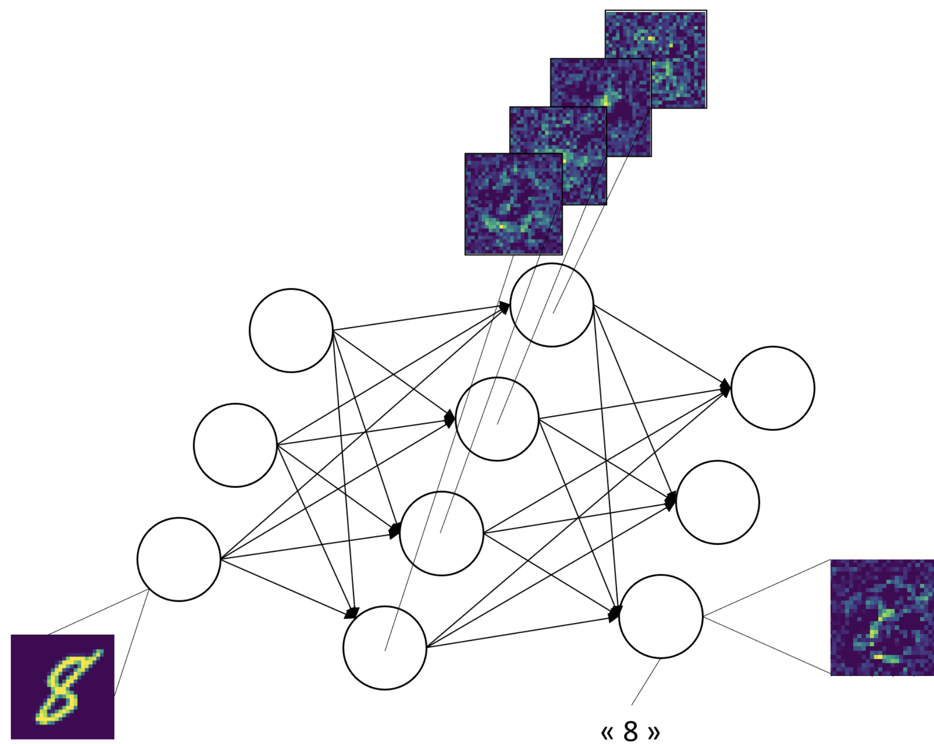


Figure 5.9: Reconstructed input patterns that maximize the activity of: 1°) one output neuron corresponding to the output neuron assigned to class "8" - 2°) hidden neurons that are the most activated during inference when presented a 8 image as an input. Hidden neurons are specialized to some specific patterns that are combined through the second weights matrix to get the inferred class "8".

spins stay in the first equilibrium state. So the gradient is 0 and the network learns nothing. This threshold in annealing parameter, above which the network learns, also depends on the size of the architecture we train. As we said before, the larger the network is, the higher the superposition we add has to be because the spins/ neurons have to be made sensitive to much more external signals.

Training the D-Wave IM with EP: from a stochastic to a deterministic machine. To achieve the results we got with the fully-connected architecture on MNIST/100, we actually had to sample many times per phase the IM. Indeed, despite being built to reach the ground state of the Hamiltonian, the D-Wave IM is a stochastic hardware for two main reasons. First, the speed of the annealing is faster than required for a perfect adiabatic calculation. Second, the machine is operated a finite temperature and not at $0K$, which adds thermal activation to the spins. This is a major drawback because one has to repeat the annealing multiple times to hopefully get a sample that is the ground state. In practice we sample the IM 10 times per training data per phase (10 times for the free phase and 10 times for the nudge phase, so 20 sampling procedures per training data).

Fortunately, it is not necessary to re-initialize the couplings and the biases on the chip for each sampling given an input data, procedure that lasts approximately $8ms$ (see Table 5.1) whereas a single annealing is only $20\mu s$. However one still has to perform the readout procedure in order to get the sample of spin states for a given input and the corresponding parameters, procedure that is

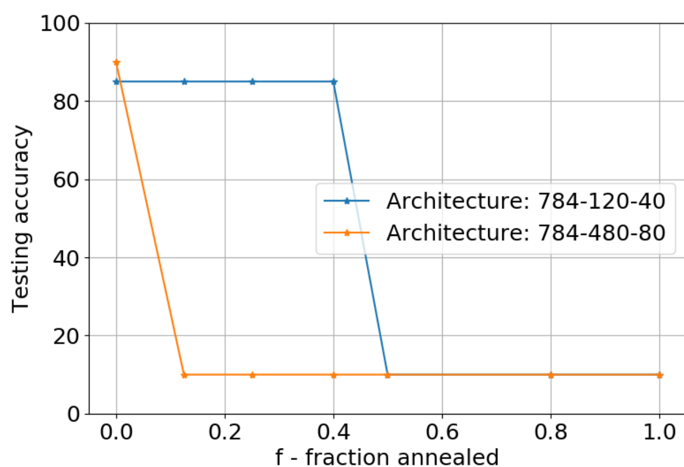


Figure 5.10: Train and test accuracy reached with Simulated Annealing for two fully-connected neural networks. We report the accuracies obtained for both architectures when changing by "how far" we go backward in the annealing process for realizing the reverse annealing during the nudge phase.

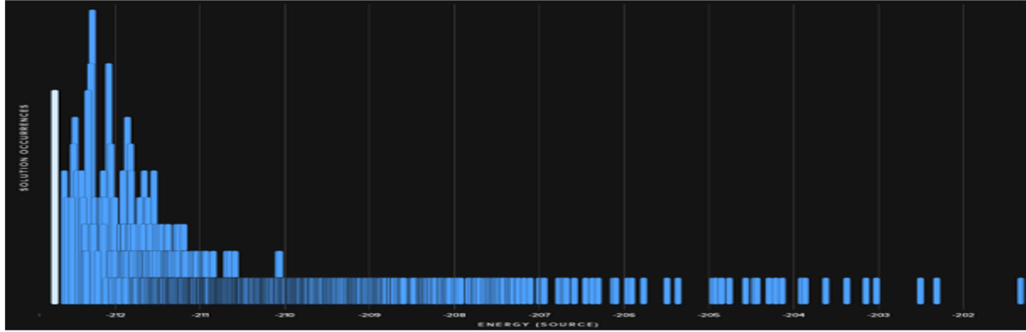
almost 10 times longer than the annealing itself.

It is crucial that the training process transforms the initial stochastic machine into a deterministic one. By deterministic we refer to the dynamics of the system that will always return the same prediction for a given input. It is true for the D-Wave IM as it is operated at a low but finite temperature so thermal effects add stochasticity to the system. This can alter the ability of this system to be trained and to be used as an inference model for deployment. But it is also true for other IMs where uncontrolled stochasticity resulting from different sources of noise (thermal, electrical,...) could also affect the performance of the IM. Hence the importance to be assured that the IM behaves deterministically at the end of the training process. This deterministic behavior will allow us to sample the IM only once and fasten a lot the inference time.

Luckily, this is what happens during the training process of the IM with EP. In Fig. 5.11a, we report the distribution of 1000 samples from the IM with the initial random weights. We see that the samples are described by a normal distribution centered around the energy -50 (though the bell curve is not observed on this figure, we can see that the bars are denser in the central region. We report the same data on histograms in Fig. 5.12) but with a large standard deviation. This distribution is characteristic of a stochastic hardware. Now it is interesting to see what happens to that same distribution but after the network has been trained. In Fig. 5.11b we see that a basin of attraction has been shaped by the training procedure such that the network converges to a state that has on average the same energy, and hopefully predicts that same class for a given input when sampled multiple times.

We also fit the distribution on samples returned by the D-Wave IM in Fig. 5.12. We compute the mean and the standard deviation of the distributions before and after training (Fig. 5.12) and plot the corresponding normal distribution below the histogram of the samples got on the D-Wave IM. We centered the energy to avoid large offsets due to the difference of the mean between the two set of samples. We clearly see that the standard deviation of the initial distribution drastically shrinks to a narrower distribution after the training process. It results in a distribution that makes that hardware quasi-deterministic, as was our initial goal.

(a) Samples distribution before training.



(b) Samples distribution after training.

Figure 5.11: Distribution of the samples before and after the training procedure with EP.

The deterministic behavior that the hardware gains during training allows us to sample the IM only once after it has been trained thus saving a lot of time during the inference process as depicted in Fig. 5.13. In this figure, we also see that in fact most of the time spent for the annealing is for programming the parameters on the chip (T_p) and some time the cool down the hardware either after the parameters are programmed on the chip ($T_{p,th}$) (the transmission lines that carry external information heat up the chip) or the read-out of the state of the system is performed ($T_{ro,th}$). The read-out step is also quite long ($T_{ro,a}$) and depends on the embedding implemented on the chip. Additionally, we see that the time for programming the chip is longer for the nudge than for the free phase. It is a consequence of the requirement to to initialize the system in the state it has reached after the free phase for each reverse annealing step which drastically increases the time of programming and thermalization.

At inference, we can imagine to have a dedicated hardware whose parameters would be fixed at the optimal set got after training so that no programming time would be required. We would only have to program the input biases that are applied to the first hidden layer which also reduces the time to thermalize the system. Finally we would only have to read-out the system once, thus saving a lot of time.

Operation	**Time**
Programming time (T_p+)	$\sim 7ms$
Thermalization time $T_{p,th}$	$1ms$
Annealing time (T_a)	$20\mu s$
Time read out ($T_{ro,a}$)	$189\mu s$
Initial state programming + thermalization (nudge phase) ($T_{p,s} + T_{th,s}$)	$\sim 7ms$

Table 5.1: Average time for each operation performed during the quantum annealing procedure (data collected on the D-Wave cloud after performing annealing test with the fully-connected architecture we have trained)

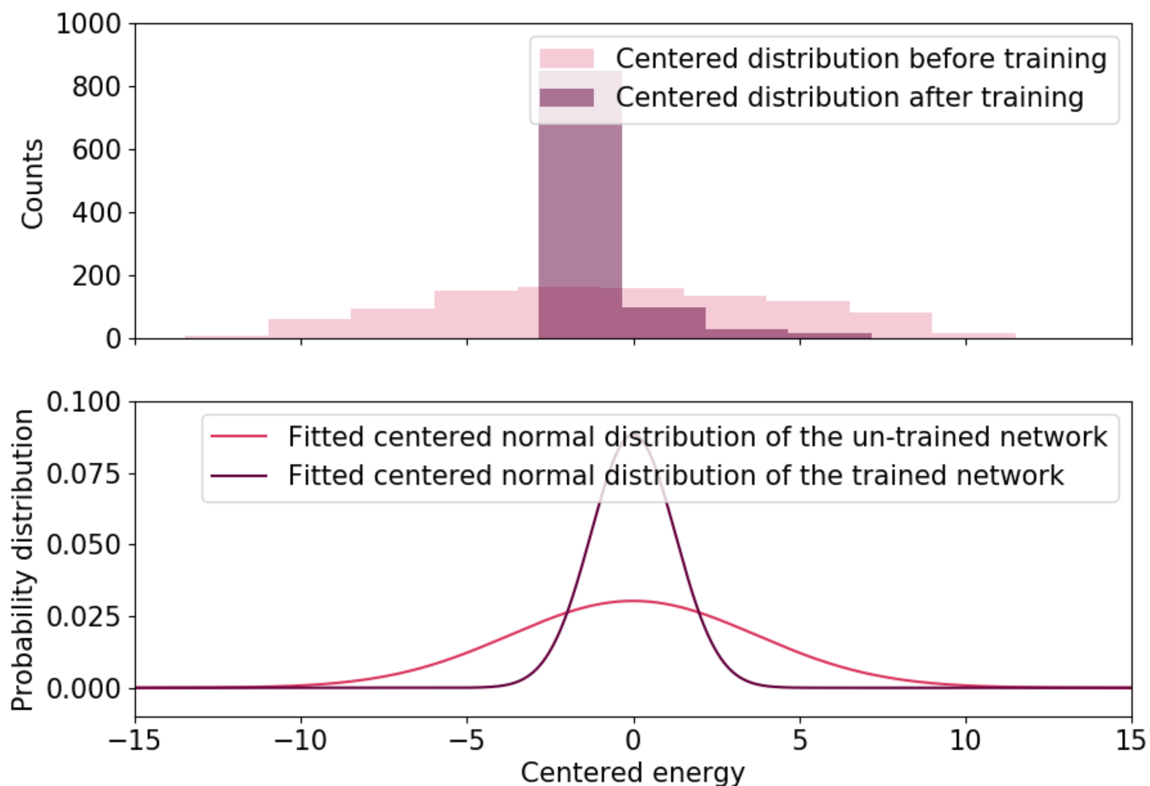
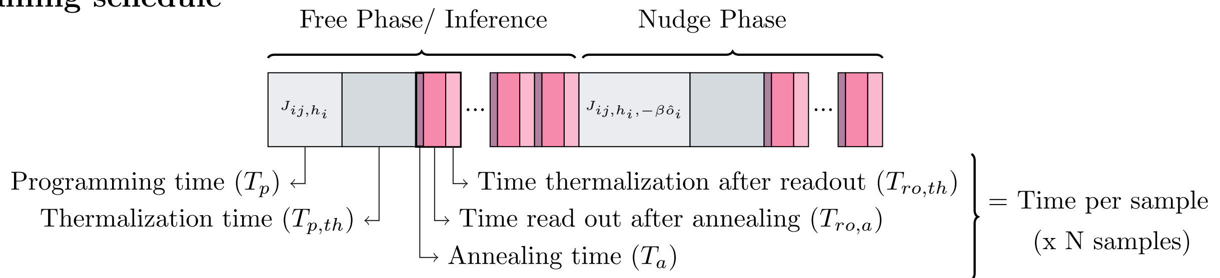


Figure 5.12: Histogram of 2000 samples drawn from the D-Wave IM. 1000 sample are drawn from the IM before it has been trained with EP for the same input and same parameters (data used for the Fig. 5.11a). The distribution is initially quite flat. The other 1000 samples are drawn after the IM has been trained with EP for the same input and same parameters (data used for the Fig. 5.11b). The initial flat distribution has shrunk and the network always predicts the same class for the input.

5.4 . Training a convolutional architecture on the IM: toward layout-guided architectures

In the last section, we have shown that we can train a fully-connected architecture on the D-Wave IM. If we succeed to get state-of-the-art accuracy on MNIST/100 with a training procedure uniquely based on the dynamics of the physical system for both the inference and the error-backpropagation phases, it was at the cost of using multiple spins on the chip to emulate a single artificial neuron which has impaired the size of the architecture we were able to embed on the IM. In this section, we overcome this strong limitation by leveraging the actual chip layout to create spin-efficient embeddings that have almost one spin per corresponding artificial neuron. Here we show we can embed a convolutional architecture on D-Wave that leverages the actual chip layout, and demonstrate that we can train it with EP (see Section 1.1.2 and Section 3.2). This work paves the way to new kinds of artificial neural networks architectures that are guided by the actual layout of the hardware on which it is supposed to be trained instead of having to build a hardware that emulates exactly the architecture of the neural network we want to train.



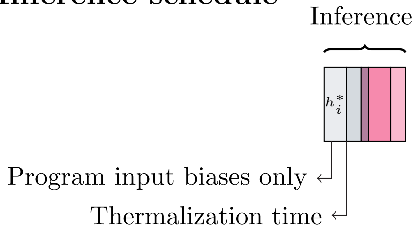


Figure 5.13: Comparison of the time required for doing the training procedure on the chip for one image vs an inference step when the network is trained and does not require multi-sampling anymore.

5.4.1 . Why layout-guided architectures?

Though we have successfully trained a fully-connected architecture on the D-Wave IM in the last sections, we had to use many qubits on the chip per artificial neuron in order to realize the dense connectivity between the two layers on hardware. This is an unfortunate consequence of the layout of the chip that offers poor global connectivity, whether we use the Chimera or the Pegasus layout. This limitation can be overcome by the embedding procedure. However, this procedure is not scalable as we only were able to train an architecture that has 120 hidden neurons despite the fact that the chip has more that 4000 possibly coupled qubits. Moreover, when we want to embed larger or denser architecture on the chip, we have to use longer chains (multiple strongly coupled qubits that represent the same neuron on the hardware) which are prone to more errors, thus altering the training process.

It is interesting to investigate whether we can take advantage of the actual layout of the chip to do computation, which would be more qubit-efficient and thus more scalable.

We show next that we can leverage the fixed layout of the DW2000 D-Wave IM to perform convolutions between small input images and small filters. We first show how we can rethink the cluster of 4x4 coupled qubits as the local multiplication of a subgroup of pixels of an input image with the actual filter of a convolutional operation. Then we describe how we can integrate this building block into a larger architecture with pooling operations and a last classifier layer. We finally demonstrate that this architecture can be trained by EP on the IM on a very simple task.

The idea to realize convolutional operations on the D-Wave IM is not new as [139] already mentioned it. However our work is very different as we really implement a convolution operation with the over-lapping feature when the filter is moved over the input and we train the entire network (the fully-connected classifier included) end-to-end on the chip by EP.

5.4.2 . Convolutional neural networks

Convolutional neural networks (CNNs) are widely used in computer vision (even with the rise of Vision Transformer architecture (ViT) [227], CNNs are still of great interest as combinations of ViTs and CNNs appear to result in higher accuracy than ViTs or CNNs alone [228]).

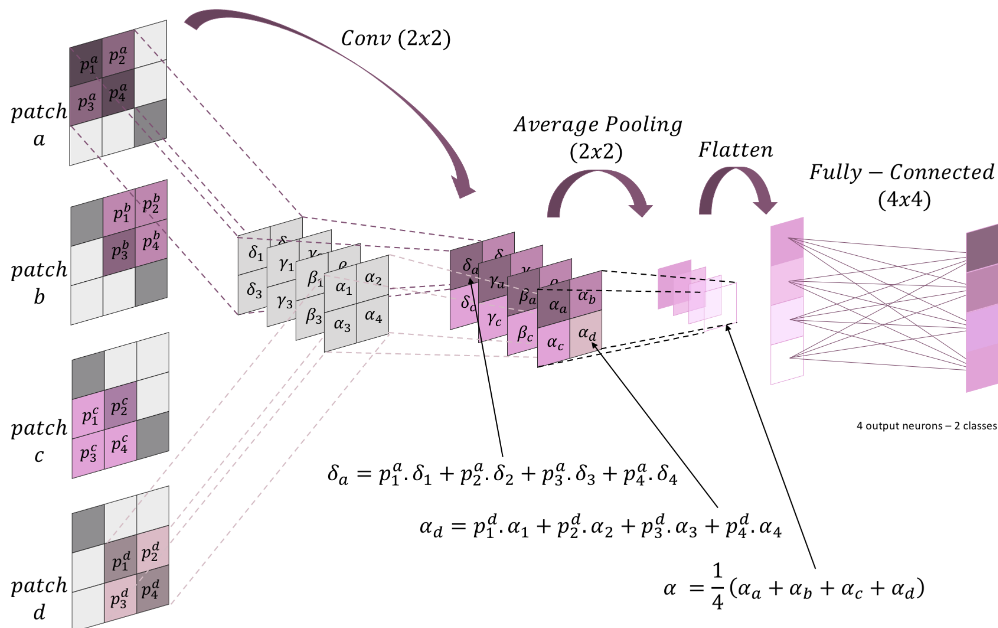


Figure 5.14: Small convolutional neural network with an input image that is a (3x3) pixels diagonal that goes from top left to bottom right and 4 filters that have (2x2) weights so they cover a subset of (2x2) pixels on the image. The filters are sequentially moved on the input image to compute an output tensor whose dimensionality is reduced with the averaged pooling function. Finally, the tensor is flatten and fed to the final classifier that is a fully-connected layer between the flatten tensor and the output layer.

CNNs allow to recognize input images by sequentially filtering the input images with filters that are learnt on the training dataset (see the *Conv (2x2)* operation made in Fig. 5.14 where 4 (2x2) filters $\{\alpha, \beta, \gamma, \delta\}$ are applied to a (3x3) pixels input image). As said in Section 1.1.2, combined with the optimization of a global cost function, this results in convolutional layers that recognize increasingly abstract features and allow to linearly separate the input in order to do classification. Convolutional layers are spaced with pooling layers that reduce the dimensionality of the input at each layer (see the *Average Pooling* operation made in Fig. 5.14). The pooling operation can be either Max-pooling where the output of the operation is the maximum element of the input window or Averaged-pooling where the output is the average of the input window. Finally, the output of the last convolutional layer is fed to a fully-connected classifier that allows to discriminate between the different classes in the dataset and to do classification (see the last *Fully-connected* layer that connects the flatten tensor to the output layer in Fig. 5.14).

5.4.3 . Local clusters of the DW2000 IM as a primitive convolution

Our idea to implement a convolutional neural network on the D-Wave IM is directly inspired by the elementary design of the local clusters of (4x4) coupled qubits that constitute the Chimera layout. Indeed, what we will call "local clusters" from now on, can simply implement the dot product between an input (2x2) matrix and 4 filter matrices (Fig. 5.15). In order to perform this computation, we have to strongly bias the qubits that represent the input data that is static for EP (see Section 3.2). On the DW2000 chip that is based on the Chimera layout, the available range available for the individual biases is $[-4; +4]$. To assign each spin a binary value (it is a limitation

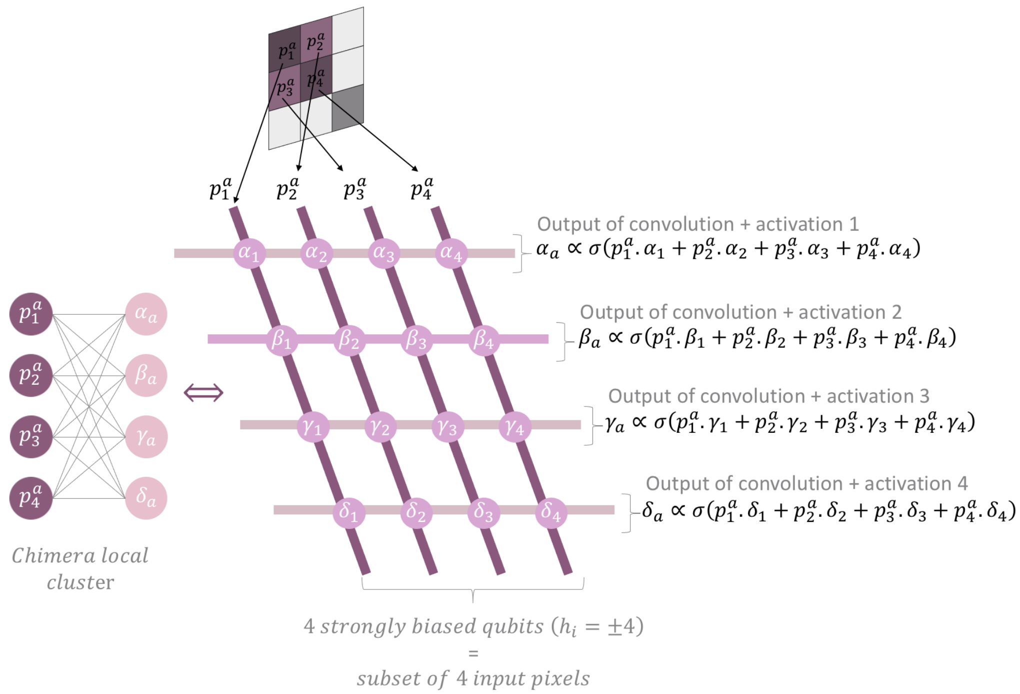


Figure 5.15: 4 elementary convolutional operations between a (2x2) pixels input and 4 (2x2) filters. Left cluster: circles are the qubits and the couplings are represented as straight lines. Right cluster: circles are now where are the couplers on chip and the qubits are both the horizontal and vertical straight lines. The state of each horizontal qubit at the end of the annealing procedure is the equivalent of having applied the binarisation function σ to the sum of weighted inputs that are the states of the strongly biased (*i.e.* the value of the spin is fixed over the annealing procedure) vertical qubits.

of our approach but it will be discussed next), we set the corresponding individual bias according to the following rule: $\sigma_i = 1 \leftrightarrow h_i = -4$ or $\sigma_i = -1 \leftrightarrow h_i = +4$.

This approach is different from the previous section (Section 5.3) where the input were outside the chip and were fed to the chip via input biases applied to the hidden neurons. Despite the fact that binary inputs can be a limitation for datasets that are not natively binary (CIFAR-10, ...) some work has been done about binarizing the input and still achieves high accuracy [175]. Whereas it does reduce the size of the dataset on which we can train the network because the input data has to fit the number of available qubits on the chip, we can, for future development, do as in the previous section and compute a first kind of embedding convolutional layer that is exterior to the system and still learn the weights with the dynamics of the system, which could improve the size of the possible input image.

A convolutional operation is made of repeated dot products between a subset of input pixels and the filters that are moved over the input image. Now that we can set the input pixel on the chip we have to realize the dot product with the filters. This dot product is simply achieved by setting the coupling parameters between the qubits that represent the input pixels and the other qubits in the cluster. The qubits that encode the output of the local dot product behave as a continuous pre-activation during the annealing but eventually collapse on one of the two values possible for the spins. We thus wrote in Fig. 5.15 that the resulting operation is the binarisation function σ applied to the dot product between the input and a specific filter. In this approach, the pooling operation

will thus have to be realized after the activation function, contrary to standard neural networks where the pooling operation is often applied before the activation function, but we suspect this does not impact much the dynamics with binary neurons.

Now we have the building brick of a convolutional neural network: a block where we can perform the local dot product between a (2x2) pixels subset of the input data (that we totally control) and 4 (2x2) filters. Fig 5.15 sums up this building brick.

We position multiple of those clusters on the chip (see Fig. 5.16) in order to perform the entire convolutional operation on the input data with the different convolutional filters. The entire convolutional operation is done in a fully-parallel way which accelerates both the inference and the error-backpropagation phases.

5.4.4 . Integrate the cluster in a primitive convolutional neural network that fits the D-Wave layout

We now discuss how we can connect these bricks together. We detail in this section how we can implement on the chip the Average Pooling function that is used to down sample each intermediate tensor on order to limit the dimension of the final flattened tensor.

The mainstream down-sampling method used in convolutional neural networks is *Max Pooling* that maps a subset of pixels $\{x_i\}_{i=1\ to\ N}$ to a single output value:

$$MaxPool(\{x_i\}_{i=1\ N}) = max(\{x_i\}_{i=1\ to\ N}) \tag{5.13}$$

However, it is not possible to implement that function on the IM as 1) the non-linearity of the operation is not realizable on this chip and 2) the result of the operation can abruptly change from -1 to $+1$ and conversely, which does not fit the quantum annealing procedure that requires smooth changes over time. Nevertheless, we overcome this issue by applying *Average Pooling* instead of *Max Pooling* that also maps a subset of pixels $\{x_i\}_{i=1\ to\ N}$ to a single output value:

$$AvgPool(\{x_i\}_{i=1\ N}) = \frac{1}{N}\sum_{i}^{N}(x_i) \tag{5.14}$$

We immediately see that *Average Pooling* is in fact very easy to implement on the D-Wave IM. We just need to couple the set of qubits $\{x_i\}_{i=1\ to\ N}$ (that carry the result of the convolutional operations for instance) to a qubit that is used to encode the results of the *Average Pooling* operation. However, we see in Fig. 5.16 that we have to use multiple qubits to encode a single output of an *Average Pooling* operation. It is because the output of the convolutional operations that are fed to an Averaged Pooling operation are delocalized on the chip. Similarly to the embedding processus described in Section 5.3.1, we strongly couple with a -1 coupling multiple qubits that are used to emulate the same neuron that encodes the result of the Averaged Pooling operation on-chip. In Fig.5.16, we drew the identical identity couplers with circles or ellipses in which there is a 1 which stands for the identity.

Finally, we feed the classifier with the output of the *Average Pooling* operation. The classifier is simply done between the 4 output neurons of that operation and 4 output neurons that encode 2 classes (2 output neurons/ class). The couplings of the final cluster are set as the weights of the classifier and also trained with EP.

We sum up all these ideas in Fig. 5.16.

Using the core layout of the chip to do special computation better uses the number of qubits on the chip as this network now requires only 1.6 qubits per neuron vs 10 previously with the brute-force approach based on the embedding processus.

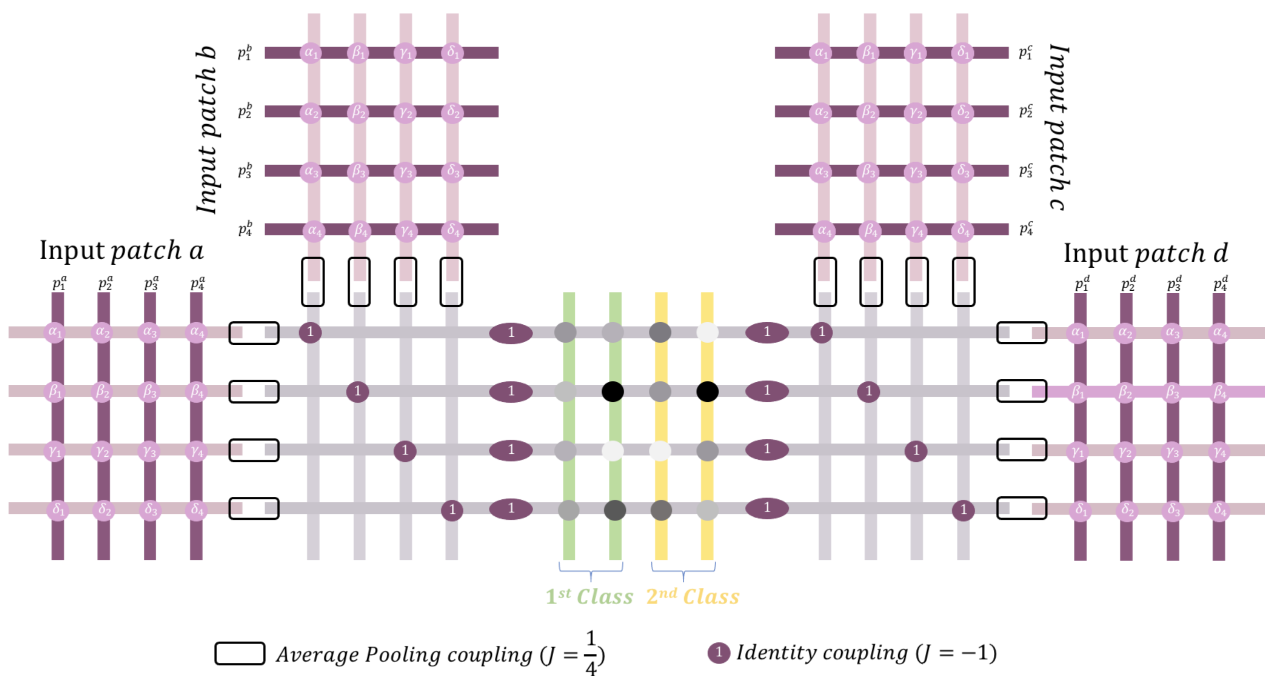


Figure 5.16: The convolutional neural network we trained on the D-Wave IM. There are 4 convolutional clusters that feed the Average Pooling operation. The result of the Average Pooling is fed to the classifier that assigns a class to the input chosen from the two possible classes (green and yellow output qubits - 2 physical qubit/ output class as required by Binary EP). All couplings are symmetric which makes this network completely compatible with an EP training.

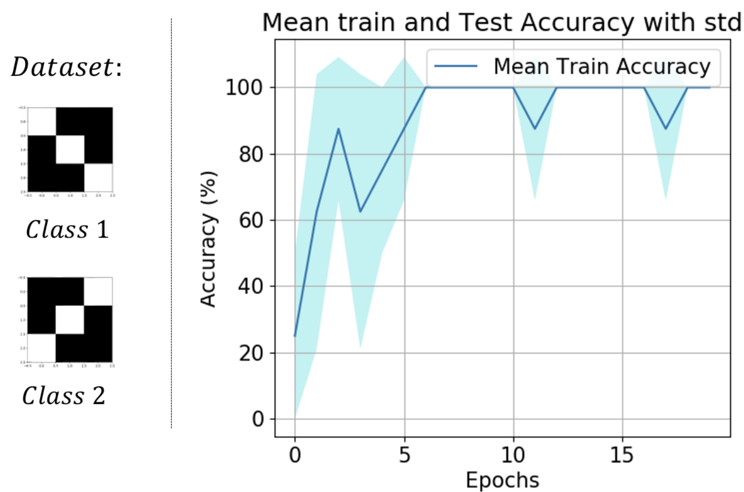


Figure 5.17: Patterns on which we train the convolutional architecture and training accuracy over epochs

Our approach is scalable as we could dispatch many of the elementary convolutional blocks on the chip and merge them with different pooling blocks, allowing to stack layers and thus to perform training on larger images. Also, the next generation of D-Wave chip (architecture Zephyr) will have (8x8) clusters, thus enabling to do convolution with almost (3x3) kernels! In fact we will be able to perform convolutions with (3x3) filters from which one element will be dropped-out. This will allow to have non-symmetric filters that perform much better than the symmetric (2x2) filters we use in this work.

5.4.5 . Results

In order to prove the concept of training a convolutional architecture on-chip with EP, we trained a small architecture on a very small dataset of 2 images of (3x3) pixels. They are orthogonal patterns of diagonals. We chose (3x3) images in order to easily perform convolution with (2x2) kernels as depicted in Fig. 5.16. We only have a training dataset in this case, as a distinct testing dataset cannot be constituted for a small task of this type which consists in binary inputs.

We use the same settings as for training the fully-connected architecture (see Section 5.3.1) where for performing both the inference and the error-backpropagation phases we sample multiple times the IM for selecting the sample that has the lowest energy from all and we use reverse annealing to perform the nudge phase.

We successfully train this convolutional architecture on this very simple dataset: Fig. 5.17. This result demonstrates the relevance of the approach and how we can train end-to-end a convolutional neural network on an IM which better use the layout than the fully-connected architecture we trained in Section 5.3. Indeed, here we use only 1.6 qubit per neuron for the embedding. For the fully-connected architecture we used more than 10 qubits per neuron to allow the dense coupling scheme.

5.5 . Perspectives

Hardware for the IM We used the IM of D-Wave mostly because of the easiness of setting the couplings on the chip that allows to constantly update the parameters on the chip while the training

progresses, as required.

The quantum superiority of this IM is still debated [225], [226] and was not the reason for our choice. In fact, we have shown that Simulated Annealing performs as well as Quantum Annealing on the fully-connected architecture which tells us that either the quantumness of the hardware is overwhelmed by some thermal effects in the hardware or the energy landscape of such a problem (a dense fully-connected problem) is quite smooth which do not favor Quantum Annealing over Simulated Annealing ([229]): Quantum Annealing shows better performances when the energy barrier between two states is high and thin.

As said in introduction, this IM is cooled down in a dilution fridge as the IM needs to be operated at the lowest temperature possible, which is far from being energy-efficient. Other IMs operate at room temperature which improve their energy-efficiency.

Finally, this IM shows limitations especially when it comes to solving dense problems as the connectivity is very limited. [230] proposed a way to overcome this limitation by using a common bus in order to couple multiple quantum oscillators to realize quantum annealing in hardware. This approach can thus solve dense problem as all-to-all connectivity is made possible but still needs to be demonstrated experimentally.

Nevertheless, applying the methods we have developed in this work to other hardware could be of great interest. New hardware could offer new possibilities for layout-guided architectures or allow better couplings between spins.

We see two promising IMs we could use to be of interest to be trained with EP.

The first is the IM made of memristors [127] that allows to couple all spins with all the others and based on a kind of Simulated Annealing procedure that drives the system toward the ground state. With an in-memory computing scheme it results in a very low-power hardware Ising Machine and has already showed great performances on a small dense problem [127].

The second promising hardware is the Coherent Ising Machine [143], [145], [146]. Based on degenerate optical parametric oscillators, it a very fast IM that better solves dense problems [123] than the D-Wave IM and with a better time-to-solution. Also, the procedure used to drive the system to the ground state proceeds by elevating the system from below the energy function which better guarantee to reach the ground state. However, it is not yet clear how we could perform the nudging phase as the system always starts in a state where the oscillators are quiet. But the preliminary results we have shown about how far we can go in the reverse annealing to ensure a successful training makes us think that we could perform two sequential annealings for the free and the nudge phase. The parameters of the Coherent Ising Machine are stored in a side FPGA that are easily modified and updated.

Toward training a pure layout-guided architecture. Taking inspiration from the actual layout of the chip in order to embed an architecture that leverages all the capacity of that same chip is very interesting for many reasons.

First we have shown in Section 5.4 that we can perform more complex operations on the chip by leveraging its intrinsic layout.

Despite the fact that we increase the density of the neurons we can embed on the D-Wave chip, we reduced the 10 qubits per neurons that were initially used on-chip to embed the fully-connected architecture to 1.6 qubit per neuron on average with the convolutional architecture.

We think we can improve even more the use of the qubits on-chip to train an architecture that fully leverages the layout of the chip, thus the term "pure layout-guided architecture".

Our idea is to combine the locally fully-connected clusters of (4x4) qubits on the Chimera layout

with the lateral connections that already connect these clusters to each other (see Fig. 5.18a). With this idea we can design a very deep neural network.

We already did the python script that creates this special embedding that also takes into account the faulty qubits (5 on the chip we used). This embedding is a dictionary that maps a neuron to the corresponding qubit(s) on the chip and conversely.

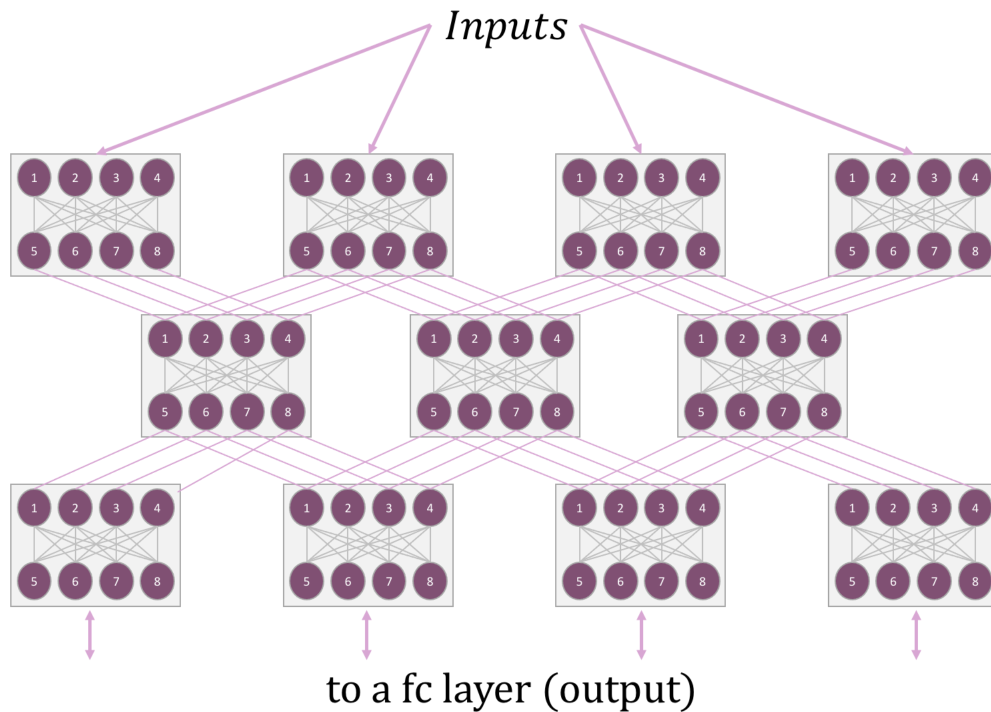
We have been able to demonstrate that we can embed the architecture of the chip but we did not demonstrate any training on the chip neither with Simulated Annealing with this architecture yet. It seems that the couplings have to be scaled in a different range depending on whether the coupling is involved in a cluster or used a lateral connection or is a weight for the final fully connected layer. Indeed, depending on the location of a specific qubit, it receives more or less couplings from neighboring qubits so it can saturate more or less fast, thus the requirement to scale with caution the weights (which we have not been able to do until now). Also, it seems that the error signal that comes from the output layer needs to be greater as it needs to propagates through many more layers. We already tried to reduce the number of classes in the training dataset to increase the number of output neurons that encode a specific class and we implemented an other kind of reverse annealing where we add a plateau when the super-position that is added to the qubits is maximum.

5.6 . Conclusion

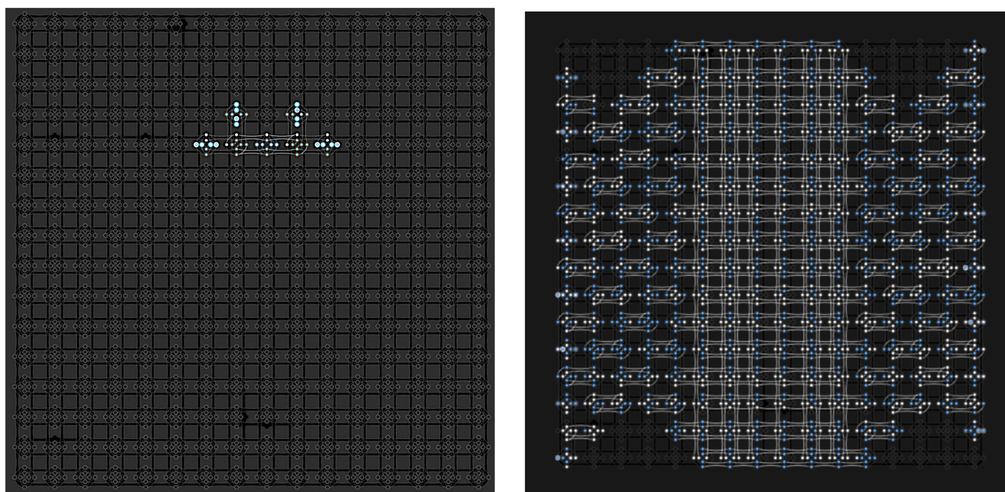
In this chapter we have demonstrated that we could train in a supervised way a physical system through its intrinsic dynamics that minimizes its energy. We have chosen to apply the EP learning framework to a hardware that is designed to minimize a specific energy function: the Ising energy function: an Ising machine.

In principle, Ising machines are ideal energy-based systems to be trained as neural network, by EP for instance. However there existed multiple challenges that we had to solve before being able to trained the D-Wave IM with EP. We have shown that we can perform both the free and the nudge phases of EP thanks to the Quantum Annealing procedure (and Reverse Quantum Annealing) that allows spins to reach the ground state of the Ising energy through the superposition of states and tunneling similarly to thermal effects that are used for Simulated Annealing.

We have demonstrated that, by using jointly Quantum Annealing and EP, we have been able to train a fully-connected architecture on MNIST/100 and to obtain 100% train accuracy and 87% test accuracy with only 120 hidden neurons. To benchmark the performance of our results we conducted software simulations with the same network (binary activations and real-value synaptic weights) but supported by either a deterministic gradient dynamics or with Simulated Annealing. We have shown that the performance of the hardware Ising Machine is equal to that we got with the software simulations, which confirms the relevance of our approach. Furthermore, we show with Simulated Annealing that the test accuracy we could get with the IM would increase if we increase the size of the training dataset. We also show that there is hierarchy of features detected on the input that emerges, which has never been demonstrated on an IM before because of the algorithms used until now. We investigate the features that the network has learnt on the input data by reconstructing the input that maximizes the activity of specific neurons in the network. Finally, we have shown that, despite the fact that the D-Wave IM behaves stochastically at the beginning of the training process, it ends up in a deterministic regime where it always predicts the same output for a given input. This result could be of great interest for the use of IM for combinatorial optimization problems as we show we can accurately get the solution of a problem by first training the IM to find the solution by just looking at labeled training data.



(a) Schematic of the layout-guided architecture



(b) Comparison of: Left: the embedded convolutional neural network we trained in Section 5.4 and Right: the embedded layout-guided architecture.

Figure 5.18: Description of the layout-guided architecture idea.

We have also explored a second approach to train a neural network on the D-Wave IM. For training a fully-connected architecture on the IM, we had to use an embedding procedure that restrains the number of neurons that can be emulated by the spins on the chip as we use multiple spins per neuron to allow the dense connectivity of the neural architecture, while the chip offers a sparse coupling scheme between qubits. To overcome this issue, we took inspiration from the actual layout of the chip and show that we can perform convolutions with the fixed layout. We demonstrate that we can train a small convolutional network on a D-Wave IM. This approach could inspire other approach were the architecture of the hardware is a bottleneck for embedding standard artificial neural networks.

This work will be the subject of a publication "Supervised learning in an Ising machine with equilibrium propagation, Laydevant et al." in a peer-reviewed journal and is currently being written.

Chapter 6

Conclusion and perspectives

6.1 . Conclusion

The main reason that motivated this thesis work that is the energy inefficiency of current standard digital hardware for training artificial neural networks that are the pillars of deep learning [1]. Indeed, the digital processors, based on the Von Neumann architecture [5] where the memory and the processing unit are physically separated, are not optimized to process the large amount of parameters that constitute the recent deep learning models. Hence, training artificial neural networks that are becoming larger and larger with this kind of processor results in an increasing electricity and carbon bill as the energetic cost of moving data largely overwhelms the cost of doing an actual operation with the data [7].

In introduction we have reviewed some standard methods that are currently developed in order to reduce that bill. These methods either aim at reducing both the computational complexity and the size of the models to train or aim at building hardware where the memory is brought closer to the processing units.

However, it has been shown that the potential energy efficiency of these approaches relying on standard digital hardware is still orders of magnitudes behind that could be reached with unconventional emerging devices [231]. Using the physical properties of some devices to compute is very promising as we leverage intrinsic properties to perform the computation that is artificially done with digital processors through multiple operations.

There exists different ways to parametrize these physical systems in order to drive them to perform the parametrized non-linear transformation that a neural network does. One promising method is to introduce this parametrization in the energy function of physical systems [9]–[11], [110], [112], in order to tune the minima of that energy function. Then we can leverage the property of physical systems to evolve towards states that minimize this energy to perform computations.

Equilibrium Propagation [14] is a learning algorithm that leverages the parametrization in the energy to perform supervised learning. EP is very promising to train low-power hardware implementations as it proposes a local learning rule to minimize a global objective. Yet, no large-scale training of a hardware by EP has been demonstrated before this work. This was mainly due to the fact that the original EP algorithm is designed to work on dynamical systems with continual, smooth variations of the states of neurons and synapses. In other words, in its original form, EP is extremely suited to implementations with emerging nano-devices which, unfortunately, remain experimental and present high intra and device-to-device variability that alter the training process. On the other, the original EP algorithm did not seem suited to systems with binary synapses, binary neurons, such as for example Ising machines or digital hardware.

In Chapter 4 we have shown that we can in fact train dynamical physical systems which have either binary synapses and/ or a binary activation function. A first challenge was to optimize the binary parameters, which is not trivial as we can not apply small steps in the gradient direction in order to perform a stochastic gradient descent [40] because of the non-continuity of the two-levels synaptics devices. Nevertheless, we merged EP and BOP [176] and successfully trained fully-connected and convolutional architectures on MNIST and CIFAR-10. We have obtained 2% of test error on MNIST

with the fully-connected architectures and 0.8% with the convolutional architecture. We got 15.66% test error on CIFAR-10 which is 1.9% below the best accuracy reported with EP. We also studied the binarization of the activation function of the neurons that constitute the neural network. We solved the issue of the vanishing gradient that was caused by the binarization of the neurons by expanding the output layer and successfully trained fully-connected and convolutional architectures which have both binary weights and activations on MNIST and respectively achieved 3% and 1.1% of test error. This work opens great perspectives for hardware implementations that we can train with EP. Future binary synapses can be implemented by exploiting the ON and OFF states of emerging nano-devices such as MRAMs (magnetic tunnel junctions) or RRAM s(memristors), which are less prone to variability than their continuous variations. We have proposed an architecture to implement in hardware BOP, using capacitors to store the momentum that is used for the binary optimization, and memristors that are used to fine-tune the time-constant of the momentum. The fully binarized version of EP also opens a possibility to implement a binary dynamical system with standard digital CMOS technology and we have shown that for specific values of the parameters of the dynamics, we can leverage elementary built-in CMOS functions to implement the binary dynamics. This can allow to conceive a digital chip especially designed to train BNNs on the edge.

Following the work on binary neural networks trained by EP of Chapter 4, we have applied in Chapter 5 some of the methods we have developed to a binary physical system of coupled spins that is intrinsically energy-based. We have chosen to use an Ising Machine designed by D'Wave [134] which drives the system of spins toward the ground state of the Ising energy function that describes the system through the procedure of quantum annealing. We successfully achieve to train with EP a fully-connected neural network on the IM on MNIST/100 and achieved to get 87% testing accuracy without any backpropagation, simply by measuring the equilibrium states that the IM successfully reaches for performing the two phases, inference and nudging, required for EP. The results we got on the IM are equivalent to those we obtained with software-based simulation either with a deterministic dynamics or with Simulated Annealing. We then leverage the layout of the IM to perform more complex computations such as convolutions on-chip. We show that we can train a small convolutional neural network on a small dataset with 100% training accuracy demonstrating the relevance of our approach. This way we better use the number of accessible spins on the chip as we get rid of the embedding procedure we had to employ for embedding the fully-connected architecture on the chip. This approach opens new possibilities to train neural network on hardware which connectivity is constrained and thus can not embed standard architectures directly.

6.2 . Perspectives

All along the work I have done for this thesis, I have developed my knowledge of the current state of the field of neuromorphic computing, and my vision about what I think will be crucial in order to drive its progress in the years to come.

First, I would like to highlight the recent progresses made with standard digital hardware that we have reviewed in introduction (Section 1.3.3): TPUs, custom implementations with FPGAs, ASICs, etc where most effort is done to shrink the physical gap between the memory and the computing units. Combined with the improvements of the software where the computational requirements for training and running the neural networks are constantly improving, these works have shown impressive energy savings while still being able to compete on complex tasks with standard processors. This has particularly struck me as I was writing this manuscript. Still, as emphasized in the introduction, most of these works are focused on inference. A first reason is they are funded by companies that can

afford the computational power for training with large data-centers and that deploy models on edge devices where they are used by the end-user. Another reason is the current trend of large models being trained only once and then made open-source for the community: since most of the energy spent during the life-time of the model is for inference time, it is better to improve the inference hardware.

Using unconventional emerging devices can fill the gap of learning-capable low-power hardware for edge applications.

However as we have seen, it is still not straightforward how to handle these devices in order to compute. Their still experimental nature makes them variable so the success of the training based on these devices is far from being guaranteed. It also took a long time to develop algorithms that could leverage their striking properties to perform the non-linear transformations of deep learning that could also allow these unconventional neural networks to reach the performance of the literature. Most of algorithms that were grounded in physics did not optimize a global cost function and that has capped the performance of these networks for a long time. Equilibrium Propagation has been a game-changer when published as it brings theoretical guarantees that physical systems could be trained in a supervised way using only the tendency of these systems to evolve toward states of minimum energy. And this changes everything.

I think it will be important in the future to have this kind of guarantee before applying such a learning algorithm to a physical system.

In this thesis we have worked with Equilibrium Propagation because it leverages the dynamics of a physical system to do supervised learning with local learning rules, which is extremely attractive for training hardware implementations. However, there exist other learning algorithms that either address the problem of memory overhead caused by backpropagation and are of great interest for training hardware implementations or that are grounded in physics like EP. For the first class we can cite Direct Feedback Aligment [64] that is already used by the startup LightOn to train neural networks with an optical hardware that performs random projections [232]. There is also Target Propagation which avoids the weight transport problem [66]. Both learning algorithms perform a gradient descent on a global cost function which guarantees the scalability of the methods in order to reach high accuracy on hard tasks. Learning algorithms that are grounded in physics are now booming [233]–[236] and could also be of great interest for computing with hardware devices by leveraging unconventional properties.

In order to design hardware neural networks we need hardware building blocks such as the synapses and the neurons. The works that aim at characterizing both building blocks are great as once characterized, their properties can be integrated in a learning framework so we know the performance the system can theoretically reach.

The recent work about training deep physical neural networks with backpropagation [109] is very interesting as it digresses from the standard operations used in neural networks and it simply uses the non-linear transformation the physical system performs as the new operation, without trying to emulate the standard operation with the physical system as it is usually done with hardware implementations. This could lead to orders of magnitude in energy savings but the transformation, despite being well characterized with physical models and experimental data, still lacks a machine learning interpretation in terms of what the physical system really compute on input data, if the global optimization with backpropagation allows to create a hierarchy of features detected at the input, ... Can this kind of physical network detect geometrical features on the input data? It does not perform a convolution with geometrical filters so the question is opened and might be of great interest in the years to come.

Concerning hardware implementations with low-power emerging devices, we have already showed that binarizing their state for an actual hardware implementation could be of great interest because it allows to overcome their intrinsic variability. I strongly believe that we should binarize memristors in order to demonstrate on-chip learning with EP with such devices. But we also need to solve to problem of symmetric synapses as it is required by EP: should we double the number of components (and double the source of variability and the energy consumption) or should we try to arrange these devices in a way that integrates them as symmetric components?

Additionally, we focused all the thesis on supervised learning as it is known to perform the best on training data. But, we also focused the thesis on potential low-power hardware implementations for doing training on edge devices that have little energy and computational budgets. However, supervised learning requires labeled data, that necessitate a human intervention. And this could be incompatible with edge computing. Thousands of smart sensors can not be individually accessed by a human in order to label their individual training data because they may be off-network and this would be a waste of time. Then, it can be interesting to investigate whether it can be possible to do self-supervised or unsupervised learning with EP. Self-supervised learning is in principle feasible as we can define a cost function that EP can minimize through gradient descent. The success of doing unsupervised learning with EP lies in the possibility to design a cost function that EP can minimizes and that improves the performance of the network after a lot of data is shown.

By using an Ising Machine to train a neural network, we realized the original idea of Hopfield that was to use a system of coupled spins to compute. Despite the fact that we have used a hardware that is supposed to exhibit a quantum superiority compared to classical procedures such as Simulated Annealing, we did not see any improvements compared to the classical simulations. As already emphasized in previous studies [225], [226], thermal effects due to the non-zero operating temperature add decoherence to the qubits which deteriorates their quantum nature. Investigating whether quantum effects can help at finding better minimum of the Ising energy function will be an interesting path to explore in the future, especially with other IMs that are based on potentially quantum systems (the Coherent Ising Machine [143], [145] for instance).

Moreover, the thermal effects that are present in the D'Wave IM also make the hardware stochastic which can alter the possibility to reach the ground state of a given Ising problem. We can overcome this issue by sampling multiple times the system but it can be lengthy. Furthermore, in order to solve combinatorial optimization problems (the main use of IMs at the moment), one has to map the parameters of a given problem to the Ising energy function, which can be quite complex depending on the initial problem. Here we propose a new scheme to solve such problems where the parameters of the problem to be solved (*i.e.* sampled on the IM) are discovered and learnt with EP on a training labeled dataset. This methodology could improve the solution of such combinatorial optimization problems as we have shown in Chapter 5 that the supervised training procedure of the IM makes the D'Wave IM a deterministic hardware that solve our problem, *i.e.* predict the class of a given input data. Deep learning for apprehending quantum systems has already been demonstrated in [237] where the authors used the attention mechanism from the transformer architecture [49] in order to reconstruct the state of a noisy quantum system. The joint use of deep learning and quantum could lead to striking results for both disciplines in the following years.

In conclusion, the convergence between the development of highly parametrizable systems for computing (quantum and neuromorphic) and the development of physics-grounded algorithms promises exciting developments for ultra-low power AI in the future.

Bibliography

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *Nature*, vol. 521, no. 7553, pp. 436–444, 2015. DOI: 10.1038/nature14539. [Online]. Available: https://doi.org/10.1038/nature14539.

[2] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen, *Hierarchical text-conditional image generation with clip latents*, 2022. DOI: 10.48550/ARXIV.2204.06125. [Online]. Available: https://arxiv.org/abs/2204.06125.

[3] T. Brown, B. Mann, N. Ryder, *et al.*, "Language models are few-shot learners", in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[4] J. Hoffmann, S. Borgeaud, A. Mensch, *et al.*, *Training compute-optimal large language models*, 2022. DOI: 10.48550/ARXIV.2203.15556. [Online]. Available: https://arxiv.org/abs/2203.15556.

[5] *First draft of a report on the edvac, john von neumann*, https://web.archive.org/web/20130314123032/http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf, Accessed: 2022-08-23.

[6] J. Backus, "Can programming be liberated from the von Neumann style? a functional style and its algebra of programs", *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, Aug. 1978, ISSN: 0001-0782. DOI: 10.1145/359576.359579. [Online]. Available: https://doi.org/10.1145/359576.359579 (visited on 08/23/2022).

[7] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks", in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16, Seoul, Republic of Korea: IEEE Press, 2016, pp. 367–379, ISBN: 9781467389471. DOI: 10.1109/ISCA.2016.40. [Online]. Available: https://doi.org/10.1109/ISCA.2016.40.

[8] W. Zhang, B. Gao, J. Tang, *et al.*, "Neuro-inspired computing chips", en, *Nature Electronics*, vol. 3, no. 7, pp. 371–382, Jul. 2020, Number: 7 Publisher: Nature Publishing Group, ISSN: 2520-1131. DOI: 10.1038/s41928-020-0435-7. (visited on 08/19/2020).

[9] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities", *Proceedings of the National Academy of Sciences*, vol. 79, no. 8, pp. 2554–2558, 1982, ISSN: 0027-8424. DOI: 10.1073/pnas.79.8.2554.

[10] B. Kosko, "Adaptive bidirectional associative memories", EN, *Applied Optics*, vol. 26, no. 23, pp. 4947–4960, Dec. 1987, Publisher: Optica Publishing Group, ISSN: 2155-3165. DOI: 10.1364/AO.26.004947. [Online]. Available: https://opg.optica.org/ao/abstract.cfm?uri=ao-26-23-4947 (visited on 08/24/2022).

[11] G. E. Hinton and T. J. Sejnowski, "Optimal Perceptual Inference", en, p. 6,

[12] G.-q. Bi and M.-m. Poo, "Synaptic modifications in cultured hippocampal neurons: Dependence on spike timing, synaptic strength, and postsynaptic cell type", *The Journal of Neuroscience*, vol. 18, no. 24, pp. 10 464–10 472, Dec. 1998. DOI: 10.1523/jneurosci.18-24-10464.1998. [Online]. Available: https://doi.org/10.1523/jneurosci.18-24-10464.1998.

[13] O. Bichler, D. Querlioz, S. J. Thorpe, J.-P. Bourgoin, and C. Gamrat, "Extraction of temporally correlated features from dynamic vision sensors with spike-timing-dependent plasticity", en, *Neural Networks*, vol. 32, pp. 339–348, Aug. 2012, ISSN: 08936080. DOI: 10.1016/j.neunet.2012.02.022. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0893608012000512 (visited on 06/23/2021).

[14] B. Scellier and Y. Bengio, "Equilibrium Propagation: Bridging the Gap between Energy-Based Models and Backpropagation", *Frontiers in Computational Neuroscience*, vol. 11, 2017, ISSN: 1662-5188. [Online]. Available: https://www.frontiersin.org/article/10.3389/fncom.2017.00024 (visited on 06/21/2022).

[15] D. Querlioz, O. Bichler, P. Dollfus, and C. Gamrat, "Immunity to Device Variations in a Spiking Neural Network With Memristive Nanodevices", *IEEE Transactions on Nanotechnology*, vol. 12, no. 3, pp. 288–295, May 2013, Publisher: Institute of Electrical and Electronics Engineers. DOI: 10.1109/TNANO.2013.2250995. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01826840 (visited on 02/28/2022).

[16] C. Baeumer, R. Valenta, C. Schmitz, *et al.*, "Subfilamentary networks cause cycle-to-cycle variability in memristive devices", *ACS Nano*, vol. 11, no. 7, pp. 6921–6929, Jul. 25, 2017. DOI: 10.1021/acsnano.7b02113. [Online]. Available: https://doi.org/10.1021/acsnano.7b02113.

[17] J. Laydevant, M. Ernoult, D. Querlioz, and J. Grollier, "Training dynamical binary neural networks with equilibrium propagation", in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, Jun. 2021, pp. 4640–4649.

[18] M. Courbariaux, Y. Bengio, and J.-P. David, *BinaryConnect: Training Deep Neural Networks with binary weights during propagations*, Number: arXiv:1511.00363 arXiv:1511.00363 [cs], Apr. 2016. [Online]. Available: http://arxiv.org/abs/1511.00363 (visited on 06/22/2022).

[19] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized Neural Networks", in *Advances in Neural Information Processing Systems*, vol. 29, Curran Associates, Inc., 2016. [Online]. Available: https://papers.nips.cc/paper/2016/hash/d8330f857a17c53d217014ee776bfd50-Abstract.html (visited on 06/23/2022).

[20] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation", en, CALIFORNIA UNIV SAN DIEGO LA JOLLA INST FOR COGNITIVE SCIENCE, Tech. Rep., Sep. 1985, Section: Technical Reports. [Online]. Available: https://apps.dtic.mil/sti/citations/ADA164453 (visited on 08/23/2022).

[21] G. Piccinini, *Physical Computation: A Mechanistic Account*. Oxford University Press UK, 2015.

[22] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity", en, *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, Dec. 1943, ISSN: 1522-9602. DOI: 10.1007/BF02478259. [Online]. Available: https://doi.org/10.1007/BF02478259 (visited on 08/11/2022).

[23] C. Touzet, "INTRODUCTION AU CONNEXIONNISME", fr, p. 129,

[24] I. T. Brain and F. Rosenblatt, *The Perceptron: A Probabilistic Model for Information Storage and Organization*.

[25] D. O. Hebb, *The Organization Of Behavior A Neuropsychological Theory*, eng. 1949. [Online]. Available: http://archive.org/details/in.ernet.dli.2015.168156 (visited on 08/24/2022).

[26] M. Minsky and S. A. Papert, *Perceptrons: An Introduction to Computational Geometry*, en. Sep. 2017. DOI: 10.7551/mitpress/11301.001.0001. [Online]. Available: https://direct.mit.edu/books/book/3132/PerceptronsAn-Introduction-to-Computational (visited on 08/23/2022).

[27] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors", en, *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986, ISSN: 0028-0836, 1476-4687. DOI: 10.1038/323533a0. [Online]. Available: http://www.nature.com/articles/323533a0 (visited on 09/01/2022).

[28] Y. LeCun, B. Boser, J. S. Denker, *et al.*, "Backpropagation applied to handwritten zip code recognition", *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989. DOI: 10.1162/neco.1989.1.4.541.

[29] *Graphic explantion of the convolutional operation.* https://mlnotebook.github.io/img/CNN/convSobel.gif, Accessed: 2022-08-31.

[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", in *Advances in Neural Information Processing Systems*, vol. 25, Curran Associates, Inc., 2012. [Online]. Available: https://papers.nips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html (visited on 07/27/2022).

[31] M. D. Zeiler and R. Fergus, "Visualizing and Understanding Convolutional Networks", en, in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds., vol. 8689, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2014, pp. 818–833. DOI: 10.1007/978-3-319-10590-1_53. [Online]. Available: http://link.springer.com/10.1007/978-3-319-10590-1_53 (visited on 08/11/2022).

[32] G. E. Hinton, "Training Products of Experts by Minimizing Contrastive Divergence", en, *Neural Computation*, vol. 14, no. 8, pp. 1771–1800, Aug. 2002, ISSN: 0899-7667, 1530-888X. DOI: 10.1162/089976602760128018. [Online]. Available: https://direct.mit.edu/neco/article/14/8/1771-1800/6687 (visited on 09/01/2022).

[33] A. Krizhevsky, "Convolutional Deep Belief Networks on CIFAR-10", en, p. 9,

[34] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition", en, in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA: IEEE, Jun. 2016, pp. 770–778, ISBN: 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.90. [Online]. Available: http://ieeexplore.ieee.org/document/7780459/ (visited on 08/11/2022).

[35] M. A. Cauchy, "Méthode générale pour la résolution des systèmes d'équations simultanées", en, p. 3,

[36] H. B. Curry, "The method of steepest descent for non-linear minimization problems", en, *Quarterly of Applied Mathematics*, vol. 2, no. 3, pp. 258–261, 1944, ISSN: 0033-569X, 1552-4485. DOI: 10.1090/qam/10667. [Online]. Available: https://www.ams.org/qam/1944-02-03/S0033-569X-1944-10667-3/ (visited on 08/31/2022).

[37] *Andrew ng stanford lecture on deep learning*, https://cs229.stanford.edu/notes/cs229-notes1.pdf, Accessed: 2022-09-18.

[38] M. Jabri and B. Flower, "Weight perturbation: An optimal architecture and learning technique for analog vlsi feedforward and recurrent multilayer networks", *IEEE Transactions on Neural Networks*, vol. 3, no. 1, pp. 154–157, 1992. DOI: 10.1109/72.105429.

[39] T. P. Lillicrap, A. Santoro, L. Marris, C. J. Akerman, and G. Hinton, "Backpropagation and the brain", *Nature Reviews Neuroscience*, vol. 21, no. 6, pp. 335–346, 2020. DOI: 10.1038/s41583-020-0277-3. [Online]. Available: https://doi.org/10.1038/s41583-020-0277-3.

[40] H. Robbins and S. Monro, "A Stochastic Approximation Method", *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, 1951. DOI: 10.1214/aoms/1177729586. [Online]. Available: https://doi.org/10.1214/aoms/1177729586.

[41] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks", *Journal of Machine Learning Research - Proceedings Track*, vol. 9, pp. 249–256, Jan. 2010.

[42] K. He, X. Zhang, S. Ren, and J. Sun, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, Number: arXiv:1502.01852 arXiv:1502.01852 [cs], Feb. 2015. [Online]. Available: http://arxiv.org/abs/1502.01852 (visited on 07/15/2022).

[43] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun, *The Loss Surfaces of Multilayer Networks*, en, Number: arXiv:1412.0233 arXiv:1412.0233 [cs], Jan. 2015. [Online]. Available: http://arxiv.org/abs/1412.0233 (visited on 06/02/2022).

[44] Y. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio, *Identifying and attacking the saddle point problem in high-dimensional non-convex optimization*, arXiv:1406.2572 [cs, math, stat], Jun. 2014. [Online]. Available: http://arxiv.org/abs/1406.2572 (visited on 08/16/2022).

[45] C. Jin, P. Netrapalli, R. Ge, S. M. Kakade, and M. I. Jordan, "On nonconvex optimization for machine learning", *Journal of the ACM*, vol. 68, no. 2, pp. 1–29, Apr. 2021. DOI: 10.1145/3418526. [Online]. Available: https://doi.org/10.1145/3418526.

[46] D. P. Kingma and J. Ba, *Adam: A Method for Stochastic Optimization*, Number: arXiv:1412.6980 arXiv:1412.6980 [cs], Jan. 2017. DOI: 10.48550/arXiv.1412.6980. [Online]. Available: http://arxiv.org/abs/1412.6980 (visited on 06/23/2022).

[47] X. Zhai, A. Kolesnikov, N. Houlsby, and L. Beyer, "Scaling Vision Transformers", en, p. 10,

[48] J. Kaplan, S. McCandlish, T. Henighan, *et al.*, *Scaling Laws for Neural Language Models*, arXiv:2001.08361 [cs, stat], Jan. 2020. [Online]. Available: http://arxiv.org/abs/2001.08361 (visited on 08/23/2022).

[49] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is All you Need", in *Advances in Neural Information Processing Systems*, vol. 30, Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html (visited on 08/23/2022).

[50] *Introducing the world's largest open multilingual language model: Bloom*, https://bigscience.huggingface.co/blog/bloom, Accessed: 2022-08-04.

[51] *Carbon intensity of electricity in france en europe.* https://www.greenit.fr/2009/04/24/combien-de-co2-degage-un-1-kwh-electrique/, Accessed: 2022-08-31.

[52] *Estimation de l'empreinte carbone de 1995 à 2020*, https://www.statistiques.developpement-durable.gouv.fr/estimation-de-lempreinte-carbone-de-1995-202, Accessed: 2022-09-01.

[53] D. Patterson, J. Gonzalez, U. Hölzle, *et al.*, *The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink*, Number: arXiv:2204.05149 arXiv:2204.05149 [cs], Apr. 2022. [Online]. Available: http://arxiv.org/abs/2204.05149 (visited on 06/20/2022).

[54] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, "Green ai", *Commun. ACM*, vol. 63, no. 12, pp. 54–63, Nov. 2020, ISSN: 0001-0782. DOI: 10.1145/3381831.

[55] B. Zoph and Q. V. Le, *Neural Architecture Search with Reinforcement Learning*, arXiv:1611.01578 [cs], Feb. 2017. [Online]. Available: http://arxiv.org/abs/1611.01578 (visited on 08/22/2022).

[56] S. Zhang, S. Roller, N. Goyal, *et al.*, *Opt: Open pre-trained transformer language models*, 2022. DOI: 10.48550/ARXIV.2205.01068. [Online]. Available: https://arxiv.org/abs/2205.01068.

[57] W. Zhang, B. Gao, J. Tang, *et al.*, "Neuro-inspired computing chips", *Nature Electronics*, vol. 3, no. 7, pp. 371–382, 2020. DOI: 10.1038/s41928-020-0435-7. [Online]. Available: https://doi.org/10.1038/s41928-020-0435-7.

[58] S. A. Janowsky, "Pruning versus clipping in neural networks", *Physical Review A*, vol. 39, no. 12, pp. 6600–6603, Jun. 1989, Publisher: American Physical Society. DOI: 10.1103/PhysRevA.39.6600. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.39.6600 (visited on 08/22/2022).

[59] J. Frankle and M. Carbin, "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks", *arXiv:1803.03635 [cs]*, Mar. 2019, arXiv: 1803.03635. [Online]. Available: http://arxiv.org/abs/1803.03635 (visited on 03/30/2021).

[60] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training Deep Neural Networks with binary weights during propagations", en, *Advances in Neural Information Processing Systems*, vol. 28, pp. 3123–3131, 2015.

[61] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1", *arXiv:1602.02830 [cs]*, Mar. 2016, arXiv: 1602.02830. [Online]. Available: http://arxiv.org/abs/1602.02830 (visited on 03/23/2021).

[62] I. Hubara, E. Hoffer, and D. Soudry, "QUANTIZED BACK-PROPAGATION: TRAINING BINA- RIZED NEURAL NETWORKS WITH QUANTIZED GRA- DIENTS", en, p. 4, 2018.

[63] A. G. Howard, M. Zhu, B. Chen, *et al.*, *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*, Number: arXiv:1704.04861 arXiv:1704.04861 [cs], Apr. 2017. [Online]. Available: http://arxiv.org/abs/1704.04861 (visited on 08/05/2022).

[64] A. Nøkland, *Direct Feedback Alignment Provides Learning in Deep Neural Networks*, arXiv:1609.01596 [cs, stat], Dec. 2016. DOI: 10.48550/arXiv.1609.01596. [Online]. Available: http://arxiv.org/abs/1609.01596 (visited on 08/23/2022).

[65] D.-H. Lee, S. Zhang, A. Fischer, and Y. Bengio, *Difference Target Propagation*, arXiv:1412.7525 [cs], Nov. 2015. [Online]. Available: http://arxiv.org/abs/1412.7525 (visited on 08/23/2022).

[66] M. Ernoult, F. Normandin, A. Moudgil, *et al.*, "Towards Scaling Difference Target Propagation by Learning Backprop Targets", *arXiv:2201.13415 [cs]*, Jan. 2022, arXiv: 2201.13415. [Online]. Available: http://arxiv.org/abs/2201.13415 (visited on 02/07/2022).

[67] L. Kusmierz, T. Isomura, and T. Toyoizumi, "Learning with three factors: Modulating Hebbian plasticity with errors", en, *Current Opinion in Neurobiology*, Computational Neuroscience, vol. 46, pp. 170–177, Oct. 2017, ISSN: 0959-4388. DOI: 10.1016/j.conb.2017.08.020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0959438817300612 (visited on 08/23/2022).

[68] *Google supercharges machine learning tasks with tpu custom chip*, https://cloud.google.com/blog/products/ai-machine-learning/google-supercharges-machine-learning-tasks-with-custom-chip, Accessed: 2022-08-05.

[69] *Build and train machine learning models on our new google cloud tpus*, https://blog.google/products/google-cloud/google-cloud-offer-tpus-machine-learning/, Accessed: 2022-08-05.

[70] Y. E. Wang, G.-Y. Wei, and D. Brooks, *Benchmarking TPU, GPU, and CPU Platforms for Deep Learning*, arXiv:1907.10701 [cs, stat], Oct. 2019. [Online]. Available: http://arxiv.org/abs/1907.10701 (visited on 08/23/2022).

[71] *The future is here: Iphone x*, https://www.apple.com/newsroom/2017/09/the-future-is-here-iphone-x/, Accessed: 2022-08-05.

[72] *Fsd chip - tesla*, https://en.wikichip.org/wiki/tesla_(car_company)/fsd_chip, Accessed: 2022-08-05.

[73] C. Mead, "Neuromorphic Electronic Systems", en, *PROCEEDINGS OF THE IEEE*, vol. 78, p. 8, 1990.

[74] F. Akopyan, J. Sawada, A. Cassidy, *et al.*, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537–1557, 2015. DOI: 10.1109/TCAD.2015.2474396.

[75] M. Davies, N. Srinivasa, T.-H. Lin, *et al.*, "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning", *IEEE Micro*, vol. 38, no. 1, pp. 82–99, Jan. 2018, ISSN: 0272-1732. DOI: 10.1109/MM.2018.112130359. [Online]. Available: http://ieeexplore.ieee.org/document/8259423/ (visited on 11/27/2020).

[76] G. Orchard, E. P. Frady, D. B. D. Rubin, *et al.*, "Efficient Neuromorphic Signal Processing with Loihi 2", in *2021 IEEE Workshop on Signal Processing Systems (SiPS)*, Coimbra, Portugal: IEEE, Oct. 2021, pp. 254–259, ISBN: 978-1-66540-144-9. DOI: 10.1109/SiPS52927.2021.00053. [Online]. Available: https://ieeexplore.ieee.org/document/9605018/ (visited on 09/02/2022).

[77] M. Davies, N. Srinivasa, T. Lin, *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning", *IEEE Micro*, vol. 38, no. 01, pp. 82–99, Jan. 2018, ISSN: 1937-4143. DOI: 10.1109/MM.2018.112130359.

[78] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. S. Maida, "Deep Learning in Spiking Neural Networks", *Neural Networks*, vol. 111, pp. 47–63, Mar. 2019, arXiv:1804.08150 [cs], ISSN: 08936080. DOI: 10.1016/j.neunet.2018.12.002. [Online]. Available: http://arxiv.org/abs/1804.08150 (visited on 08/05/2022).

[79] E. O. Neftci, H. Mostafa, and F. Zenke, "Surrogate Gradient Learning in Spiking Neural Networks: Bringing the Power of Gradient-Based Optimization to Spiking Neural Networks", en, *IEEE Signal Processing Magazine*, vol. 36, no. 6, pp. 51–63, Nov. 2019, ISSN: 1053-5888, 1558-0792. DOI: 10.1109/MSP.2019.2931595. [Online]. Available: https://ieeexplore.ieee.org/document/8891809/ (visited on 08/24/2022).

[80] E. Goto, "The Parametron, a Digital Computing Element Which Utilizes Parametric Oscillation", *Proceedings of the IRE*, vol. 47, no. 8, pp. 1304–1316, Aug. 1959, ISSN: 0096-8390. DOI: 10.1109/JRPROC.1959.287195. [Online]. Available: http://ieeexplore.ieee.org/document/4065825/ (visited on 07/21/2022).

[81] G. E. Moore, "Progress in digital integrated electronics [Technical literaiture, Copyright 1975 IEEE. Reprinted, with permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.]", *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 36–37, Sep. 2006, Conference Name: IEEE Solid-State Circuits Society Newsletter, ISSN: 1098-4232. DOI: 10.1109/N-SSC.2006.4804410.

[82] C. Mead and M. Ismail, *Analog VLSI implementation of neural systems*. Springer Science & Business Media, 1989, vol. 80.

[83] C. A. Mead and M. A. Mahowald, "A silicon model of early visual processing", en, *Neural Networks*, vol. 1, no. 1, pp. 91–97, Jan. 1988, ISSN: 0893-6080. DOI: 10.1016/0893-6080(88)90024-X. [Online]. Available: https://www.sciencedirect.com/science/article/pii/089360808890024X (visited on 09/01/2022).

[84] L. Chua, "Memristor-The missing circuit element", *IEEE Transactions on Circuit Theory*, vol. 18, no. 5, pp. 507–519, Sep. 1971, Conference Name: IEEE Transactions on Circuit Theory, ISSN: 2374-9555. DOI: 10.1109/TCT.1971.1083337.

[85] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found", en, *Nature*, vol. 453, no. 7191, pp. 80–83, May 2008, Number: 7191 Publisher: Nature Publishing Group, ISSN: 1476-4687. DOI: 10.1038/nature06932. [Online]. Available: https://www.nature.com/articles/nature06932 (visited on 08/24/2022).

[86] P. Yao, H. Wu, B. Gao, *et al.*, "Fully hardware-implemented memristor convolutional neural network", en, *Nature*, vol. 577, no. 7792, pp. 641–646, Jan. 2020, ISSN: 0028-0836, 1476-4687. DOI: 10.1038/s41586-020-1942-4. [Online]. Available: http://www.nature.com/articles/s41586-020-1942-4 (visited on 04/12/2021).

[87] C. Mackin, M. J. Rasch, A. Chen, *et al.*, "Optimised weight programming for analogue memory-based deep neural networks", en, *Nature Communications*, vol. 13, no. 1, p. 3765, Dec. 2022, ISSN: 2041-1723. DOI: 10.1038/s41467-022-31405-1. [Online]. Available: https://www.nature.com/articles/s41467-022-31405-1 (visited on 07/25/2022).

[88] I. Kataeva, S. Ohtsuka, H. Nili, *et al.*, "Towards the Development of Analog Neuromorphic Chip Prototype with 2.4M Integrated Memristors", in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, Sapporo, Japan: IEEE, May 2019, pp. 1–5, ISBN: 978-1-72810-397-6. DOI: 10.1109/ISCAS.2019.8702125. [Online]. Available: https://ieeexplore.ieee.org/document/8702125/ (visited on 09/02/2022).

[89] H. Kim, M. R. Mahmoodi, H. Nili, and D. B. Strukov, "4K-memristor analog-grade passive crossbar circuit", en, *Nature Communications*, vol. 12, no. 1, p. 5198, Aug. 2021, Number: 1 Publisher: Nature Publishing Group, ISSN: 2041-1723. DOI: 10.1038/s41467-021-25455-0. [Online]. Available: https://www.nature.com/articles/s41467-021-25455-0 (visited on 09/02/2022).

[90] F. Cüppers, S. Menzel, C. Bengel, *et al.*, "Exploiting the switching dynamics of HfO2-based ReRAM devices for reliable analog memristive behavior", *APL Materials*, vol. 7, no. 9, p. 091 105, Sep. 2019, Publisher: American Institute of Physics. DOI: 10.1063/1.5108654. [Online]. Available: https://aip.scitation.org/doi/10.1063/1.5108654 (visited on 09/01/2022).

[91] M. Terai, Y. Sakotsubo, S. Kotsuji, and H. Hada, "Resistance Controllability of $\hboxTa_2 \hboxO_5/\hboxTiO_2$ Stack ReRAM for Low-Voltage and Multilevel Operation", *IEEE Electron Device Letters*, vol. 31, no. 3, pp. 204–206, Mar. 2010, Conference Name: IEEE Electron Device Letters, ISSN: 1558-0563. DOI: 10.1109/LED.2009.2039021.

[92] M. Stanisavljevic, H. Pozidis, A. Athmanathan, N. Papandreou, T. Mittelholzer, and E. Eleftheriou, "Demonstration of Reliable Triple-Level-Cell (TLC) Phase-Change Memory", in *2016 IEEE 8th International Memory Workshop (IMW)*, May 2016, pp. 1–4. DOI: 10.1109/IMW.2016.7495263.

[93] N. H. Seong, S. Yeo, and H.-H. S. Lee, "Tri-level-cell phase change memory: Toward an efficient and reliable memory system", in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, New York, NY, USA: Association for Computing Machinery, Jun. 2013, pp. 440–451, ISBN: 978-1-4503-2079-5. DOI: 10.1145/2485922.2485960. [Online]. Available: https://doi.org/10.1145/2485922.2485960 (visited on 09/01/2022).

[94] A. Fantini, V. Sousa, L. Perniola, *et al.*, "N-doped GeTe as performance booster for embedded Phase-Change Memories", in *2010 International Electron Devices Meeting*, ISSN: 2156-017X, Dec. 2010, pp. 29.1.1–29.1.4. DOI: 10.1109/IEDM.2010.5703441.

[95] D. Querlioz, P. Dollfus, O. Bichler, and C. Gamrat, "Learning with memristive devices: How should we model their behavior?", in *2011 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, ser. 2011 IEEE/ACM International Symposium on Nanoscale Architectures, San Diego, United States: IEEE, Jun. 2011. DOI: 10.1109/NANOARCH.2011.5941497. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01827056 (visited on 09/01/2022).

[96] O. Krestinskaya, A. Irmanova, and A. James, "Memristive non-idealities: Is there any practical implications for designing neural network chips?", May 2019. DOI: 10.1109/ISCAS.2019.8702245.

[97] Z. Liao, J. Fu, and J. Wang, "Level scaling and pulse regulating methods to mitigate cycle-to-cycle variation in memristor based learning system", Jul. 2020. DOI: 10.21203/rs.3.rs-49761/v1.

[98] V. Ntinas, I.-A. Fyrigos, G. C. Sirakoulis, *et al.*, "Noise-induced performance enhancement of variability-aware memristor networks", in *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2019, pp. 731–734. DOI: 10.1109/ICECS46596.2019.8965134.

[99] S. H. Jo, T. Chang, I. Ebong, B. B. Bhadviya, P. Mazumder, and W. Lu, "Nanoscale Memristor Device as Synapse in Neuromorphic Systems", en, *Nano Letters*, vol. 10, no. 4, pp. 1297–1301, Apr. 2010, ISSN: 1530-6984, 1530-6992. DOI: 10.1021/nl904092h. [Online]. Available: https://pubs.acs.org/doi/10.1021/nl904092h (visited on 07/26/2022).

[100] D. Joksas and A. Mehonic, "Badcrossbar: A Python tool for computing and plotting currents and voltages in passive crossbar arrays", en, *SoftwareX*, vol. 12, p. 100 617, Jul. 2020, ISSN: 2352-7110. DOI: 10.1016/j.softx.2020.100617. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711020303307 (visited on 08/24/2022).

[101] N. Leroux, D. Marković, E. Martin, *et al.*, "Radio-Frequency Multiply-And-Accumulate Operations with Spintronic Synapses", *arXiv:2011.07885 [cond-mat]*, Nov. 2020, arXiv: 2011.07885. [Online]. Available: http://arxiv.org/abs/2011.07885 (visited on 11/25/2020).

[102] N. Leroux, A. Mizrahi, D. Marković, *et al.*, "Hardware realization of the multiply and accumulate operation on radio-frequency signals with magnetic tunnel junctions", *Neuromorphic Computing and Engineering*, vol. 1, no. 1, p. 011 001, Jul. 2021. DOI: 10.1088/2634-4386/abfca6. [Online]. Available: https://doi.org/10.1088/2634-4386/abfca6.

[103] C. S. Thakur, J. L. Molin, G. Cauwenberghs, *et al.*, "Large-Scale Neuromorphic Spiking Array Processors: A Quest to Mimic the Brain", English, *Frontiers in Neuroscience*, vol. 12, 2018, Publisher: Frontiers, ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00891. (visited on 06/15/2020).

[104] Z. Wang, H. Wu, G. W. Burr, *et al.*, "Resistive switching materials for information processing", *Nature Reviews Materials*, vol. 5, no. 3, pp. 173–195, 2020. DOI: 10.1038/s41578-019-0159-3. [Online]. Available: https://doi.org/10.1038/s41578-019-0159-3.

[105] J. Torrejon, M. Riou, F. A. Araujo, *et al.*, "Neuromorphic computing with nanoscale spintronic oscillators", en, *Nature*, vol. 547, no. 7664, pp. 428–431, Jul. 2017, ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature23011. [Online]. Available: http://www.nature.com/articles/nature23011 (visited on 08/05/2022).

[106] E. Skibinsky-Gitlin, M. Alomar, C. Frasser, *et al.*, "Cyclic Reservoir Computing with FPGA Devices for Efficient Channel Equalization", in Jan. 2018, pp. 226–234, ISBN: 978-3-319-91252-3. DOI: 10.1007/978-3-319-91253-0_22.

[107] D. Marković, N. Leroux, M. Riou, *et al.*, "Reservoir computing with the frequency, phase, and amplitude of spin-torque nano-oscillators", *Applied Physics Letters*, vol. 114, no. 1, p. 012 409, Jan. 2019, Publisher: American Institute of Physics, ISSN: 0003-6951. DOI: 10.1063/1.5079305. [Online]. Available: https://aip.scitation.org/doi/10.1063/1.5079305 (visited on 09/01/2022).

[108] J. Dudas, J. Grollier, and D. Marković, *Coherently coupled quantum oscillators for quantum reservoir computing*, arXiv:2204.14273 [quant-ph], Apr. 2022. [Online]. Available: http://arxiv.org/abs/2204.14273 (visited on 09/01/2022).

[109] L. G. Wright, T. Onodera, M. M. Stein, *et al.*, "Deep physical neural networks trained with backpropagation", en, *Nature*, vol. 601, no. 7894, pp. 549–555, Jan. 2022, Number: 7894 Publisher: Nature Publishing Group, ISSN: 1476-4687. DOI: 10.1038/s41586-021-04223-6. [Online]. Available: https://www.nature.com/articles/s41586-021-04223-6 (visited on 08/30/2022).

[110] W. A. Little, "The existence of persistent states in the brain", en, *Mathematical Biosciences*, vol. 19, no. 1, pp. 101–120, Feb. 1974, ISSN: 0025-5564. DOI: 10.1016/0025-5564(74)90031-5. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0025556474900315 (visited on 07/15/2022).

[111] E. Ising, "Beitrag zur Theorie des Ferromagnetismus", de, *Zeitschrift für Physik*, vol. 31, no. 1, pp. 253–258, Feb. 1925, ISSN: 0044-3328. DOI: 10.1007/BF02980577. [Online]. Available: https://doi.org/10.1007/BF02980577 (visited on 08/24/2022).

[112] J. J. Hopfield, "Neurons with graded response have collective computational properties like those of two-state neurons.", *Proceedings of the National Academy of Sciences*, vol. 81, no. 10, pp. 3088–3092, May 1984, Publisher: Proceedings of the National Academy of Sciences. DOI: 10.1073/pnas.81.10.3088. [Online]. Available: https://www.pnas.org/doi/10.1073/pnas.81.10.3088 (visited on 08/24/2022).

[113] "The non-local storage of temporal information", *Proceedings of the Royal Society of London. Series B. Biological Sciences*, vol. 171, no. 1024, pp. 327–334, Dec. 1968. DOI: 10.1098/rspb.1968.0074. [Online]. Available: https://doi.org/10.1098/rspb.1968.0074.

[114] "Associative memory. a system-theoretical approach", *Acta Biotheoretica*, vol. 28, no. 1, pp. 70–71, 1979. DOI: 10.1007/bf00054681. [Online]. Available: https://doi.org/10.1007/bf00054681.

[115] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing", *Science*, vol. 220, no. 4598, pp. 671–680, May 1983, Publisher: American Association for the Advancement of Science. DOI: 10.1126/science.220.4598.671. [Online]. Available: https://www.science.org/doi/10.1126/science.220.4598.671 (visited on 08/24/2022).

[116] M. Keyvanrad and M. Homayoonpoor, "A brief survey on deep belief networks and introducing a new object oriented MATLAB toolbox (DeeBNet V2.0)", Aug. 2014.

[117] R. Salakhutdinov and G. Hinton, "Deep Boltzmann Machines", en, in *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, ISSN: 1938-7228, PMLR, Apr. 2009, pp. 448–455. [Online]. Available: https://proceedings.mlr.press/v5/salakhutdinov09a.html (visited on 08/11/2022).

[118] Y. Zhou, D. Arpit, I. Nwogu, and V. Govindaraju, *Is Joint Training Better for Deep Auto-Encoders?*, arXiv:1405.1380 [cs, stat], Jun. 2015. [Online]. Available: http://arxiv.org/abs/1405.1380 (visited on 08/16/2022).

[119] W. H. L. Pinaya, A. Gadelha, O. M. Doyle, *et al.*, "Using deep belief network modelling to characterize differences in brain morphometry in schizophrenia", eng, *Scientific Reports*, vol. 6, p. 38 897, Dec. 2016, ISSN: 2045-2322. DOI: 10.1038/srep38897.

[120] S. Dillavou, M. Stern, A. J. Liu, and D. J. Durian, "Demonstration of Decentralized, Physics-Driven Learning", *arXiv:2108.00275 [cond-mat]*, Mar. 2022, arXiv: 2108.00275. [Online]. Available: http://arxiv.org/abs/2108.00275 (visited on 03/15/2022).

[121] A. Lucas, "Ising formulations of many NP problems", *Frontiers in Physics*, vol. 2, 2014, ISSN: 2296-424X. [Online]. Available: https://www.frontiersin.org/article/10.3389/fphy.2014.00005 (visited on 06/24/2022).

[122] F. Barahona, "On the computational complexity of Ising spin glass models", en, *Journal of Physics A: Mathematical and General*, vol. 15, no. 10, pp. 3241–3253, Oct. 1982, ISSN: 0305-4470, 1361-6447. DOI: 10.1088/0305-4470/15/10/028. [Online]. Available: https://iopscience.iop.org/article/10.1088/0305-4470/15/10/028 (visited on 07/05/2022).

[123] N. Mohseni, P. L. McMahon, and T. Byrnes, "Ising machines as hardware solvers of combinatorial optimization problems", *Nature Reviews Physics*, vol. 4, no. 6, pp. 363–379, 2022.

[124] R. J. Glauber, "Time dependent statistics of the ising model", *Journal of Mathematical Physics*, vol. 4, no. 2, pp. 294–307, 1963. DOI: 10.1063/1.1703954. eprint: https://doi.org/10.1063/1.1703954. [Online]. Available: https://doi.org/10.1063/1.1703954.

[125] V. Granville, M. Krivanek, and J.-P. Rasson, "Simulated annealing: A proof of convergence", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 16, no. 6, pp. 652–656, Jun. 1994, Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence, ISSN: 1939-3539. DOI: 10.1109/34.295910.

[126] D. Vodenicarevic, N. Locatelli, A. Mizrahi, *et al.*, "Low-energy truly random number generation with superparamagnetic tunnel junctions for unconventional computing", *Phys. Rev. Applied*, vol. 8, p. 054 045, 5 Nov. 2017. DOI: 10.1103/PhysRevApplied.8.054045. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevApplied.8.054045.

[127] F. Cai, S. Kumar, T. Van Vaerenbergh, *et al.*, "Power-efficient combinatorial optimization using intrinsic noise in memristor Hopfield neural networks", en, *Nature Electronics*, vol. 3, no. 7, pp. 409–418, Jul. 2020, Number: 7 Publisher: Nature Publishing Group, ISSN: 2520-1131. DOI: 10.1038/s41928-020-0436-6. [Online]. Available: https://www.nature.com/articles/s41928-020-0436-6 (visited on 07/11/2022).

[128] H. Goto, K. Endo, M. Suzuki, *et al.*, "High-performance combinatorial optimization based on classical mechanics", *Science Advances*, vol. 7, no. 6, eabe7953, 2021. DOI: 10.1126/sciadv.abe7953. eprint: https://www.science.org/doi/pdf/10.1126/sciadv.abe7953. [Online]. Available: https://www.science.org/doi/abs/10.1126/sciadv.abe7953.

[129] R. Hamerly, T. Inagaki, P. L. McMahon, *et al.*, "Experimental investigation of performance differences between coherent Ising machines and a quantum annealer", en, *Science Advances*, vol. 5, no. 5, eaau0823, May 2019, ISSN: 2375-2548. DOI: 10.1126/sciadv.aau0823. [Online]. Available: https://www.science.org/doi/10.1126/sciadv.aau0823 (visited on 05/19/2022).

[130] T. Honjo, T. Sonobe, K. Inaba, *et al.*, "100,000-spin coherent ising machine", *Science Advances*, vol. 7, no. 40, eabh0952, 2021. DOI: 10.1126/sciadv.abh0952. eprint: https://www.science.org/doi/pdf/10.1126/sciadv.abh0952. [Online]. Available: https://www.science.org/doi/abs/10.1126/sciadv.abh0952.

[131] F. Cai, S. Kumar, T. Van Vaerenbergh, *et al.*, *Harnessing Intrinsic Noise in Memristor Hopfield Neural Networks for Combinatorial Optimization*, Number: arXiv:1903.11194 arXiv:1903.11194 [cs], Apr. 2019. [Online]. Available: http://arxiv.org/abs/1903.11194 (visited on 08/03/2022).

[132] Y. Koshka and M. A. Novotny, "Comparison of d-wave quantum annealing and classical simulated annealing for local minima determination", *IEEE Journal on Selected Areas in Information Theory*, vol. 1, no. 2, pp. 515–525, 2020. DOI: 10.1109/JSAIT.2020.3014192.

[133] C. Baldassi and R. Zecchina, "Efficiency of quantum vs. classical annealing in nonconvex learning problems", *Proceedings of the National Academy of Sciences*, vol. 115, no. 7, pp. 1457–1462, 2018. DOI: 10.1073/pnas.1711456115. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.1711456115. [Online]. Available: https://www.pnas.org/doi/abs/10.1073/pnas.1711456115.

[134] R. Harris, M. W. Johnson, T. Lanting, *et al.*, "Experimental Investigation of an Eight Qubit Unit Cell in a Superconducting Optimization Processor", en, *Physical Review B*, vol. 82, no. 2, p. 024 511, Jul. 2010, arXiv: 1004.1628, ISSN: 1098-0121, 1550-235X. DOI: 10.1103/PhysRevB.82.024511. [Online]. Available: http://arxiv.org/abs/1004.1628 (visited on 02/04/2022).

[135] S. Adachi, "Application of Quantum Annealing to Training of Deep Neural Networks", en, p. 18,

[136] D. Venturelli, S. Mandrà, S. Knysh, B. O'Gorman, R. Biswas, and V. Smelyanskiy, "Quantum optimization of fully connected spin glasses", *Phys. Rev. X*, vol. 5, p. 031 040, 3 Sep. 2015. DOI: 10.1103/PhysRevX.5.031040. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevX.5.031040.

[137] J. E. Dorband, "A Boltzmann Machine Implementation for the D-Wave", in *2015 12th International Conference on Information Technology - New Generations*, Las Vegas, NV, USA: IEEE, Apr. 2015, pp. 703–707, ISBN: 978-1-4799-8828-0. DOI: 10.1109/ITNG.2015.118. [Online]. Available: http://ieeexplore.ieee.org/document/7113557/ (visited on 02/03/2022).

[138] M. Kim, D. Venturelli, and K. Jamieson, "Leveraging quantum annealing for large mimo processing in centralized radio access networks", in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19, Beijing, China: Association for Computing Machinery, 2019, pp. 241–255, ISBN: 9781450359566. DOI: 10.1145/3341302.3342072. [Online]. Available: https://doi.org/10.1145/3341302.3342072.

[139] J. Liu, F. M. Spedalieri, K.-T. Yao, *et al.*, "Adiabatic Quantum Computation Applied to Deep Learning Networks", en, *Entropy*, vol. 20, no. 5, p. 380, May 2018, Number: 5 Publisher: Multidisciplinary Digital Publishing Institute, ISSN: 1099-4300. DOI: 10.3390/e20050380. [Online]. Available: https://www.mdpi.com/1099-4300/20/5/380 (visited on 02/03/2022).

[140] D. Venturelli and A. Kondratyev, "Reverse quantum annealing approach to portfolio optimization problems", *Quantum Machine Intelligence*, vol. 1, no. 1-2, pp. 17–30, Apr. 2019. DOI: 10.1007/s42484-019-00001-w. [Online]. Available: https://doi.org/10.1007/s42484-019-00001-w.

[141] V. Dixit, R. Selvarajan, M. A. Alam, T. S. Humble, and S. Kais, "Training Restricted Boltzmann Machines With a D-Wave Quantum Annealer", *Frontiers in Physics*, vol. 9, 2021, ISSN: 2296-424X. [Online]. Available: https://www.frontiersin.org/article/10.3389/fphy.2021.589626 (visited on 02/03/2022).

[142] C. Carugno, M. F. Dacrema, and P. Cremonesi, "Evaluating the job shop scheduling problem on a d-wave quantum annealer", *Scientific Reports*, vol. 12, no. 1, Apr. 2022. DOI: 10.1038/s41598-022-10169-0. [Online]. Available: https://doi.org/10.1038/s41598-022-10169-0.

[143] P. L. McMahon, A. Marandi, Y. Haribara, *et al.*, "A fully programmable 100-spin coherent ising machine with all-to-all connections", *Science*, vol. 354, no. 6312, pp. 614–617, Nov. 2016. DOI: 10.1126/science.aah5178. [Online]. Available: https://doi.org/10.1126/science.aah5178.

[144] T. Inagaki, Y. Haribara, K. Igarashi, *et al.*, "A coherent ising machine for 2000-node optimization problems", *Science*, vol. 354, no. 6312, pp. 603–606, Nov. 2016. DOI: 10.1126/science.aah4243. [Online]. Available: https://doi.org/10.1126/science.aah4243.

[145] Y. Yamamoto, K. Aihara, T. Leleu, *et al.*, “Coherent Ising machines—optical neural networks operating at the quantum limit”, en, *npj Quantum Information*, vol. 3, no. 1, p. 49, Dec. 2017, ISSN: 2056-6387. DOI: 10.1038/s41534-017-0048-9. [Online]. Available: http://www.nature.com/articles/s41534-017-0048-9 (visited on 03/14/2022).

[146] Y. Yamamoto, T. Leleu, S. Ganguli, and H. Mabuchi, “Coherent ising machines—quantum optics and neural network perspectives”, *Applied Physics Letters*, vol. 117, no. 16, p. 160 501, Oct. 2020. DOI: 10.1063/5.0016140. [Online]. Available: https://doi.org/10.1063/5.0016140.

[147] T. Honjo, T. Sonobe, K. Inaba, *et al.*, “100, 000-spin coherent ising machine”, *Science Advances*, vol. 7, no. 40, Oct. 2021. DOI: 10.1126/sciadv.abh0952. [Online]. Available: https://doi.org/10.1126/sciadv.abh0952.

[148] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors”, *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986. DOI: 10.1038/323533a0. [Online]. Available: https://doi.org/10.1038/323533a0.

[149] J. R. Movellan, “Contrastive hebbian learning in the continuous hopfield model”, in *Connectionist models*, Elsevier, 1991, pp. 10–17.

[150] J. Kendall, R. Pantone, K. Manickavasagam, Y. Bengio, and B. Scellier, “Training End-to-End Analog Neural Networks with Equilibrium Propagation”, *arXiv:2006.01981 [cs]*, Jun. 2020, arXiv: 2006.01981. [Online]. Available: http://arxiv.org/abs/2006.01981 (visited on 04/09/2021).

[151] Y. LeCun and C. Cortes, “MNIST handwritten digit database”, 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/.

[152] M. Stern, D. Hexner, J. W. Rocks, and A. J. Liu, “Supervised Learning in Physical Networks: From Machine Learning to Learning Machines”, en, *Physical Review X*, vol. 11, no. 2, p. 021 045, May 2021, ISSN: 2160-3308. DOI: 10.1103/PhysRevX.11.021045. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevX.11.021045 (visited on 07/09/2021).

[153] ——, “Supervised learning in physical networks: From machine learning to learning machines”, *Phys. Rev. X*, vol. 11, p. 021 045, 2 May 2021. DOI: 10.1103/PhysRevX.11.021045. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevX.11.021045.

[154] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database”, in *2009 IEEE conference on computer vision and pattern recognition*, Ieee, 2009, pp. 248–255.

[155] A. Laborieux and F. Zenke, *Holomorphic equilibrium propagation computes exact gradients through finite size oscillations*, 2022. DOI: 10.48550/ARXIV.2209.00530. [Online]. Available: https://arxiv.org/abs/2209.00530.

[156] B. Scellier, *A deep learning theory for neural networks grounded in physics*, 2021. DOI: 10.48550/ARXIV.2103.09985. [Online]. Available: https://arxiv.org/abs/2103.09985.

[157] 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 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 7081–7091.

[158] A. Krizhevsky, V. Nair, and G. Hinton, “Cifar-10 (canadian institute for advanced research)”,

[159] P. Chrabaszcz, I. Loshchilov, and F. Hutter, *A downsampled variant of imagenet as an alternative to the cifar datasets*, 2017. DOI: 10.48550/ARXIV.1707.08819. [Online]. Available: https://arxiv.org/abs/1707.08819.

[160] K. Helwegen, J. Widdicombe, L. Geiger, Z. Liu, K.-T. Cheng, and R. Nusselder, “Latent weights do not exist: Rethinking binarized neural network optimization”, in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019, pp. 7533–7544.

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

[162] B. Scellier, A. Goyal, J. Binas, T. Mesnard, and Y. Bengio, *Generalization of equilibrium propagation to vector field dynamics*, 2018. DOI: 10.48550/ARXIV.1808.04873. [Online]. Available: https://arxiv.org/abs/1808.04873.

[163] G. Zoppo, F. Marrone, and F. Corinto, "Equilibrium propagation for memristor-based recurrent neural networks", *Frontiers in neuroscience*, vol. 14, p. 240, 2020.

[164] Z. Ji and W. Gross, "Towards efficient on-chip learning using equilibrium propagation", in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2020, pp. 1–5.

[165] 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", *arXiv:2006.03824 [cs]*, Jun. 2020, arXiv: 2006.03824. [Online]. Available: http://arxiv.org/abs/2006.03824 (visited on 11/24/2020).

[166] Y. Yan, D. Kappel, F. Neumärker, *et al.*, "Efficient reward-based structural plasticity on a spinnaker 2 prototype", *IEEE Transactions on Biomedical Circuits and Systems*, vol. 13, no. 3, pp. 579–591, 2019. DOI: 10.1109/TBCAS.2019.2906401.

[167] P. Blouw, X. Choo, E. Hunsberger, and C. Eliasmith, *Benchmarking keyword spotting efficiency on neuromorphic hardware*, 2018. DOI: 10.48550/ARXIV.1812.01739. [Online]. Available: https://arxiv.org/abs/1812.01739.

[168] T. Mesnard, W. Gerstner, and J. Brea, "Towards deep learning with spiking neurons in energy based models with contrastive Hebbian plasticity", *arXiv:1612.03214 [cs, q-bio]*, Dec. 2016, arXiv: 1612.03214.

[169] E. Martin, M. Ernoult, J. Laydevant, *et al.*, "EqSpike: Spike-driven Equilibrium Propagation for Neuromorphic Implementations", *arXiv:2010.07859 [cs]*, Jan. 2021, arXiv: 2010.07859. (visited on 02/03/2021).

[170] M. Ernoult, J. Grollier, D. Querlioz, Y. Bengio, and B. Scellier, *Equilibrium propagation with continual weight updates*, 2020. DOI: 10.48550/ARXIV.2005.04168. [Online]. Available: https://arxiv.org/abs/2005.04168.

[171] E. Martin, "Réseaux de neurones à l'échelle nano. Quels modèles d'apprentissage ?", Theses, Université Paris-Saclay, Jul. 2022. [Online]. Available: https://tel.archives-ouvertes.fr/tel-03774146.

[172] Y. Bengio, N. Léonard, and A. C. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation", *CoRR*, vol. abs/1308.3432, 2013. arXiv: 1308.3432. [Online]. Available: http://arxiv.org/abs/1308.3432.

[173] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks", en, in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2016, pp. 525–542, ISBN: 978-3-319-46493-0. DOI: 10.1007/978-3-319-46493-0_32.

[174] T. Hirtzlin, B. Penkovsky, M. Bocquet, J.-O. Klein, J.-M. Portal, and D. Querlioz, "Stochastic Computing for Hardware Implementation of Binarized Neural Networks", *IEEE Access*, vol. 7, pp. 76 394–76 403, 2019, Conference Name: IEEE Access, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2921104.

[175] V. Parmar, B. Penkovsky, D. Querlioz, and M. Suri, "Hardware-Efficient Stochastic Binary CNN Architectures for Near-Sensor Computing", *Frontiers in Neuroscience*, vol. 15, 2022, ISSN: 1662-453X. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2021.781786 (visited on 06/29/2022).

[176] K. Helwegen, J. Widdicombe, L. Geiger, Z. Liu, K.-T. Cheng, and R. Nusselder, "Latent Weights Do Not Exist: Rethinking Binarized Neural Network Optimization", in *Advances in Neural Information Processing Systems*, vol. 32, Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/9ca8c9b0996bbf05ae7753d34667a6fd-Abstract.html (visited on 06/24/2022).

[177] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", *arXiv:1502.03167 [cs]*, Mar. 2015, arXiv: 1502.03167. [Online]. Available: http://arxiv.org/abs/1502.03167 (visited on 05/20/2021).

[178] Y. Umuroglu, N. J. Fraser, G. Gambardella, *et al.*, "Finn: A framework for fast, scalable binarized neural network inference", in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17, Monterey, California, USA: Association for Computing Machinery, 2017, pp. 65–74, ISBN: 9781450343541. DOI: 10.1145/3020078.3021744. [Online]. Available: https://doi.org/10.1145/3020078.3021744.

[179] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "Fp-bnn: Binarized neural network on fpga", *Neurocomputing*, vol. 275, Oct. 2017. DOI: 10.1016/j.neucom.2017.09.046.

[180] R. Zhao, W. Song, W. Zhang, *et al.*, "Accelerating binarized convolutional neural networks with software-programmable fpgas", in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17, Monterey, California, USA: Association for Computing Machinery, 2017, pp. 15–24, ISBN: 9781450343541. DOI: 10.1145/3020078.3021741. [Online]. Available: https://doi.org/10.1145/3020078.3021741.

[181] N. Fraser, Y. Umuroglu, G. Gambardella, *et al.*, "Scaling binarized neural networks on reconfigurable logic", Jan. 2017. DOI: 10.1145/3029580.3029586.

[182] T. Hirtzlin, B. Penkovsky, J.-O. Klein, *et al.*, *Implementing Binarized Neural Networks with Magnetoresistive RAM without Error Correction*, Number: arXiv:1908.04085 arXiv:1908.04085 [cs], Aug. 2019. [Online]. Available: http://arxiv.org/abs/1908.04085 (visited on 06/23/2022).

[183] T. Hirtzlin, M. Bocquet, J.-O. Klein, *et al.*, *Outstanding Bit Error Tolerance of Resistive RAM-Based Binarized Neural Networks*, Number: arXiv:1904.03652 arXiv:1904.03652 [cs], Apr. 2019. [Online]. Available: http://arxiv.org/abs/1904.03652 (visited on 06/23/2022).

[184] T. Hirtzlin, M. Bocquet, M. Ernoult, *et al.*, "Hybrid analog-digital learning with differential rram synapses", in *2019 IEEE International Electron Devices Meeting (IEDM)*, IEEE, 2019, pp. 22–6.

[185] T. Hirtzlin, M. Bocquet, B. Penkovsky, *et al.*, "Digital biologically plausible implementation of binarized neural networks with differential hafnium oxide resistive memory arrays", *Frontiers in Neuroscience*, vol. 13, p. 1383, 2020, ISSN: 1662-453X. DOI: 10.3389/fnins.2019.01383.

[186] 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*, arXiv: 2006.03824, 2020. arXiv: 2006.03824 [cs.NE].

[187] D. Marković, A. Mizrahi, D. Querlioz, and J. Grollier, "Physics for neuromorphic computing", en, *Nature Reviews Physics*, pp. 1–12, Jul. 2020, Publisher: Nature Publishing Group, ISSN: 2522-5820. DOI: 10.1038/s42254-020-0208-2. (visited on 08/19/2020).

[188] R. Gopalakrishnan, Y. Chua, P. Sun, A. J. Sreejith Kumar, and A. Basu, "Hfnet: A cnn architecture co-designed for neuromorphic hardware with a crossbar array of synapses", *Frontiers in Neuroscience*, vol. 14, p. 907, 2020, ISSN: 1662-453X. DOI: 10.3389/fnins.2020.00907.

[189] S. Ambrogio, P. Narayanan, H. Tsai, *et al.*, "Equivalent-accuracy accelerated neural-network training using analogue memory", *Nature*, vol. 558, Jun. 2018. DOI: 10.1038/s41586-018-0180-5.

[190] W. Zhang, B. Gao, J. Tang, *et al.*, "Neuro-inspired computing chips", en, *Nature Electronics*, vol. 3, no. 7, pp. 371–382, Jul. 2020, ISSN: 2520-1131. DOI: 10.1038/s41928-020-0435-7. [Online]. Available: http://www.nature.com/articles/s41928-020-0435-7 (visited on 09/01/2022).

[191] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks", en, in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2016, pp. 525–542, ISBN: 978-3-319-46493-0. DOI: 10.1007/978-3-319-46493-0_32.

[192] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", en, in *International Conference on Machine Learning*, ISSN: 1938-7228, PMLR, Jun. 2015, pp. 448–456. (visited on 02/04/2021).

[193] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks", in *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds., Curran Associates, Inc., 2016, pp. 4107–4115.

[194] C. Laurent, G. Pereyra, P. Brakel, Y. Zhang, and Y. Bengio, "Batch normalized recurrent neural networks", in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2016, pp. 2657–2661.

[195] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", in *2015 IEEE International Conference on Computer Vision (ICCV)*, ISSN: 2380-7504, Dec. 2015, pp. 1026–1034. DOI: 10.1109/ICCV.2015.123.

[196] A. Bulat and G. Tzimiropoulos, "XNOR-Net++: Improved Binary Neural Networks", *arXiv:1909.13863 [cs, eess]*, Sep. 2019, arXiv: 1909.13863.

[197] K. F. Cheung, L. E. Atlas, and R. J. Marks, "Synchronous vs asynchronous behavior of hopfield's cam neural net", *Appl. Opt.*, vol. 26, no. 22, pp. 4808–4813, Nov. 1987. DOI: 10.1364/AO.26.004808.

[198] B. Scellier and Y. Bengio, "Equivalence of equilibrium propagation and recurrent backpropagation", *Neural computation*, vol. 31, no. 2, pp. 312–329, 2019.

[199] S. Bartunov, A. Santoro, B. A. Richards, L. Marris, G. E. Hinton, and T. P. Lillicrap, "Assessing the scalability of biologically-motivated deep learning algorithms and architectures", in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18, Montréal, Canada: Curran Associates Inc., 2018, pp. 9390–9400.

[200] S. Tulyakov, S. Jaeger, V. Govindaraju, and D. Doermann, "Review of classifier combination methods", in *Machine Learning in Document Analysis and Recognition*, S. Marinai and H. Fujisawa, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 361–386, ISBN: 978-3-540-76280-5. DOI: 10.1007/978-3-540-76280-5_14. [Online]. Available: https://doi.org/10.1007/978-3-540-76280-5_14.

[201] Y. Bengio, A. C. Courville, and P. Vincent, "Unsupervised feature learning and deep learning: A review and new perspectives", *ArXiv*, vol. abs/1206.5538, 2012.

[202] T. Hirtzlin, M. Bocquet, B. Penkovsky, *et al.*, "Digital biologically plausible implementation of binarized neural networks with differential hafnium oxide resistive memory arrays", *Frontiers in neuroscience*, vol. 13, p. 1383, 2020.

[203] P. O'Connor, E. Gavves, and M. Welling, "Training a spiking neural network with equilibrium propagation", in *Proceedings of Machine Learning Research*, K. Chaudhuri and M. Sugiyama, Eds., ser. Proceedings of Machine Learning Research, vol. 89, PMLR, Apr. 2019, pp. 1516–1523.

[204] S. Ambrogio, P. Narayanan, H. Tsai, *et al.*, "Equivalent-accuracy accelerated neural-network training using analogue memory", eng, *Nature*, vol. 558, no. 7708, pp. 60–67, Jun. 2018, ISSN: 1476-4687. DOI: 10.1038/s41586-018-0180-5.

[205] D. Vodenicarevic, N. Locatelli, A. Mizrahi, *et al.*, "Low-Energy Truly Random Number Generation with Superparamagnetic Tunnel Junctions for Unconventional Computing", *Physical Review Applied*, vol. 8, no. 5, p. 054 045, Nov. 2017, Publisher: American Physical Society. DOI: 10.1103/PhysRevApplied.8.054045. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevApplied.8.054045 (visited on 09/01/2022).

[206] G. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors", *arXiv preprint*, vol. arXiv, Jul. 2012.

[207] I. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout networks", in *Proceedings of the 30th International Conference on Machine Learning*, S. Dasgupta and D. McAllester, Eds., ser. Proceedings of Machine Learning Research, vol. 28, Atlanta, Georgia, USA: PMLR, Jun. 2013, pp. 1319–1327. [Online]. Available: https://proceedings.mlr.press/v28/goodfellow13.html.

[208] Y. Bengio, N. Léonard, and A. Courville, *Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation*, Number: arXiv:1308.3432 arXiv:1308.3432 [cs], Aug. 2013. [Online]. Available: http://arxiv.org/abs/1308.3432 (visited on 07/13/2022).

[209] H. Neven, V. S. Denchev, G. Rose, and W. G. Macready, "Training a Large Scale Classifier with the Quantum Adiabatic Algorithm", *arXiv:0912.0779 [quant-ph]*, Dec. 2009, arXiv: 0912.0779. [Online]. Available: http://arxiv.org/abs/0912.0779 (visited on 07/09/2021).

[210] H. Neven, G. Rose, and W. G. Macready, "Image recognition with an adiabatic quantum computer I. Mapping to quadratic unconstrained binary optimization", en, *arXiv:0804.4457 [quant-ph]*, Apr. 2008, arXiv: 0804.4457. [Online]. Available: http://arxiv.org/abs/0804.4457 (visited on 03/14/2022).

[211] V. S. Denchev, N. Ding, S. V. N. Vishwanathan, and H. Neven, "Robust Classification with Adiabatic Quantum Optimization", en, p. 8,

[212] H. Neven, V. S. Denchev, M. Drew-Brook, J. Zhang, W. G. Macready, and G. Rose, "NIPS 2009 Demonstration: Binary Classification using Hardware Implementation of Quantum Annealing", en, p. 17,

[213] N. T. T. Nguyen, A. E. Larson, and G. T. Kenyon, "Generating Sparse Representations Using Quantum Annealing: Comparison to Classical Algorithms", in *2017 IEEE International Conference on Rebooting Computing (ICRC)*, Washington, DC: IEEE, Nov. 2017, pp. 1–6, ISBN: 978-1-5386-1553-9. DOI: 10.1109/ICRC.2017.8123653. [Online]. Available: http://ieeexplore.ieee.org/document/8123653/ (visited on 03/14/2022).

[214] N. T. T. Nguyen and G. T. Kenyon, "Image classification using quantum inference on the D-Wave 2X", *2018 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–7, Nov. 2018, arXiv: 1905.13215. DOI: 10.1109/ICRC.2018.8638596. [Online]. Available: http://arxiv.org/abs/1905.13215 (visited on 02/03/2022).

[215] J. Sleeman, J. Dorband, and M. Halem, "A hybrid quantum enabled RBM advantage: convolutional autoencoders for quantum image compression and generative learning", in *Quantum Information Science, Sensing, and Computation XII*, E. Donkor and M. Hayduk, Eds., International Society for Optics and Photonics, vol. 11391, SPIE, 2020, 113910B. DOI: 10.1117/12.2558832. [Online]. Available: https://doi.org/10.1117/12.2558832.

[216] M. W. Johnson, P. Bunyk, F. Maibaum, *et al.*, "A scalable control system for a superconducting adiabatic quantum optimization processor", *Superconductor Science and Technology*, vol. 23, no. 6, p. 065 004, Jun. 2010, arXiv:0907.3757 [cond-mat, physics:physics, physics:quant-ph], ISSN: 0953-2048, 1361-6668. DOI: 10.1088/0953-2048/23/6/065004. [Online]. Available: http://arxiv.org/abs/0907.3757 (visited on 08/18/2022).

[217] W. M. Kaminsky, S. Lloyd, and T. P. Orlando, *Scalable Superconducting Architecture for Adiabatic Quantum Computation*, arXiv:quant-ph/0403090, Mar. 2004. [Online]. Available: http://arxiv.org/abs/quant-ph/0403090 (visited on 08/18/2022).

[218] M. Born and V. Fock, "Beweis des Adiabatensatzes", de, *Zeitschrift für Physik*, vol. 51, no. 3, pp. 165–180, Mar. 1928, ISSN: 0044-3328. DOI: 10.1007/BF01343193. [Online]. Available: https://doi.org/10.1007/BF01343193 (visited on 07/07/2022).

[219] N. Dattani, S. Szalay, and N. Chancellor, *Pegasus: The second connectivity graph for large-scale quantum annealing hardware*, 2019. DOI: 10.48550/ARXIV.1901.07636. [Online]. Available: https://arxiv.org/abs/1901.07636.

[220] T. Boothby, A. D. King, and A. Roy, "Fast clique minor generation in chimera qubit connectivity graphs", *Quantum Information Processing*, vol. 15, no. 1, pp. 495–508, 2016. DOI: 10.1007/s11128-015-1150-6. [Online]. Available: https://doi.org/10.1007/s11128-015-1150-6.

[221] C. Klymko, B. Sullivan, and T. Humble, "Adiabatic quantum programming: Minor embedding with hard faults", *Quantum Information Processing*, vol. 13, Oct. 2012. DOI: 10.1007/s11128-013-0683-9.

[222] M. H. Amin, "Searching for quantum speedup in quasistatic quantum annealers", *Phys. Rev. A*, vol. 92, p. 052 323, 5 Nov. 2015. DOI: 10.1103/PhysRevA.92.052323. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.92.052323.

[223] E. K. Grant and T. S. Humble, *Adiabatic quantum computing and quantum annealing*, Jul. 2020. DOI: 10.1093/acrefore/9780190871994.013.32. [Online]. Available: https://doi.org/10.1093/acrefore/9780190871994.013.32.

[224] M. Nielsen, *Reduced mnist: How well can machines learn from small data?*, https://cognitivemedium.com/rmnist, Accessed: 2022-09-03.

[225] S. Boixo, T. Albash, F. M. Spedalieri, N. Chancellor, and D. A. Lidar, "Experimental signature of programmable quantum annealing", *Nature Communications*, vol. 4, no. 1, p. 2067, 2013. DOI: 10.1038/ncomms3067. [Online]. Available: https://doi.org/10.1038/ncomms3067.

[226] T. F. Rønnow, Z. Wang, J. Job, *et al.*, "Defining and detecting quantum speedup", *Science*, vol. 345, no. 6195, pp. 420–424, Jul. 2014. DOI: 10.1126/science.1252319. [Online]. Available: https://doi.org/10.1126/science.1252319.

[227] A. Dosovitskiy, L. Beyer, A. Kolesnikov, *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale", in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=YicbFdNTTy.

[228] T. Xiao, M. Singh, E. Mintun, T. Darrell, P. Dollár, and R. Girshick, "Early convolutions help transformers see better", *Advances in Neural Information Processing Systems*, vol. 34, pp. 30 392–30 400, 2021.

[229] E. Farhi, J. Goldstone, S. Gutmann, J. Lapan, A. Lundgren, and D. Preda, "A Quantum Adiabatic Evolution Algorithm Applied to Random Instances of an NP-Complete Problem", *Science*, vol. 292, no. 5516, pp. 472–475, Apr. 2001, Publisher: American Association for the Advancement of Science. DOI: 10.1126/science.1057726. [Online]. Available: https://www.science.org/doi/10.1126/science.1057726 (visited on 06/24/2022).

[230] T. Onodera, "A quantum annealer with fully programmable all-to-all coupling via Floquet engineering", en, *npj Quantum Information*, p. 10, 2020.

[231] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks", *SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 367–379, Jun. 2016, ISSN: 0163-5964. DOI: 10.1145/3007787.3001177. [Online]. Available: https://doi.org/10.1145/3007787.3001177.

[232] J. Launay, I. Poli, K. Müller, *et al.*, "Hardware beyond backpropagation: A photonic co-processor for direct feedback alignment", *CoRR*, vol. abs/2012.06373, 2020. arXiv: 2012.06373. [Online]. Available: https://arxiv.org/abs/2012.06373.

[233] S. Still, D. A. Sivak, A. J. Bell, and G. E. Crooks, "Thermodynamics of prediction", *Phys. Rev. Lett.*, vol. 109, p. 120 604, 12 Sep. 2012. DOI: 10.1103/PhysRevLett.109.120604. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.109.120604.

[234] J. Sacramento, R. Ponte Costa, Y. Bengio, and W. Senn, "Dendritic cortical microcircuits approximate the backpropagation algorithm", *Advances in neural information processing systems*, vol. 31, 2018.

[235] S. Greydanus, M. Dzamba, and J. Yosinski, "Hamiltonian neural networks", *Advances in neural information processing systems*, vol. 32, 2019.

[236] M. Cranmer, S. Greydanus, S. Hoyer, P. Battaglia, D. Spergel, and S. Ho, "Lagrangian neural networks", *arXiv preprint arXiv:2003.04630*, 2020.

[237] P. Cha, P. Ginsparg, F. Wu, J. Carrasquilla, P. L. McMahon, and E.-A. Kim, "Attention-based quantum tomography", en, *Mach. Learn.*, p. 8, 2022.

Chapter A

Training Fully Connected Layers Networks with Equilibrium Propagation

In this section, we describe and define all the operations we used to train with EP a fully connected neural network recurrently connected through bidirectional synapses. We describe the dynamics and the underlying learning rules for the weights and the biases in the energy-based and prototypical settings. The units of the system are denoted $\mathbf{s} = \{h, y\}$ where h are the hidden units and y are the output units. The variable y is the one-hot encoded target vector. The inputs x are always clamped and are static.

A.1 . Energy-Based Settings

In the energy based settings, we introduce an energy function for the network, that defines the neuron dynamics during the two phases of EP. We then derive the learning rules from the energy function.

A.1.1 . Energy Function

We consider the following energy function [14]:

$$E(s, \rho(s), \theta = \{W, b\}) := \frac{1}{2}\sum_i s_i - \frac{1}{2}\sum_{i \neq j} W_{ij}\rho(s_i)\rho(s_i) - \sum_i b_i \rho(s_i) \tag{A.1}$$

where ρ is the activation function of the neurons, W_{ij} the weight connecting the unit s_i to s_j and reciprocally as synapses are symmetric for the system to converge and b_i the bias of unit s_i.

We also define ℓ the cost function describing how far are the output units of the system ($\hat{y}$) from their target state (y). We usually employ the mean squared error as a cost function with EP:

$$\ell(s, \mathrm{y}) = MSE(s, \mathrm{y}) := \frac{1}{2}\sum ||\mathrm{y} - \hat{y}||^2 \tag{A.2}$$

where y denotes a given target output.

Dynamics

The dynamics of neurons in the free phase evolve according to the energy function E:

$$\frac{ds}{dt} = -\frac{\partial E}{\partial s} \tag{A.3}$$

which translates for the neuron i and the energy function E defined in Eq.A.1 as:

$$\frac{ds_i}{dt} = -s_i + \rho'(s_i)(\sum_{j \neq i} W_{ij}\rho(s_j) + b) \tag{A.4}$$

The system eventually settles to a fixed steady state s_*.

During the nudged phase the dynamics differs from the free phase as the neurons now evolve to decrease the cost function ℓ:

$$\frac{ds}{dt} = -\frac{\partial E}{\partial s} - \beta \frac{\partial \ell}{\partial s} \tag{A.5}$$

which translates for the hidden unit h_i, the output unit $\hat{y}_i$ and the target y to:

$$\begin{cases} \frac{dh_i}{dt} = -h_i + \rho'(h_i)(\sum_{j \neq i} W_{ij}\rho(s_j) + b_i) \\ \frac{d\hat{y}_i}{dt} = -\hat{y}_i + \rho'(y_i)(\sum_{j \neq i} W_{ij}\rho(h_j) + b_i) + \beta \times (\mathsf{y_i} - \hat{y}_i) \end{cases} \tag{A.6}$$

The systems eventually reaches a second steady state denoted s_*^β.

Learning Rule

Scellier & Bengio [14] showed that the gradient of the loss $\mathcal{L}_*$ (defined in Eq. (...)) with respect to any parameter in the system can be approximated by the derivative of the energy function E with regard to the parameter evaluated at the two equilibrium points s_* and s_*^β:

$$-\frac{\partial \mathcal{L}_*}{\partial \theta} = \lim_{\beta \to 0} \frac{1}{\beta} \left(\frac{\partial E}{\partial \theta}(x, s_*^\beta, \theta) - \frac{\partial E}{\partial \theta}(x, s_*, \theta) \right) \tag{A.7}$$

In the energy-based settings, the resulting learning rules for the weights and biases are expressed as a function of the two steady states:

$$\begin{cases} \Delta W_{ij} = \frac{1}{\beta}(\rho(s_{i,*}^\beta)\rho(s_{j,*}^\beta) - \rho(s_{i,*})\rho(s_{j,*})) \\ \Delta b_i = \frac{1}{\beta}(\rho(s_{i,*}^\beta) - \rho(s_{i,*})) \end{cases} \tag{A.8}$$

A.2 . Prototypical Settings

Ernoult *et al*. [157] introduced the prototypical settings for EP where the dynamics no longer derived from an energy function in a continuous-time setting but more generally from a scalar primitive in a discrete-time setting. As in Ernoult *et al*. [157], we chose a dynamics close to the one of conventional RNNs. We then write a primitive function from which the dynamics derives. Finally we obtain the learning rules from the primitive function.

A.2.1 . Dynamics

We choose the same discrete time dynamics as in [157]:

$$\begin{cases} h_i^{t+1} = \rho(\sum_j W_{ij} s_j^t + b) \\ y_i^{t+1} = \rho(\sum_j W_{ij} h_j^t + b) + \beta \times (\mathsf{y_i} - \hat{y}_i) \text{ where } \beta = 0 \text{ during the free phase} \end{cases} \tag{A.9}$$

The nudge still derives from the MSE cost function as defined in Eq. A.2. The system also sequentially settles to two fixed steady states s_* and s_*^β at the end of the free and the nudged phase respectively.

A.2.2 . Primitive Function

We define the primitive function as the function from which the dynamics could derive:

$$s^{t+1} \approx \frac{\partial \Phi}{\partial s} \tag{A.10}$$

which gives, ignoring the activation function ρ:

$$\Phi = \frac{1}{2} s^T W s \tag{A.11}$$

A.2.3 . Learning Rule

Similarly to the energy-based settings, we now compute the gradient of the primitive function with regard to a parameter of the system in order to perform optimization. The learning rule, expressed as a function of the two equilibrium points s_* and s_*^β, now reads:

$$\Delta\theta = \frac{1}{\beta}\left(\frac{\partial \Phi}{\partial \theta}(x, s_*^\beta, \theta) - \frac{\partial \Phi}{\partial \theta}(x, s_*, \theta)\right) \tag{A.12}$$

The learning rules for the weights and the biases read:

$$\begin{cases} \Delta W_{ij} = \frac{1}{\beta}(s_{i,*}^\beta s_{j,*}^\beta - s_{i,*} s_{j,*}) \\ \Delta b_i = \frac{1}{\beta}(s_{i,*}^\beta - s_{i,*}) \end{cases} \tag{A.13}$$

Chapter B

Training Convolutional Networks with Equilibrium Propagation

In this section, we describe and define all operations used to train with EP a convolutional neural network recurrently connected with symmetric synapses. We describe the dynamics and the underlying learning rules for the weights and the biases in the prototypical and the energy-based settings. We denote $N^{\rm conv}$ and $N^{\rm fc}$ the number of convolutional layers and fully connected layers in the convolutional system, and $N^{\rm tot} = N^{\rm conv} + N^{\rm fc}$. The units of the system are denoted by s and listed from $s^0 = x$ the input to the output $s^{N^{\rm tot}}$.

B.1 . Operations involved in the convolutional system

We detail here the operations involved in the dynamics of a convolutional RNN in both the prototypical and the energy-based settings.

- The 2-D convolution between w with dimension $(C_{\rm in}, C_{\rm out}, F, F)$ and an input x of dimensions $(C_{\rm in}, H_{\rm in}, S_{\rm in})$ and stride one is a tensor y of size $(C_{\rm out}, H_{\rm out}, W_{\rm out})$ defined by:

$$y_{c,h,s} = (w \star x)_{c,h,s} = B_c + \sum_{i=0}^{C_{\rm in}-1} \sum_{j=0}^{F-1} \sum_{k=0}^{F-1} w_{c,i,j,k} x_{i,j+h,k+s}, \tag{B.1}$$

  where B_c is a channel-wise bias.

- The 2-D transpose convolution of y by $\tilde{w}$ is then defined in this work as the gradient of the 2-D convolution with respect to its input:

$$(\tilde{w} \star y) = \frac{\partial (w \star x)}{\partial x} \cdot y \tag{B.2}$$

- The dot product "$\bullet$" generalized to pairs of tensors of same shape (C, H, S) writes:

$$a \bullet b = \sum_{c=0}^{C-1} \sum_{h=0}^{H-1} \sum_{w=0}^{S-1} a_{c,h,s} b_{c,h,s}. \tag{B.3}$$

- The pooling operation $\mathcal{P}$ with stride F and filter size F of x:

$$\mathcal{P}_F(x)_{c,h,s} = \max_{i,j \in [0, F-1]} \left\{ x_{c, F(h-1)+1+i, F(s-1)+1+j} \right\}, \tag{B.4}$$

  with relative indices of maximums within each pooling zone given by:

$$\mathrm{ind}_{\mathcal{P}}(x)_{c,h,s} = \underset{i,j \in [0, F-1]}{\mathrm{argmax}} \left\{ x_{c, F(h-1)+1+i, F(s-1)+1+j} \right\} = (i^*(x,h), j^*(x,s)). \tag{B.5}$$

- The unpooling operation $\mathcal{P}^{-1}$ of y with indices $\mathrm{ind}_{\mathcal{P}}(x)$ is then defined as:

$$\mathcal{P}^{-1}(y, \mathrm{ind}_{\mathcal{P}}(x))_{c,h,s} = \sum_{i,j} y_{c,i,j} \cdot \delta_{h,F(i-1)+1+i^*(x,h)} \cdot \delta_{sF(j-1)+1+j^*(x,s)}, \tag{B.6}$$

  which consists in filling a tensor with the same dimensions as x with the values of y at the indices $\mathrm{ind}_{\mathcal{P}}(x)$, and zeroes elsewhere. For notational convenience, we omit to write explicitly the dependence on the indices except when appropriate. We can also see unpooling as the gradient of the pooling operation with respect to its input.

- The flattening operation $\mathcal{F}$ is defined as reshaping a tensor of dimensions (C, H, S) to $(1, CHS)$. We denote by $\mathcal{F}^{-1}$ its inverse.

B.2 . Prototypical Settings

B.2.1 . Equations of the dynamics

We derive here the dynamics of the convolutional network with symmetric connections and with the mean square error as loss function in the prototypical settings. In this case, the dynamics reads:

$$\begin{cases} s_{t+1}^{n+1} = \rho\left(\mathcal{P}(w_{n+1} \star s_t^n) + \tilde{w}_{n+2} \star \mathcal{P}^{-1}(s_t^{n+2})\right), & \forall n \in [0, N^{\text{conv}} - 2] \\ s_{t+1}^{N^{\text{conv}}} = \rho\left(\mathcal{P}(w_{N^{\text{conv}}} \star s_t^{N^{\text{conv}}-1}) + \mathcal{F}^{-1}({w_{N^{\text{conv}}+1}}^\top \cdot s_t^{N^{\text{conv}}+1})\right), \\ s_{t+1}^{N^{\text{conv}}+1} = \rho\left(w_{N^{\text{conv}}+1} \cdot \mathcal{F}(s_t^{N^{\text{conv}}}) + {w_{N^{\text{conv}}+2}}^\top \cdot s_t^{N^{\text{conv}}+2}\right), \\ s_{t+1}^{n+1} = \rho\left(w_{n+1} \cdot s_t^n + {w_{n+2}}^\top \cdot s_t^{n+2}\right), & \forall n \in [N^{\text{conv}} + 1, N^{\text{tot}} - 2] \\ s_{t+1}^{N^{\text{tot}}} = \rho\left(w_{N^{\text{tot}}} \cdot s_t^{N^{\text{tot}}-1}\right) + \beta(\mathrm{y} - s_t^{N^{\text{tot}}}), & \text{with } \beta = 0 \text{ during the first phase,} \end{cases} \tag{B.7}$$

where we take the convention $s^0 = x$, the input. In this case, we have $s^{N^{\text{tot}}} = \hat{y}$, the output layer. Considering the function:

$$\begin{aligned} \Phi(x, s^1, \cdots, s^{N^{\text{tot}}}) = & \sum_{n=N_{\text{conv}}+2}^{N_{\text{tot}}-1} {s^{n+1}}^\top \cdot w_n \cdot s^n + s^{N_{\text{conv}}+1} \cdot w_{N_{\text{conv}}+1} \cdot \mathcal{F}(s^{N_{\text{conv}}}) \\ & + \sum_{n=1}^{N_{\text{conv}}-1} s^{n+1} \bullet \mathcal{P}\left(w_{n+1} \star s^n\right) + s^1 \bullet \mathcal{P}\left(w_1 \star x\right), \end{aligned}$$

when ignoring the activation function, we have:

$$\forall n \in [1, N^{\text{tot}}]: \quad s_t^n \approx \frac{\partial \Phi}{\partial s^n}. \tag{B.8}$$

B.2.2 . Learning rules

We derive the learning from the primitive function with the help of Eq. A.12. In the prototypical settings, the learning rules read:

$$
\begin{cases}
\Delta w_1 = \frac{1}{\beta}\left(\mathcal{P}^{-1}(s_*^{1,\beta}) \star x - \mathcal{P}^{-1}(s_*^1) \star x\right) \\
\forall n \in [1, N_{\text{conv}} - 1]: \quad \Delta w_{n+1} = \frac{1}{\beta}\left(\mathcal{P}^{-1}(s_*^{n+1,\beta}) \star s_*^{n,\beta} - \mathcal{P}^{-1}(s_*^{n+1}) \star s_*^n\right) \\
\Delta w_{N_{\text{conv}}+1} = \frac{1}{\beta}\left(s_*^{N_{\text{conv}}+1,\beta} \cdot \mathcal{F}\left(s_*^{N_{\text{conv}},\beta}\right)^\top - s_*^{N_{\text{conv}}+1} \cdot \mathcal{F}\left(s_*^{N_{\text{conv}}}\right)^\top\right) \\
\forall n \in [N_{\text{conv}} + 2, N_{\text{tot}} - 1]: \quad \Delta w_n = \frac{1}{\beta}\left(s_*^{n+1,\beta} \cdot s_*^{n,\beta^\top} - s_*^{n+1} \cdot s_*^{n^\top}\right)
\end{cases}
\tag{B.9}
$$

B.3 . Energy-Based Settings

B.3.1 . Equations of the Dynamics

Inspired by the primitive function derived in the prototypical settings we define an energy function which applies to an energy-based convolutional system, and rely on the same operations defined above:

$$
\begin{aligned}
E(x, s^1, \cdots, s^{N^{\text{tot}}}) = \frac{1}{2}\sum_{n=1}^{N_{tot}}(s^n)^2 - \sum_{n=1}^{N_{\text{tot}}} b_n \rho(s^n) - \frac{1}{2}\sum_{n=N_{\text{conv}}+2}^{N_{tot}-1} \rho(s^{n+1})^T.w_n.\rho(s^n) \\
- \rho(s^{N_{\text{conv}}+1}) \cdot w_{N_{\text{conv}}+1} \cdot \mathcal{F}(\rho(s^{N_{\text{conv}}})) - \sum_{n=1}^{N_{\text{conv}}-1} \rho(s^{n+1}) \bullet \mathcal{P}\left(w_{n+1} \star \rho(s^n)\right) - \rho(s^1) \bullet \mathcal{P}\left(w\right.
\end{aligned}
$$

The dynamics is then derived from this energy function with the help of Eq. 3.16:

$$
\begin{cases}
\frac{\partial s^1}{\partial t} = -s^1 + \frac{\partial \rho(s^1)}{\partial s^1} \times \left(\mathcal{P}(w_1 \star \rho(x)) + \tilde{w}_2 \star \mathcal{P}^{-1}(\rho(s^2))\right), \\
\frac{\partial s^{n+1}}{\partial t} = -s^{n+1} + \frac{\partial \rho(s^{n+1})}{\partial s^{n+1}} \times \left(\mathcal{P}(w_{n+1} \star \rho(s^n)) + \tilde{w}_{n+2} \star \mathcal{P}^{-1}(\rho(s^{n+2}))\right), \qquad \forall n \in [1, N^{\text{conv}} - 2] \\
\frac{\partial s^{N^{\text{conv}}}}{\partial t} = -s^{N^{\text{conv}}} + \frac{\partial \rho(s^{N^{\text{conv}}})}{\partial s^{N^{\text{conv}}}} \times \left(\mathcal{P}(w_{N^{\text{conv}}} \star \rho(s^{N^{\text{conv}}-1})) + \mathcal{F}^{-1}({w_{N^{\text{conv}}+1}}^\top \cdot \rho(s^{N^{\text{conv}}+1}))\right), \\
\frac{\partial s^{N^{\text{conv}}+1}}{\partial t} = -s^{N^{\text{conv}}+1} + \frac{\partial \rho(s^{N^{\text{conv}}+1})}{\partial s^{N^{\text{conv}}+1}} \times \left(w_{N^{\text{conv}}+1} \cdot \mathcal{F}(\rho(s^{N^{\text{conv}}})) + {w_{N^{\text{conv}}+2}}^\top \cdot \rho(s^{N^{\text{conv}}+2})\right), \\
\frac{\partial s^{n+1}}{\partial t} = -s^{n+1} + \frac{\partial \rho(s^{n+1})}{\partial s^{n+1}} \times \left(w_{n+1} \cdot \rho(s^n) + {w_{n+2}}^\top \cdot \rho(s^{n+2})\right), \qquad \forall n \in [N^{\text{conv}} + 1, N^{\text{tot}} - 2] \\
\frac{\partial s^{N^{\text{tot}}}}{\partial t} = -s^{N^{\text{tot}}} + \frac{\partial \rho(s^{N^{\text{tot}}})}{\partial s^{N^{\text{tot}}}} \times \left(w_{N^{\text{tot}}} \cdot \rho(s^{N^{\text{tot}}-1})\right) + \beta(\mathrm{y} - s^{N^{\text{tot}}}), \quad \text{with } \beta = 0 \text{ during the first phase.}
\end{cases}
\tag{B.10}
$$

where again we have $s^{N^{\text{tot}}} = \hat{y}$, the output layer.

B.3.2 . Learning Rules

We derive the learning from the primitive function with the help of Eq. A.7. In the energy-based settings, the learning rules read:

$$
\begin{cases}
\Delta w_1 = \frac{1}{\beta}\left(\mathcal{P}^{-1}(\rho(s_*^{1,\beta})) \star x - \mathcal{P}^{-1}(\rho(s_*^1)) \star x\right) \\
\forall n \in [1, N_{\mathrm{conv}} - 1]: \quad \Delta w_{n+1} = \frac{1}{\beta}\left(\mathcal{P}^{-1}(\rho(s_*^{n+1,\beta})) \star \rho(s_*^{n,\beta}) - \mathcal{P}^{-1}(\rho(s_*^{n+1})) \star \rho(s_*^n)\right) \\
\Delta w_{N_{\mathrm{conv}}+1} = \frac{1}{\beta}\left(\rho(s_*^{N_{\mathrm{conv}}+1,\beta}) \cdot \mathcal{F}\left(\rho(s_*^{N_{\mathrm{conv}},\beta})\right)^\top - \rho(s_*^{N_{\mathrm{conv}}+1}) \cdot \mathcal{F}\left(\rho(s_*^{N_{\mathrm{conv}}})\right)^\top\right) \\
\forall n \in [N_{\mathrm{conv}} + 2, N_{\mathrm{tot}} - 1]: \quad \Delta w_n = \frac{1}{\beta}\left(\rho(s_*^{n+1,\beta}) \cdot \rho(s_*^{n,\beta^\top}) - \rho(s_*^{n+1}) \cdot \rho(s_*^{n^\top})\right)
\end{cases}, \tag{B.11}
$$

One should notice that we only need to store the activation $\rho(s)$ of the neurons to compute the gradient for each parameter which turns out to be very interesting when the activation function ρ outputs binary values, as we do in Section 4.3.

Chapter C

A Scaling Factor for Equilibrium Propagation

In this section, we discuss in detail the scaling factor introduced in Section 4.2. We first describe the initialization of the scaling factor. We then show that a naive initialization for the scaling factor inspired by XNOR-Net leads to good performance, but that tuning more precisely the scaling factor can increase the accuracy. Finally we derive learning rules for the scaling factors allowing EP to optimize by itself the value of the scaling factors. We show that systems learning their scaling factors better fit the training set but also learn faster.

C.1 . Learning the Scaling Factor with EP

Results in the previous subsection show that optimizing the value of α can give rise to enhanced performance. Here we show that this optimization can be achieved through EP. In the context of EP we can indeed derive a learning rule for any parameter in the primitive or energy function. In this section, the scaling factor is first initialized with the method described in Alg. 7 and is then optimized with SGD with the gradient extracted by EP. For clarity, we decompose the binary weights W from $\pm\alpha$ to $\alpha \times w$ where $w = \pm 1$.

C.1.1 . Learning Rules in the Prototypical settings

Fully connected layers architecture.

For a given fully connected layer, the scaling factor α can be introduced in the primitive function of the system as:

$$\Phi(s) = \frac{1}{2}\alpha \times s^T ws \tag{C.1}$$

Eq. A.12 then indicates that the learning rule for the scaling factors in a fully connected architecture in the prototypical settings of EP is:

$$\Delta\alpha_{l,l+1} = \frac{1}{2\beta}\left((s_l^T ws_{l+1})^\beta - (s_l^T ws_{l+1}))^0\right) \tag{C.2}$$

where l denotes the index of a layer in the system.

Convolutional architecture:

The scaling factors in use for the classifier are updated with the gradient given by the learning rule stated above.

For convolutional layers, we use one scaling factor per output feature map which gives C_{out} scaling factors for a layer with C_{out} feature maps.

Thus for each channel in a convolutional layer c in C_{out} we can write:

$$\mathcal{P}\left(W_{n+1} \star s^n\right)_c = \alpha_c \times \mathcal{P}\left(w_{n+1} \star s^n\right)_c \tag{C.3}$$

where $W_{n+1} = \alpha_c \times w_{n+1}$ are the normalized weights for a channel and $w_{n+1} \in \{-1.1\}$. Following this observation, we can also rewrite a primitive function with α as we did for the fully

connected architecture. From this primitive function, we can derive the learning rule for the scaling factors of the convolutional part which reads, channel-wisely:

$$\begin{cases} \forall n \in [1, N_{\text{conv}} - 1] : \quad \Delta\alpha_c^{n+1} = \frac{1}{\beta}((s_c^{n+1} \bullet \mathcal{P}\left(w_{n+1} \star s^n\right)_c)^\beta - (s_c^{n+1} \bullet \mathcal{P}\left(w_{n+1} \star s^n\right)_c)^0) \\ \Delta\alpha_c^1 = \frac{1}{\beta}((s_c^1 \bullet \mathcal{P}\left(w_1 \star x\right)_c)^\beta - (s_c^1 \bullet \mathcal{P}\left(w_1 \star x\right)_c)^0) \end{cases} \tag{C.4}$$

C.1.2 . Learning Rules in the Energy-Based Settings

Fully connected layers architecture:

Similarly to the way we introduced α in the primitive function, we re-write the energy function of a fully connected layers architecture as a function of α:

$$E(s) = \frac{1}{2}\sum_i s_i - \frac{1}{2}\sum_{i \neq j} \alpha_{ij} w_{ij} \rho(s_i)\rho(s_i) - \sum_i b_i \rho(s_i) \tag{C.5}$$

Again, with the help of Eq. A.7 we derive a learning rule for the scaling factors in a fully connected architecture in the energy-based settings of EP which reads as follow:

$$\Delta\alpha_{l,l+1} = \frac{1}{2\beta}\left((\rho(s_l^T) w \rho(s_{l+1}))^\beta - (\rho(s_l^T) w \rho(s_{l+1}))^0\right) \tag{C.6}$$

where l denotes the index of a layer in the system.

Convolutional architecture:

The scaling factors in use for the classifier are updated with the gradient given by the learning rule stated above.

In our convolutional networks, we use one scaling factor per feature map which gives C_{out} scaling factors for a layer with C_{out} feature maps. For each feature map, we have Eq. C.3 verified and we can also easily derive the learning rule of the scaling factors of the convolutional layers which reads, channel-wise:

$$\begin{cases} \forall n \in [1, N_{\text{conv}} - 1] : \quad \Delta\alpha_c^{n+1} = \frac{1}{\beta}((\rho(s_c^{n+1}) \bullet \mathcal{P}\left(W_{n+1} \star \rho(s^n)_c\right))^\beta - (\rho(s_c^{n+1}) \bullet \mathcal{P}\left(W_{n+1} \star \rho(s^n)_c\right)^0)) \\ \Delta\alpha_c^1 = \frac{1}{\beta}((\rho(s_c^1) \bullet \mathcal{P}\left(W_1 \star x\right)_c)^\beta - (\rho(s_c^1) \bullet \mathcal{P}\left(W_1 \star x\right)_c)^0) \end{cases} \tag{C.7}$$

C.2 . Simulations Details - Hyperparameters and Training Curves

C.2.1 . Binary Synapses

We detail in this section all settings and parameters used for the simulations for EP with binary synapses and full-precision activations (hardsigmoid activation function). We ran the simulations with PyTorch and speed them up on a GPU. The duration of the simulations runs from 30 mins for the shallow network to 5 days for the convolutional architecture on CIFAR-10.

For these simulations, we use the prototypical settings of EP for the sake of saving simulation time. The energy-based settings would perform the same way but such models are much longer to train.

We found that comparatively to full-precision models trained by EP, the error signal vanishes through the system and thus deep layers need a greater learning rate for the biases and greater γ for the weights. All hyperparameters are reported in Table C.1.

The target is one-hot encoded and the prediction is computed by taking the argmax of the state of the output neurons. The output layer is designed in a way that we have one output neuron per class of the dataset. We initialize the binary weights taking the sign of randomly-initialized weights matrices.

We choose the sign of beta randomly at each mini-batch which is known to give better results [14], [186]. For all simulations we used mini-batches of size of 64 as we found it performs better.

All figures report the mean of the training and testing errors computed with 5 trials each $\pm$ 1 standard deviation.

MNIST - fully connected layer - 1 hidden layer. We train a network with a fully connected architecture and 1 hidden layer on MNIST. We first tuned the EP hyperparameters (T, K, β) making EP gradient estimates match those given by BPTT [157]. At the same time we tuned BOP hyperparameters in order to fit the flipping metric (Eq. 4.1) in the range leading to successful training as described in Section 4.2. We found that contrarily to Helwegen *et al*. [160], the flipping metric starts at high level (between 0 and $-4/-5$) and decreases over epochs to reach a region below -5.

We initialize one scaling factor per weight matrix with the method described in Alg. 7. When the scaling factors are learnt, we use the same learning rate for all scaling factors. Despite the fact that the learning rule for the scaling factors requires the sign of the weights ± 1 for the computation, we found that using the scaled weights $\pm\alpha$ performs the same way so we used the scaled weight matrix to compute the gradient.

To reach an accuracy at levels of reported results in the literature with such architecture trained by EP on MNIST, we needed to increase by 8 the number of neurons in the hidden layer as shown in Fig. C.1 when the scaling factors are fixed which justifies the architecture we trained: 784-4096-10.

We report all hyperparameters in Table C.1. We initialize the biases with the native PyTorch random initialization and the state of the neurons to zero as it has proven to perform better.

We performed two sets of simulations:

- Simulations where the scaling factors were fixed which achieve accuracy (Table 4.1, Fig. C.2) close to those reported in the EP literature: [157].

- Simulations where the scaling factors were learnt. We show that learning the scaling factors improves by a considerable margin the training -0.7% and the testing -0.4% errors: Fig. C.2, Fig. C.3 and Table 4.2. We link the better testing error to a better fit on the training set as the network seems to overfit a bit: the testing error starts to increase after 10 epochs which also highlights the fact that when we learn the scaling factors, we can use less neurons per hidden layer and still get accuracy close to those reported in the EP literature. We also report that the training is at least five times faster, as after 10 epochs the training and testing errors are below the levels obtained after 50 epochs with fixed scaling factors. Learning the scaling factors makes the flipping metric of BOP to decrease more quickly than when the scaling factors are fixed.

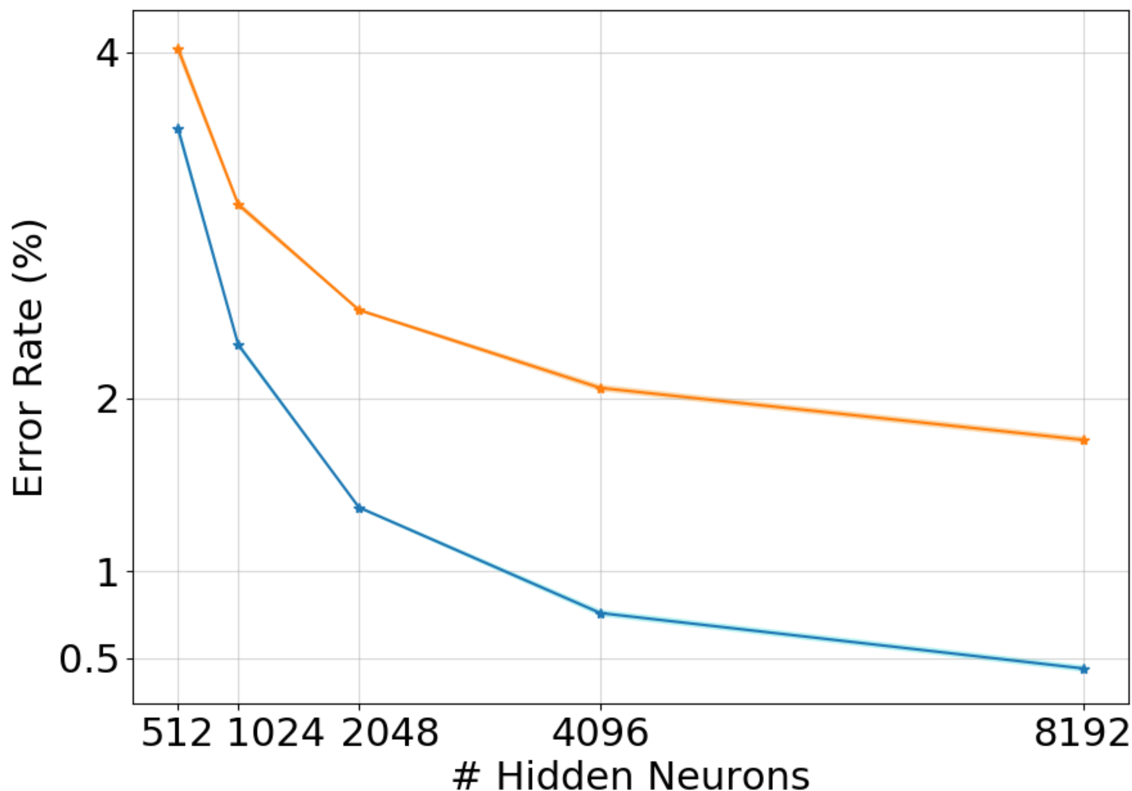


Figure C.1: Averaged train (blue) and test (orange) errors on MNIST with a fully connected architecture with one hidden layers as a function of the number of hidden neurons - We average the errors over 5 trials and plot the average $\pm$ 1 standard deviation

MNIST - fully connected layer - 2 hidden layers. We train a network with a fully connected architecture which has 2 hidden layers on MNIST. We initially chose EP and BOP hyperparameters close to the hyperparameters chosen for training the network with one hidden layer network and then fine-tuned them to achieve the best accuracy. The metric of BOP (Eq. 4.1) also decreases over epochs to reach a level below -5 in the good range for BOP.

Again, we initialize with Alg. 7 one scaling factor per weight matrix which gives 3 scaling factors for this architecture. We also use the same learning rate for all scaling factors and the scaled weights for computing the gradient as done with the architecture which has 1 hidden layer.

We kept the same number of neurons (4096) in each hidden layer as for the architecture which has only 1 hidden layer.

We report all hyperparameters in Table C.1. We initialize the weights with the native PyTorch random initialization and the state of the neurons to zero as it has proven to perform better.

We performed two sets of simulations:

- Simulations where the scaling factors were fixed, which achieve accuracy (Table 4.1, Fig. C.4) close to those reported in the EP literature [157].
- Simulations where the scaling factors were learnt. We show that learning the scaling factors

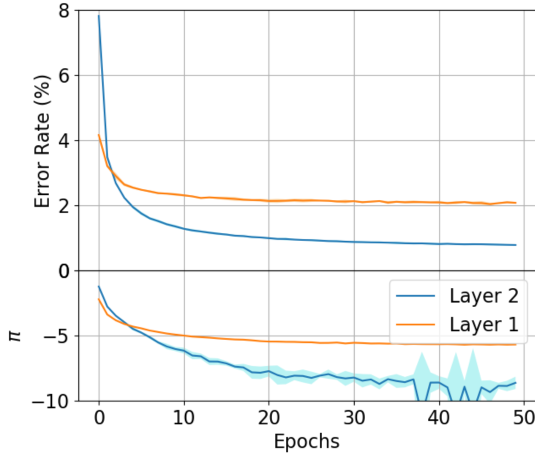


Figure C.2: Top: Train (blue) and test (orange) error on MNIST with a fully connected architecture with one hidden layer of 4096 neurons trained with EP with binary synapses - The scaling factors are fixed - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output

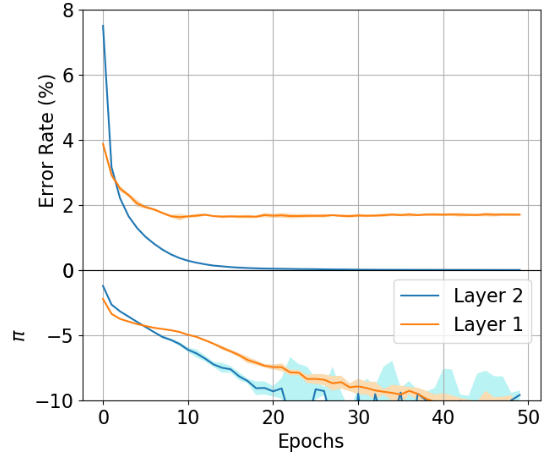


Figure C.3: Top: Train (blue) and test (orange) error on MNIST with a fully connected architecture with one hidden layer of 4096 neurons trained with EP with binary synapses - The scaling factors are learnt - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output

improves a lot the fit by 0.15% on the train set and thus improves the testing accuracy by 0.2% in Fig. C.4, Fig. C.5 and Table 4.2. Finally learning the scaling factors also speed up the training by at least a factor 5.

MNIST - convolutional architecture: We train a convolutional network on MNIST. The architecture used consists in the following: 2 convolutional layers of respectively 32 and 64 channels. We use convolutional kernels of size 5×5, padding of 2 and a stride of 1. Each convolutional operation is followed by a 2 Max Pooling operation with a stride of 2. We flatten the output of the last convolutional layer to feed the output layer of 10 neurons.

We tuned EP hyperparameters (T,K,β) making EP gradient estimates match the gradient given by BPTT [157]. We tuned BOP hyperparameters to make the metric in the range below -5.

The scaling factors α are initialized channel-wise in each convolutional layer which gives 32 scaling factors for the first convolutional layer and 64 scaling factors for the second convolutional layer with the architecture used here. Again we use the scaled weights to compute the gradient of each scaling factor.

We performed two sets of simulations:

- Simulations where the scaling factors were fixed. We report accuracy slightly below the one reported with EP on MNIST with the same convolutional architecture in [157]: -0.4% for the training and -0.2% for the testing errors. Two things one: as underscored before, BOP seems to regularize the training with EP but also we used the sign of β randomly which is known to better estimate the gradient given by EP and thus improve the training.

- Simulations where the scaling factors were learnt. Learning the scaling factors allow the system to better fit the training set (-0.5% of training error). But this makes the system to overfit as the testing error increases to 0.88% after 50 epochs after having reached a minimum at

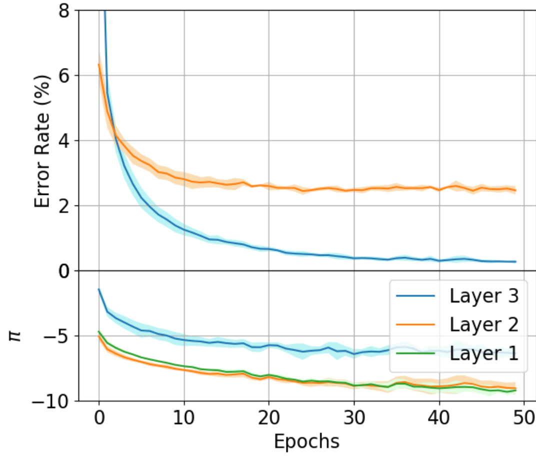


Figure C.4: Top: Train (blue) and test (orange) error on MNIST with a fully connected architecture with two hidden layers of 4096 neurons trained with EP with binary synapses - The scaling factors are fixed - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output

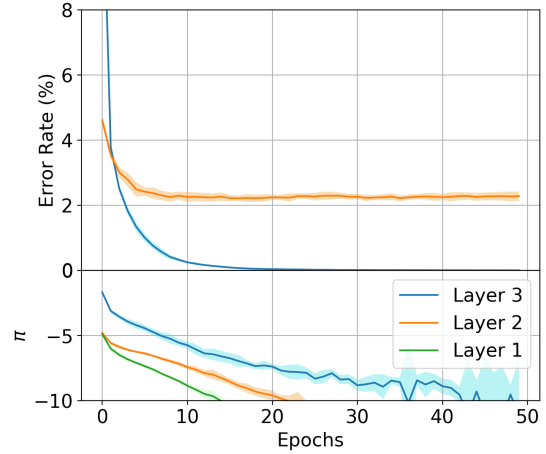


Figure C.5: Top: Train (blue) and test (orange) error on MNIST with a fully connected architecture with two hidden layers of 4096 neurons trained with EP with binary synapses - The scaling factors are learnt - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output

0.76% after 25 epochs. Learning the scaling factors also decreases more the flipping metric π as shown in Fig. C.6, Fig. C.7 and Table 4.3.

Table C.1: Hyperparameters used for training systems with EP and binary synapses - $lrBias$ are the learning rates used for updating the biases with SGD and given from input to output layer - γ is layer-dependent and given from input to output layer

Dataset	Method	Architecture	EP			BOP		
			T	K	β	γ	τ	$lrBia$
MNIST	fc	784-4096-10	50	10	0.3	1e-4-1e-5	5e-7	0.05-0.0
MNIST	fc	784-4096(2)-10	250	10	0.3	2e-5-2e-5-5e-6	5e-7	0.2-0.1-0
MNSIT	conv	1-32-64-fc	150	10	0.3	5e-8	1e-8	0.1-0.05-0
CIFAR-10	conv	3-64-128-256(2)-fc	150	10	0.3	1e-7(2)-2e-7(2)-5e-8	1e-8	0.4-0.2-0.1-0.

CIFAR-10 - convolutional architecture. We train a convolutional network on CIFAR-10. The architecture used consists in the following: 3-64-128-256-256-fc(10): 4 convolutional layers of respectively 64, 128, 256 and 256 channels, one output layer of 10 neurons. We use convolutional kernels of size 5×5, padding of 2 and a stride of 1. Each convolutional operation is followed by a 2 Max Pooling operation with a stride of 2. We flatten the output of the last convolutional layer to feed the output layer.

Because we used twice as less feature maps at each convolutional layer to speed up our training

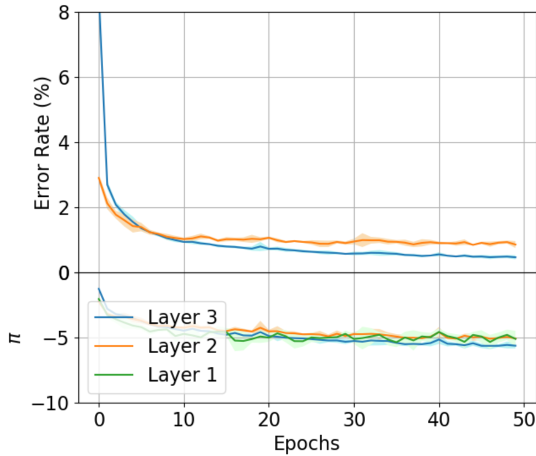


Figure C.6: Top: Train (blue) and test (orange) error on MNIST with a convolutional architecture with 2 convolutional layers of respectively 32 and 64 channels trained with EP with binary synapses - The scaling factors are fixed - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output.

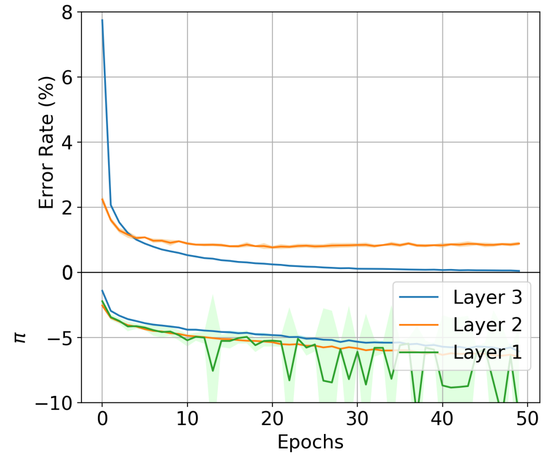


Figure C.7: Top: Train (blue) and test (orange) error on MNIST with a convolutional architecture with 2 convolutional layers of respectively 32 and 64 channels trained with EP with binary synapses - The scaling factors are learnt - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output.

simulations compared to the original network Laborieux *et al*. [186], we used the available code [1] to run simulations with the same architecture as ours to benchmark our technique.

We chose EP hyperparameters equal to those used for the convolutional architecture trained on MNIST as it has shown to work well. We tuned BOP hyperparameters to make the metric in the good range for BOP.

The scaling factors α are initialized channel-wise in each convolutional layer which for instance gives 64 scaling factors for the first convolutional layer with the architecture used here. Again we use the scaled weights to compute the gradient of each scaling factor.

We performed two sets of simulations:

- Simulations where the scaling factors were fixed which achieve accuracy (Table 4.1, Fig. C.8) close to those reported in the EP literature ([186]).
- Simulations where the scaling factors were learnt. We show that learning the scaling factors improves a lot the fit by 1.4% on the train set and thus improves the testing accuracy by 1.2% in Fig. C.8, Fig. C.9 and Table 4.4.

We pre-processed CIFAR-10 with the following data augmentation and normalization techniques before feeding it to the system:

- Random Horizontal Flip with $p = 0.5$
- Random Crop with $padding = 4$

[1] The code is available at: https://github.com/Laborieux-Axel/Equilibrium-Propagation.

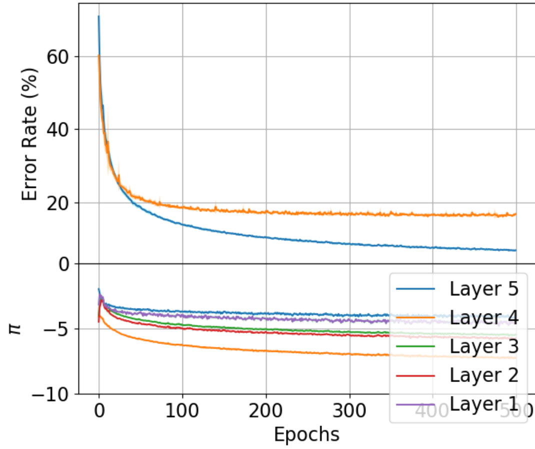


Figure C.8: Top: Train (blue) and test (orange) error on CIFAR-10 with a convolutional architecture with 4 convolutional layers of respectively 64, 128, 256 and 256 channels trained with EP with binary synapses - The scaling factors are fixed - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output.

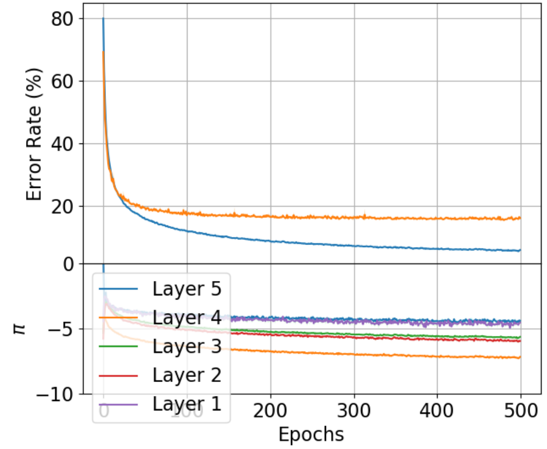


Figure C.9: Top: Train (blue) and test (orange) error on CIFAR-10 with a convolutional architecture with 4 convolutional layers of respectively 64, 128, 256 and 256 channels trained with EP with binary synapses - The scaling factors are learnt - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output.

- Normalization with $\mu = (0.4914, 0.4822, 0.4465)$ and $\sigma = (0.247, 0.243, 0.261)$ for each rgb input channel.

C.2.2 . Binary Synapses and Activations

We detail in this section all settings and parameters used for the simulations of EP with binary synapses and binary activations. We ran the simulations with PyTorch and sped them up on a GPU. For all simulations we used mini-batches of size of 64 as we found it performs better. The time of the simulations runs from 30 mins for the shallow network to 5 days for the convolutional architecture on CIFAR-10.

For the simulations of EP with binary synapses and binary activations we use the energy-based settings of EP with the rules derived in Section 4.3 such as the pseudo-derivative of the Heavyside step function and the enlarged output layer.

In this section, we explore how τ can be finely tuned layer-wise in order to give the best performance while having the same γ for all layers which could be more hardware friendly as we could use the same devices to store the momentum and only change the threshold layer-wise. All hyperparameters are reported in Table C.1.

The target is one-hot encoded and then replicated N_{perclass} times to match the size of the output layer. We make the prediction with the two methods described in 4.3. We initialize the binary weights taking the sign of randomly-initialized weights matrices (native random initialization of Pytorch which is the Uniform Kaiming initialisation).

The input data is kept full-precision thus the MAC operation for the first layer of each architecture is full-precision and also the gradient.

We choose the sign of beta randomly at each mini-batch which is known to give better results

[14], [157], [165] for all simulations except when training the network with the fully connected architecture and which has 2 hidden layers where we only used $\beta > 0$.

We used the Heaviside step function as the binary activation function as emphasised in Section 4.3. For defining the pseudo-derivative function $\hat{\rho}'(s)$ (Eq. 4.8) we used $\sigma = 0.5$ despite using a binary activation.

All figures report the mean of the training and testing errors computed with 5 trials each $\pm$ 1 standard deviation.

MNIST - fully connected layer - 1 hidden layer: We train a network with a fully connected architecture and 1 hidden layer on MNIST.

We chose EP hyperparameters (T, K, β) close to those used for training the same architecture but with binary synapses and full-precision activations. At the same time we tuned BOP hyperparameters in order to fit the flipping metric in the range leading to successful training as described in Section 4.2.

We initialize one scaling factor per weight matrix with the method described in Alg. 7. Simulations with a learnt scaling factors gave results only for 1 hidden layer. When we deepened the network to be trained, learning the scaling factor does not behave well, which we think it is due to the nudging strategy (notably when $\beta < 0$) which does not give an accurate estimation of the gradient.

To reach an accuracy at the level of reported results in the literature with such architecture trained by EP on MNIST, we needed to increase by 16 the number of neurons in the hidden layer which gives the architecture we trained: 784-8192-100. We chose 100 output neurons as it is approximately the number of input units times the sparsity of MNIST data and 100 has shown to perform the best.

We report all hyperparameters in Table C.2. We initialize the biases with the native PyTorch random initialization and the state of the neurons to one as it has proven to perform better.

We performed two sets of simulations:

- Simulations where the scaling factors were fixed which achieve accuracy (Table 4.5, Fig. C.10) close to those reported in the EP literature: [157].
- Simulations where the scaling factors were learnt. We show that learning the scaling factors only improve a little bit the training error: -0.1% but not the testing error: Fig. C.10, Fig. C.11.

MNIST - fully connected layer - 2 hidden layers We train a network with a fully connected architecture and 1 hidden layer on MNIST.

We chose EP hyperparameters (T,K,β) close to those used for training the same architecture but with binary synapses and full-precision activations. At the same time we tuned BOP hyperparameters in order to fit the flipping metric in the range leading to successful trainings as described in Section 4.3.

We initialize one scaling factor per weight matrix with the method described in Alg. 7.

We chose 6000 output neurons as it gives the best accuracy but also as it scales as the number of hidden neurons in the penultimate hidden layer times some sparsity in the layer.

We initially perform a nudging with the sign of β chosen randomly at each mini-batch. But when we nudge the system with $\beta < 0$, it appears that we should let the system evolve during K time steps with K very large (of the order of at least 500 time steps). Finally, we chose to nudge only using the sign of $\beta > 0$ despite the trainings perform less than if we used the sign of beta

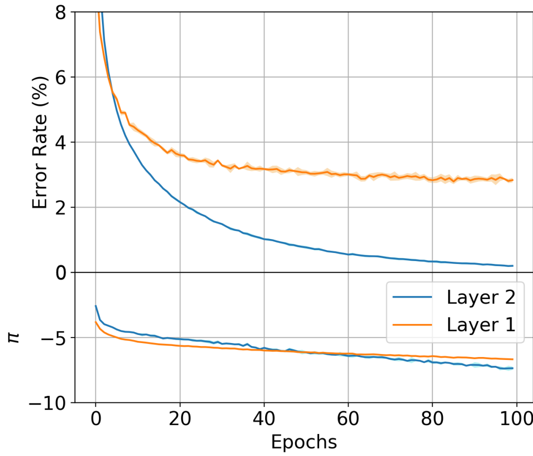


Figure C.10: Top: Train (blue) and test (orange) error on MNIST with a fully connected architecture with one hidden layer of 8192 neurons trained with EP with binary synapses and binary activations - The scaling factors are fixed - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output.

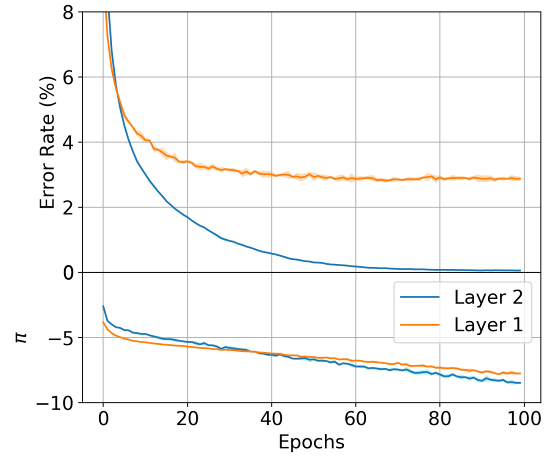


Figure C.11: Top: Train (blue) and test (orange) error on MNIST with a fully connected architecture with one hidden layer of 8192 neurons trained with EP with binary synapses and binary activations - The scaling factor is learnt - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output.

randomly. Monitoring the temporal evolution of some neurons in the network can also help at tuning EP hyperparameters.

To reach an accuracy at levels of reported results in the literature with such architecture trained by EP on MNIST, we used the following architecture we trained: 784-8192-8192-6000. We report all hyperparameters in Table C.2. We initialize the biases with the native PyTorch random initialization and the state of the neurons to one as it has proven to perform better.

We report all hyperparameters in Table C.2. We initialize the biases with the native PyTorch random initialization and the state of the neurons to one as it has proven to perform better.

MNIST - convolutional architecture We train a convolutional network on MNIST. The architecture used consists in the following: 2 convolutional layers of respectively 256 and 512 channels. We use convolutional kernels of size 5×5, padding of 1 and a stride of 1. Each convolutional operation is followed by a 3 Max Pooling operation with a stride of 3. We flatten the output of the last convolutional layer to feed the output layer of 700 neurons.

We tuned BOP hyperparameters to make the metric in the range below -5.

The scaling factors α are initialized channel-wise in each convolutional layer which gives 256 scaling factors for the first convolutional layer and 512 scaling factors for the second convolutional layer with the architecture used here.

Again, learning the scaling factors did not show better accuracy and could be also linked to the nudging strategy.

We initialize the biases at 0 and the state of the neurons to one as it has proven to perform better.

Finally, here we adopted another nudging implementation: although the nudging is usually performed by adding the derivative of the loss function with respect to the units of the output layer

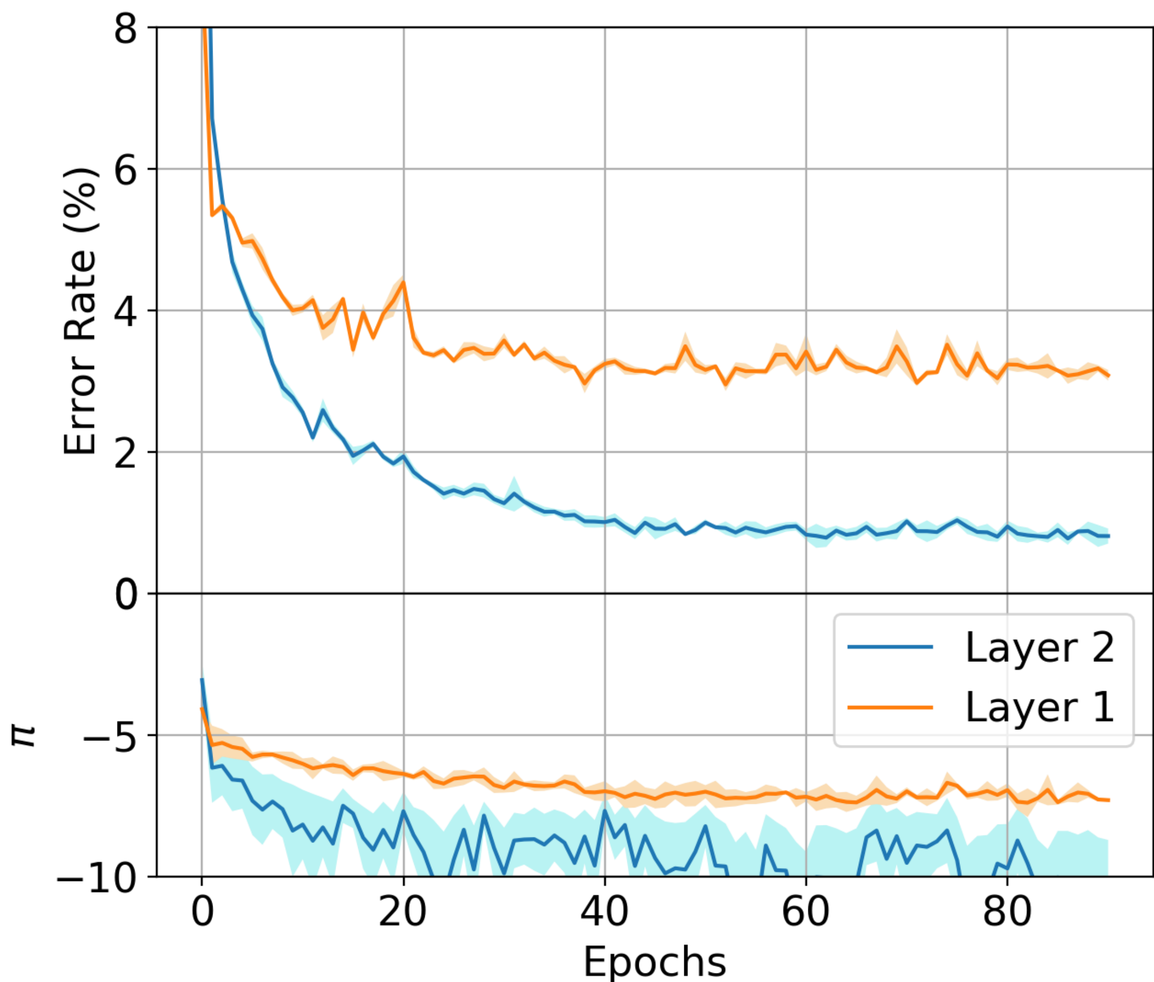


Figure C.12: Top: Train (blue) and test (orange) error on MNIST with a fully connected architecture with two hidden layers of 8192 neurons trained with EP with binary synapses & binary neurons - The scaling factors are fixed - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output.

$+\beta(\mathrm{y}-\hat{y})$, we implemented a constant nudge: $+\beta(\mathrm{y}-\hat{y}_*)$, where $\hat{y}_*$ stands for the first steady state reached by the output units at the end of the first phase. This nudge has shown to perform better than the classic nudge.

We report all hyperparameters in Table C.2. We initialize the biases with the native PyTorch random initialization and the state of the neurons to one as it has proven to perform better.

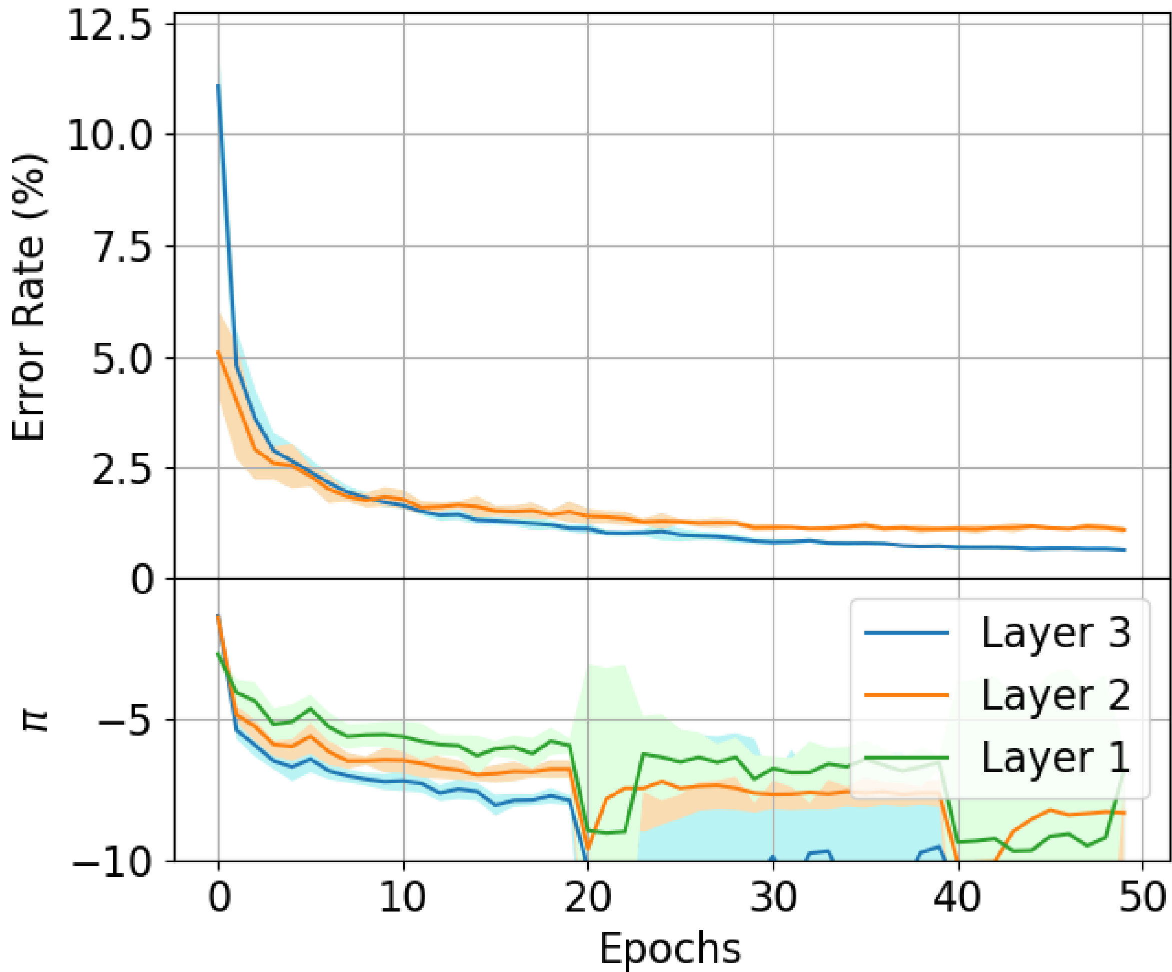


Figure C.13: Top: Train (blue) and test (orange) error on MNIST with a convolutional architecture with 2 convolutional layers of respectively 256 and 512 channels trained with EP with binary synapses & binary neurons - The scaling factors are fixed - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output.

Table C.2: Hyperparameters used for training systems with EP and binary synapses with binary neurons - γ is layer-dependent and given from input to output layer when multiple values are given - γ has the same value for all layers when a single value is given.

			EP			BOP		
Dataset	Method	Architecture	T	K	β	γ	τ	$lrBias$
MNIST	fc	784-8192-100	20	10	2	2e-6	2.5e-7 - 2e-7	1e-7
MNIST	fc	784-8192(2)-8000	30	80	2	1e-6	2e-8 - 1e-8 - 5e-8	1e-6
MNIST	conv	1-256-512-1600(fc)	100	50	1	5e-8	8e-8 - 8e-8 - 2e-7	2e-6 - 5e-6 - 1e-5