Rethinking biologically inspired learning algorithmstowards better credit assignment for on-chip learning

Maxence Ernoult

Thèse de doctorat de Sorbonne Université

École doctorale Physique en Île-de-France (EDPIF)

Centre de Nanosciences et de Nanotechnologies (C2N),
Unité Mixte de Recherche CNRS, Thalès

Rethinking biologically inspired learning algorithms towards better credit assignment for on-chip learning

—

Repenser les algorithmes d'apprentissage inspirés de la biologie pour estimer les gradients par la physique des systèmes neuromorphiques

---

présentée par

Maxence Ernoult

Thèse de doctorat de Physique dirigée par
Julie Grollier et Damien Querlioz

Présentée et soutenue publiquement 30/06/2020 devant le jury composé de:

Pr. Pierre Bessière	Examinateur
Pr. Laurent Daudet	Examinateur
Pr. Julie Grollier	Directrice de thèse
Dr. Timothée Masquelier	Rapporteur
Dr. Teodora Petrisor	Examinatrice
Dr. Damien Querlioz	Invité
Pr. Blake Richards	Rapporteur

À mes parents

Remerciements

Je tiens en tout premier lieu à remercier mes directeurs de thèse Damien Querlioz et Julie Grollier de m'avoir accepté de me recruter sur un domaine de compétences qui n'était pas le mien initialement. Merci Damien pour ton suivi technique extrêmement rigoureux et ton engagement quasiment paternel auprès de ton laboratoire, pendant tes WEs, tard le soir, ou bien entre deux avions. Merci infiniment Julie de m'avoir poussé à travailler avec bienveillance sur ce merveilleux sujet de recherche qu'est Equilibrium Propagation. Ton approche transverse et courageuse du calcul neuromorphique me donne à croire, plus que n'importe quelle autre dans la littérature actuelle, qu'elle est celle qui a le plus de chances d'aboutir à des sytèmes qui passent à l'échelle pour l'apprentissage sur puce. Enfin, merci à vous deux de m'avoir soutenu à fond dans ma collaboration si épanouissante avec Benjamin Scellier.

Je voudrais exprimer ma profonde gratitude à mon ami et collègue Benjamin Scellier, co-inventeur génial de Equilibrium Propagation avec Yoshua Bengio, grâce à qui ma thèse a connu un incroyable tournant. Tu n'imagines pas combien je te suis reconnaissant de m'avoir aussi simplement proposé de travailler avec toi lorsque nous nous sommes rencontrés à Hanovre en décembre 2018 — conférence dont j'ai découvert par hasard l'existence au détour d'un couloir de l'UMΦ. Merci de m'avoir accordé ta confiance, ta gentillesse et ton soutien si précieux à 8 fuseaux horaires de moi pendant une grande partie de notre collaboration. Travailler avec toi est un immense plaisir dont j'ai énormément appris et que j'espère poursuivre aussi longtemps que possible. Si aujourd'hui je fais le choix de poursuivre la recherche, c'est en immense partie grâce à toi.

Je tiens également à remercier Timothée Masquelier et Blake Richards d'avoir accepté de rapporter ma thèse. Vos questions pertinentes et votre analyse détaillée ont clairement témoigné de l'intérêt que vous y avez accordé. Je remercie également les autres membres de mon jury Teodora Petrisor, Pierre Bessière, Laurent Daudet, pour avoir accepté d'évaluer mon travail et pour l'attention que vous y avez portée.

Je voudrais également exprimer ma gratitude à Yoshua Bengio pour le soutien qu'il a apporté aux travaux que j'ai entrepris avec Benjamin, ainsi que pour m'avoir accueilli au Mila en décembre 2019 et intégré à un projet de recherche aux côtés de Blake Richards et de Damjan Kalajdzievski.

Ma thèse n'aurait pas été aussi agréable et épanouissante sans tous les autres membres du C2N et de l'UMΦ.

Du côté du C2N, je voudrais remercier en premier lieu Liza Herrera Diez avec qui j'ai partagé mon bureau pendant la première moitié de ma thèse. Merci pour ta gentillesse, ton soutien et ton écoute si importants pendant les moments difficiles, ainsi que pour les soirées plus légères à base de Lindy Hop et de pizza avec Aloyse !
J'en arrive à mes très (très) chers camarades de bureau qui ont du supporter mes brimades et mes éclats de rire intempestifs au quotidien. Un immense merci à Tifenn Hirtzlin, mon frère de thèse si j'ose dire, qui m'aura accompagné au C2N du début à la fin. Un très grand merci à mon autre frère de thèse, Axel Laborieux avec qui j'ai eu l'immense bonheur de pouvoir travailler sur les questions théoriques liées à l'apprentissage continu dans les réseaux binaires ainsi que sur le passage à l'échelle d'Equilibrium Propagation — nous ne sommes pas encore arrivés au bout de tout ce qu'on peut faire ! Je veux également remercier Kamel Eddine Harabi pour sa bonté de coeur et le regard fraternel qu'il porte sur les personnes qui l'entourent au sein du laboratoire. Merci à Mamour Sall pour sa bonne humeur et sa gentillesse permanente malgré son Paris-Caen aller-retour quotidien pendant plus d'un an et demi (..!) : Spin-Ion sont chanceux de t'avoir parmi eux ! Merci à tous les quatre pour les parties de baby-foot enflammées, moments indispensables de détente et de soutien mutuel.
Malgré son départ inopiné vers les contrées obscures du bitcoin photonique disruptif, je veux sincèrement remercier Bogdan Penkovskyi pour avoir modernisé nos techniques de recherche, notamment en nous incitant à nous mettre à PyTorch et à GitHub. Je suis également reconnaissant envers Damir Vodenicarevic, Christopher Bennett et Adrien Vincent que j'ai eu la chance de connaître sur la fin de leurs thèses de doctorat pour les échanges extrêmement intéressants que nous avons pu avoir ensemble. Merci Damir de m'avoir appris les bases de Linux et de l'écriture de scripts en shell, me permettant ainsi de pouvoir travailler efficacement dès le début de ma thèse ! Je veux aussi remercier Atreya Majumdar et Xing Chen pour leur implication joyeuse au sein du groupe sur les derniers mois de ma thèse. Enfin, j'adresse toute ma sympathie à Dafiné Ravelosona, Philippe Dollfus et Laurie Calvet pour nos interactions quotidiennes.
Je voudrais également chaleureusement remercier Alain Péan et Christophe Chassat pour leur constante disponibilité pour la logistique informatique absolument indispensable pour la bonne tenue de nos recherches. Merci à Lydia Andalon, Carole Bonnot, Laurence Sidibé et Bernadette Laborde pour la grande qualité de leur accompagnement administratif tout au long de ma thèse.

Du côté de l'UMΦ, je voudrais remercier l'ensemble de l'équipe de m'avoir nourri en divers pots de départs ou thèses puisque ce n'est qu'à ces moments précis que je me rendais au labo — corroborant ainsi une théorie du complot longuement ourdie par Florian Godel et Bruno Dlubak *. Plus sérieusement, un immense merci à mes camarades de bureau David Perconte (pour son inimitable sens de l'humour), Mafalda Jotta Garcia (pour son écoute et nos séances

*"Ah t'es là ? Y a un pot aujourd'hui ?"

de natation), Salvatore Mesoraca (for your unfettered support*) et James Boust (lors de son bref, mais heureux, passage en stage).
Je veux exprimer ma grande reconnaissance à toute l'équipe neuromorphique de Julie : Erwann Martin, Jérémie Laydevant, Shuai Li (a.k.a. Jack), Alice Mizrahi, Danijela Markovic, Dédalo Sanz Hernandez, Nathan Leroux, Mathieu Riou, Teodora Petrisor, André Chanthbouala, Juan Trastoy Quintela, Marie Drouhin. J'ai tellement hâte de voir évoluer votre recherche les mois et les années qui viennent : vous participez aux côtés de Julie à révolutionner notre domaine et c'est une chance immense, et pour Julie et pour vous. En particulier, un très grand merci à Erwann et Jérémie avec qui j'ai eu le bonheur de travailler sur l'implémentation matérielle de Equilibrium Propagation. Un petit mot spécial pour Mathieu qui m'a convaincu de venir en thèse avec Julie; merci Mathious par ailleurs pour tous les moments très drôles passés en compagnie de ton acolyte de toujours Philippe Talatchian. Merci Phillou pour ton soutien et pour m'avoir convaincu d'acheter une tour de compétition avec un super GPU rue Montgallet.
Je tiens à exprimer ma reconnaissance à l'ensemble des étudiants et des chercheurs de l'UMΦ qui ont contribué à rendre mon quotidien heureux : Florian Godel, Bruno Dlubak, Hugo Merbouche, Lucile Soumah, Faycal Bouamrane, Denis Crété, Cécile Carrétéro, Sophie Collin, Eliana Recoba Pawlowski, Marie-Blandine Martin, Laurette-Apolline Jerro, Benoit Quinard, Nicolas Reyren, Steffen Wittrock et Victor Zatko. Je voudrais remercier Frédéric Nguyen Van Dau et Frédéric Petroff pour l'ambiance chaleureuse qu'ils ont entretenu au sein de l'UMΦ sous leur direction. Enfin merci à Anne Dussart et Nathalie Lesauvage pour leur grande gentillesse et la qualité de leur suivi administratif tout au long de ma thèse.

Je veux exprimer ma gratitude à tous les enseignants qui m'ont donné le goût de la rigueur et des sciences, et l'envie de faire de la recherche : François Guinard et Christelle Dano au lycée, Bruno Morel et Michel Volcker en prépa, Jean Dalibard, Denis Bernard, Antoine Broaweys, Roland Lehoucq et Henri-Jean Drouhin à l'X. Je remercie tout particulièrement Henri-Jean de m'avoir donné l'opportunité de réaliser mon stage de M1 à Harvard (merci à Federico Capasso de m'y avoir accueilli) puisque c'est cette expériene de recherche qui m'a définitivement conforté dans l'idée de faire une thèse.

Je voudrais maintenant remercier les personnes extérieures au labo qui m'ont été absolument indispensables pour m'amener à faire une thèse puis la mener jusqu'au bout. Un immense merci en premier lieu à mes merveilleux amis de Ginette pour m'avoir constamment et infailliblement soutenu à travers les épreuves depuis 11 ans maintenant: Quentin (pour ton infatigable fidélité et pour l'élévation intellectuelle sans égale que tu apportes aux débats politiques à table), Sophie Marbach (pour tes gâteaux et tes roulés de knackis servis rue de la Huchette), Romain (pour ton éternelle chaleur paternelle), Léiou (pour ton sourire solaire), Billou & Mathou (pour les délicieuses soirées à Alesia), Amaury (parce que tu transes absolument tout), Sylvain (pour m'avoir accompagné à Taizé en juin 2019 et pour tout le reste), Sophie Gros (pour toutes les soirées raclette organisées chez vous), Mathieu (pour supporter Quentin au quotidien) et Florian Le Roux (reviens en France b****l). Egalement

*"Maxence, you're STUPID!"

de Ginette, un grand merci à Yann Hicke, Alexandre Krajenbrink, Maxime Déporte, Pierre Laurent, Victor Cluzel, Sébastien Mannai et Luc Lambert pour leur amitié toutes ces années passées. Je remercie également mes chers amis de l'X : Florian Feppon, Camille Gillot, Grégoire Dequidt, Benjamin Charbaut, Nicole Lindsay, Frédérique Robin, Ruben Zakine, Pierre-Philippe Crépin, Guillaume Magnien et Sébastien Demortain. Un petit mot en particulier pour mon cher ami Florian qui m'a accompagné à la piscine plusieurs midis par semaine pendant toute la thèse et dont l'immense talent scientifique n'a d'égale que sa gentillesse et son humilité absolument exemplaire. Un grand merci à mes deux camarades de Cambridge Antoine Chevenet, Adam Foster, Anika Krause, Lea Volke et Pim de Haan pour leur soutien malgré la distance géographique. Enfin je remercie également Guillaume Aoust, Andrés Ritter, Blanche Magarinos, Arthur Surzur, Thierry de Fougerolles, Garance de Turenne, Sophie Bardet et Victor David pour leur fidélité amicale pendant toute la thèse.

Un immense merci à ma Constance pour son merveilleux soutien pendant les derniers mètres de la thèse qui n'auraient certainement pas été aussi productifs autrement, tout particulièrement dans le contexte de la crise sanitaire. Enfin merci à mes parents et ma soeur qui m'ont absolument tout donné, en respectant et en accompagnant tous mes choix avec amour, et sans qui je ne serais jamais parvenu à être là où je suis aujourd'hui.

Main table of contents

Detailed table of contents

Résumé de la thèse

Motivation de la thèse

Autour de l'année 2012, l'apprentissage profond (ou "deep learning") s'est imposé à l'ensemble de la société en apportant des solutions à beaucoup de problèmes tels que la classification et la génération automatique d'images et de discours, le traitement automatique du langage naturel, le raisonnement et les jeux vidéos. La persévérance des pionniers du domaine — Geoffrey Hinton, Yoshua Bengio et Yann Lecun — a été récompensée par le prix Turing 2018. Historiquement, l'apprentissage profond est le fruit de nombreuses idées provenant des neurosciences computationnelles, des mathématiques et de l'optimisation. Les réseaux de neurones artificiels ont réellement connu leur essor lorsque les réseaux de neurones dits "profonds" — avec beaucoup de couches modélisant l'organisation hiérarchique des aires du cerveau — ont pu être entraînés sur de grandes bases de données grâce à l'utilisation des cartes graphiques ("Graphical Processing Units" ou "GPUs") afin de réaliser les calculs requis de façon très efficace.

Aller au delà des capacités des GPUs pour l'entraînement des réseaux de neurones profonds est la motivation principale de cette thèse de doctorat. En effet, les ordinateurs tels que nous les connaissons aujourd'hui sont confrontés à deux limitations technologiques fondamentales. D'une part, les progrès réalisés sur les architectures d'ordinateurs sur les dernières décennies s'appuient sur le paradigme de Von Neumann selon lequel la mémoire et le processeur sont physiquement séparés, entraînant ainsi une très forte consommation énergétique pour transporter les données entre les deux. D'autre part, la loi de Moore prédisant une diminution exponentielle de la taille des transistors a atteint une limite physique au delà de laquelle l'état physique du transistor ne peut plus être fiable de façon déterministe.

Une approche possible pour surmonter ces limitations technologiques est le calcul neuromorphique, proposée pour la première fois par Carver Mead et consistant à repenser l'ordinateur à partir de zéro en imitant les caractéristiques du cerveau. Bien que la recherche en calcul neuromorphique ait longtemps été conduite sur des technologies CMOS, il y a eu ces dernières années un attrait grandissant pour les technologies analogiques de taille

nanométrique pour réaliser des neurones et des synapses artificielles. En particulier les *mémoires résistives* ou *memristors*, qui peuvent stocker des valeurs de poids synaptiques sous forme de d'états de conductance, sont des candidats extrêmement prometteurs pour réaliser des synapses artificielles. Pour entraîner des réseaux de neurones physiques composés de memristors, une approche possible est de déterminer la valeur numérique des conductances des memristors en dehors de la puce à l'aide de simulations menées sur un ordinateur, puis d'importer physiquement ces valeurs sur les memristors à l'aide de protocoles de programmation précis et élaborés. Une autre approche excitante pour entraîner des réseaux de neurones physiques composés de memristors serait de réaliser l'entraînement *directement sur la puce* : un tel dispositif pourrait réaliser tout à la fois l'inférence, le calcul du gradient et la mise à jour correspondante des conductance des memristors.

Cependant, l'apprentissage sur puce est extrêmement difficile pour deux raisons. Tout d'abord, le calcul du gradient de la fonction objectif d'apprentissage appelle à première vue à l'utilisation de l'algorithme de rétropropagation du gradient, plus connu sous le nom de "backpropagation". Cet algorithme d'apprentissage est le plus utilisé pour entraîner les réseaux de neurones profonds. Néanmoins, la loi d'apprentissage prescrite par l'algorithme de backpropagation pour une synapse donnée n'est pas spatialement locale : l'incrément de poids synaptique à appliquer ne dépend pas seulement des deux neurones adjacents à la synapse considérée, rendant ainsi difficile son implémentation sur une puce neuromorphique. Les approches existantes sont parvenues à implémenter des lois d'apprentissage Hebbiennes telles que la plasticité fonction du temps d'occurrence des impulsions (plus couramment connue sous sa version anglaise comme "Spike Timing Dependent Plasticity", ou "STDP") en utilisant des memristors. En dépit de l'élégance de cette approche, ce type de loi d'apprentissage ne passe pas à l'échelle sur des réseaux de neurones plus profonds pour traiter des tâches plus compliquées, très certainement en raison de leur manque de garanties théoriques quant à l'optimisation d'une métrique d'apprentissage. Le second défi de l'apprentissage sur puce est l'incrément de condutance à réaliser étant donnée une valeur de gradient. Les memristors présentent de nombreuses imperfections : ils sont sujets à la non-linéarité, à une gamme de conductance restreinte, à une variabilité intrinsèque et leurs propriétés peuvent beaucoup varier d'un composant à un autre. Ces irrégularités physiques entravent considérablement l'apprentissage sur puce.

Dans cette thèse, nous proposons de démêler ces deux aspects de l'apprentissage sur puce — le calcul du gradient et l'incrément de conductance — sur deux algorithmes d'apprentissage biologiquement inspirés. D'une part, nous étudions l'effet des imperfections des memristors sur l'apprentissage des *Machines de Boltzmann Restreintes*, et proposons des stratégies de programmation adaptées. D'autre part, nous développons *Equilibrium Propagation*, un équivalent de l'algorithme de backpropagation dont la règle d'apprentissage, calculée par la physique du système lui-même, est spatialement locale et mathématiquement fondée. Plus précisément, les contributions de cette thèse sont les suivantes:

- Dans la partie II, nous étudions empiriquement l'utilisation de mémoires résistives dans différentes variantes de Machines de Boltzmann Restreintes. Nous proposons différentes stratégies de programmation pour atténuer l'effet de la non-linéarité, de la variabilité d'un cycle de programmation à un autre et celle d'un composant à un autres sur l'apprentissage. Nous proposons également une technique qui évite de devoir ajuster précisément le temps de programmation des composants.

- Dans la partie IV, nous proposons une version en temps discret d'Equilibrium Propagation, et nous montrons qu'elle est équivalente à l'algorithme de rétropropagation du gradient à travers le temps — plus couramment connu comme *Backpropagation Through Time* (BPTT) — sur des réseaux de neurones récurrents convergents qui recoivent une entrée statique et atteignent un état stationaire. Cette version de l'Equilibrium Propagation permet d'accélérer l'entraînement d'un facteur 5 à 8 en comparaison à sa version originale ainsi que d'entraîner une architecture convolutionnelle pour la première fois. Nous obtenons ainsi la meilleure performance jamais réalisée sur la reconnaissance de chiffres manuscrits (la base de données MNIST) dans la littérature existante sur Equilibrium Propagation.

- Enfin dans la partie V, nous proposons une version de Equilibrium Propagation plus adaptée à une implémentation neuromorphique, que nous appelons *Continual Equilibrium Propagation* (C-EP) où les synapses et les neurones évoluent dynamiquement au cours de la seconde phase de l'algorithme. De cette façon, la loi d'apprentissage de l'Equilibrium Propagation devient locale en temps. Nous montrons que dans la limite asymptotiquement lente de changement de poids synaptiques, C-EP est équivalent à BPTT. Nous étendons C-EP à une situation plus biologiquement réaliste où les connections synaptiques sont asymétriques. Enfin, nous montrons empiriquement que plus un modèle satisfait le théorème d'équivalence entre C-EP et BPTT *avant l'apprentissage*, meilleure est la performance de ce modèle après entraînement par C-EP. Ces résultats pourraient servir de guide pour l'implémentation neuromorphique de Equilibrium Propagation.

Résumé de la partie II

Dans cette partie, nous étudions la composante de l'apprentissage sur puce qui concerne l'incrément de conductance dans plusieurs variantes de Machines de Boltzmann Restreintes.

Avec des valeurs typiques d'imperfections de composants pour la non-linéarité, la variabilité d'un cycle de programmation à un autre et d'un composant à un autre, nous montrons que la Machine de Boltzmann Restreinte Discriminative ("Discriminative RBM" en anglais) est la meilleure architecture candidate en termes de performance à l'entraînement sur la reconnaissance de chiffres manuscrits (la base de données MNIST). Notamment, nos simulations

mettent en évidence comment les imperfections agissent sur le temps de programmation optimal des composants : la non-linéarité sélectionne des temps de programmation courts et à l'inverse, la variabilité d'un cycle de programmation à un autre des temps de programmation plus longs. De façon importante, une pile de Machine de Boltzmann Restreintes constituées de memristors, lorsque chacune des Machines de Boltzmann Restreintes sont apprises séparément et successivement ("greedy learning" en anglais), ne bénéficie pas de la profondeur du réseau de neurones résultant pendant l'apprentissage. Au contraire, les effets des imperfections des composants se cumulent lorsque les caractéristiques extraites par une Machine de Boltzmann Restreinte sont transmises à la suivante dans la pile. Cette limitation vient du faire de ne pas transmettre les signaux d'erreur d'une Machine de Boltzmann Restreinte à une autre en utilisant l'algorithme de backpropagation, afin de préserver la localité de la loi d'apprentissage employée.

Nous montrons également que moyenner sur différents exemples et différentes réalisations stochastiques la loi d'apprentissage fournie par la technique de Contrastive Divergence améliore considérablement la résilience des Machines de Boltzmann Restreintes Discriminatives vis à vis des imperfections des composants. Puisque cette technique sélectionne des temps de programmation plus courts, elle atténue à la fois les effets de la non-linéarité et ceux de variabilité. Nous proposons également l'utilisation de l'algorithme de "Resilient Propagation" (RProp) afin de faciliter l'ajustement du temps de programmation des composants. Nous montrons que RProp n'affecte pas la résilience des Machines de Boltzmann Restreintes Discriminatives aux défauts des composants, obéit à une logique très simple et permet d'élargir considérablement la gamme de temps de programmation des composants jusqu'à deux ordres de grandeur.

En conclusion, cette étude propose des stratégies pour faciliter l'implémentation neuromorphique de Machines de Boltzmann Restreintes pour l'apprentissage sur puce avec des composants memristifs réalistes, proposant ainsi de résoudre l'un des défis principaux de l'apprentissage dans les dispositifs embarqués.

Résumé de la partie IV

Dans cette partie, nous nous concentrons sur la composante de l'apprentissage qui concerne le calcul du gradient de la fonction objectif de l'apprentissage avec Equilibrium Propagation.

Nous proposons une version en temps discret de cet algorithme : dans ce contexte, la version originale de l'algorithme en temps continu peut être vue comme un choix particulier de fonction primitive Φ pour la dynamique du système. Nous montrons que notre version

en temps discret de Equilibrium Propagation est équivalente à l'algorithme de Backpropagation Through Time (BPTT) si le Jacobien de la dynamique est symmétrique (ce qui est équivalent à l'existence d'une fonction primitive pour la dynamique) et l'equilibre est atteint à la fin de la première phase. Plus précisément, les incréments synaptiques calculés *au cours du temps par la dynamique du système* pendant la deuxième phase de Equilibrium Propagation sont égaux, à chaque instant, aux gradients de la fonction objectif de l'apprentissage par rapport aux poids synaptiques calculés par BPTT en *remontant artificiellement dans le temps* de la première phase. Nous appelons cette propriété "Gradient Descending Updates" (GDU) et nous la vérifions numériquement sur deux classes de modèles: les modèles à base d'énergie et les modèles prototypiques. Après avoir défini théoriquement les architectures entièrement connectées pour ces deux classes de modèles, nous montrons que la propriété GDU est de façon générale très bien satisfaite numériquement. De façon plus quantitative, en utilisant une métrique d'erreur quadratique moyenne relative ("Relative Mean Squared Error" ou "RelMSE" en anglais), nous montrons que plus le réseau est profond, plus la RelMSE est grande, suggérant ainsi que l'entraînement d'architectures profondes par Equilibrium Propagation est difficile. Enfin, nous proposons un modèle convolutionnel décrit dans le cadre prototypique préalablement introduit et entraînable par Equilibrium Propagation. Nous montrons que cette architecture satisfait également bien la propriété GDU and réalise la meilleure performance sur MNIST jamais rapportée dans la littérature de Equilibrium Propagation ($\sim 1\%$ d'erreur sur l'ensemble de test). Nous montrons que l'utilisation de notre cadre prototypique permet d'accélérer l'entraînement par Equilibrium Propagation d'un facteur 5 à 8 comparé au cadre à base d'énergie.

Ce travail facilite la conception de modèles de réseaux de neurones entraînable par Equilibrium Propagation, tant par l'usage pratique du Théorème 4 que par l'accélération à l'entraînement apporté par le cadre prototypique, un aspect intéressant pour l'implémentation de Equilibrium Propagation sur des dispositifs neuromorphiques. Ces résultats rapproche Equilibrium Propagation de l'apprentissage automatique conventionnel et pourraient aider à faire passer Equilibrium Propagation à l'échelle pour résoudre des problèmes plus compliqués.

Résumé de la partie V

Enfin dans cette partie, nous étendons l'étude de la partie IV à une situation plus proche d'une implémentation neuromorphique où la loi d'apprentissage prescrite par Equilibrium Propagation devient locale en temps.

Dans cette nouvelle version d'Equilibrium Propagation que nous appelons *Continual Equilibrium Propagation* (C-EP), les synapses évoluent dynamiquement en même temps que les neurones pendant la deuxième phase de l'Equilibrium Propagation. Nous montrons que le théorème précédemment introduit dans la partie IV peut être étendu à ce cadre : dans

la limite de changement synaptiques asymptotiquement lents, la propriété GDD ("Gradient Descending Dynamics") est satisfaite (Theorem 10). Nous montrons que la propriété GDD est satisfaite sur différents modèles et montrons les résultats d'entraînement par C-EP d'un modèle prototypique avec des connexions synaptiques asymmétriques sur MNIST, bénéficiant ainsi que l'accélération à l'entraînement évoquée dans la partie précédente. Nous observons une légère dégradation de performance à l'entraînement comparé à la version originale de Equilibrium Propagation dont nous pouvons directement rendre compte : la vitesse d'apprentissage ("learning rate" en anglais) doit être assez petit afin que la propriété GDD soit assez suffisamment bien satisfaite, mais pas trop afin que la convergence ait lieu en un nombre d'époques raisonnable. Nous étendons l'entraînement par C-EP à des réseaux de neurones dont les connexions synaptiques sont asymmétriques et appelons cette version de l'algorithme *Continual Vector Field* Equilibrium Propagation (C-VF). Nous montrons que C-VF parvient à entraîner des réseaux de neurones avec des connexions synaptiques asymmétriques sur MNIST. De plus nous montrons que, étant donné un modèle, "plus" le théorème 10 est satisfait *avant l'entraînement* (en termes d'angle entre l'incrément de poids synaptique total sur l'ensemble de la seconde phase de C-EP et le gradient négatif fourni par BPTT), meilleure est la performance de ce modèle après entraînement par C-EP.

Ce travail rapproche Equilibrium Propagation des contraintes hardware et de la biologie : C-VF peut être vu comme un équivalent, en fréquence, de la Spike Timing Dependent Plasticity (STDP). De nouveau, Theorem 10 fournit ici une démarche qui peut aider le déploiement d'Equilibrium Propagation sur des systèmes neuromorphiques.

Introduction

It is not until 2012 that the deep learning approach to artificial intelligence took upon the whole society by producing solutions to many problems as image and speech classification and generation, language processing and translation, reasoning and game playing. The pain-stacking perseverance of the pioneers of this field - Geoffrey Hinton, Yoshua Bengio and Yann Le Cun - has been rewarded by the 2018 Turing Prize. Historically, deep learning is the result of long standing ideas coming from computational neurosciences, mathematics and optimization. Artificial neural networks became significantly successful when "deep" networks - with many layers, modelling hierarchical brain regions - could be trained on large datasets thanks to the use of Graphical Computing Units (GPUs) to carry out the required computations very efficiently.

Going beyond the capability of the GPUs for deep neural network training is the core motivation of this PhD thesis. Today's modern computer architectures face two limitations. First, progress in computer architectures over the past decades has built upon the von Neumann paradigm where memory and computation are physically separated, entailing tremendous energy consumption costs to route data in between. Second, Moore's law predicting ever shrinking transistors has come to a physical limit beyond which the transistor state can no longer be deterministically reliable.

One possible approach to overcome these limitations was ushered in by Carver Mead as neuromorphic computing, which consists in rethinking the computer from scratch by mimicking brain features. Although neuromorphic computing research has long been conducted for CMOS-based engines, there has been a growing move towards novel nanometric analog substrates to emulate neurons and synapses. Among them, *resistive memories* or *memristors* which can store a weight value as a conductance state are promising artificial synapse candidates. For memristor-based hardware neural networks, one possible path for training is to find the conductance values offline on software, and subsequently import these values onto the memristor with precises conductance tuning protocols. A very appealing approach would be *on-chip learning*: the chip could sustain inference, gradient computation and subsequent conductance update altogether.

Nonetheless, on-chip learning is extremely challenging for two reasons. First, computing the gradient value of an objective function, a problem often called *credit assignment*, would call at first sight for the use of *backpropagation*, the most widely used learning algorithm for deep neural networks. However, the learning rule prescribed by backpropagation for a given synapse is not spatially local: the weight update to be performed does not solely depend on the two adjacent neurons, therefore creating a bottleneck on a chip. Existing approaches have successfully employed Hebbian event-based learning rule like *Spike Timing Dependent Plasticity* (STDP) along with the use of memristive devices, but scale poorly to deeper architectures, possibly because of their lack of theoretical guarantees. The second challenge of on-chip learning is the conductance update to be performed given a gradient value. Memristors are very imperfect devices subject to non-linearity, finite conductance range, cycle-to-cycle and device-to-device variability. These imperfections significantly hamper on-chip learning.

In this thesis, we propose to disentangle these two aspects of on-chip learning - gradient computation and conductance update - on two different biologically inspired algorithms. On the one hand, we study the effect of memristive device imperfections on the training of *Restricted Boltzmann Machines*, and propose appropriate programming strategies. On the other hand, we build upon *Equilibrium Propagation*, a hardware friendly counterpart of backpropagation whose learning rule, computed by the physics of the system itself, is spatially local. More precisely, the key contributions of this thesis are the following:

- We investigate empirically the use of resistive memories in different variants of Restricted Boltzmann Machines. We propose programming strategies to alleviate the effect of non-linearity, cycle-to-cycle and device-to-device variability on training and remove the need to tune precisely the programming pulse width.
- We propose a discrete-time version of Equilibrium Propagation, which we show to be equivalent to Backpropagation, or more precisely *Backpropagation Through Time* (BPTT) in the context of convergent recurrent neural networks that receive a static input and settle to a steady state. This version of Equilibrium Propagation enables to speed up training simulations by a factor 5 to 8 compared to the original version, and to train the first convolutional architecture, yielding the best training accuracy on MNIST ever reported in the literature of the algorithm.
- We propose a hardware-friendly version of Equilibrium Propagation, which we call *Continual Equilibrium Propagation* (C-EP), where the synapses and the neurons evolve both dynamically throughout the second phase. In this way, the learning rule of Equilibrium Propagation becomes local in time. We show that in the limit of slow synaptic changes, C-EP is also equivalent to BPTT. We extend C-EP to the case where the connections are asymmetric. Finally, we show empirically that the best a model satisfies the theorem *before training*, the best the resulting training performance with C-EP. These results can provide a guidance for the neuromorphic engineering of Equilibrium Propagation.

Part I

Deep learning, neuromorphic computing and on-chip learning

Chapter 1

What is "learning" in neural networks?

The aim of this chapter is to introduce in historical order the elemental concepts and techniques that paved the way towards modern artificial neural networks that are widely used today. With these elements in hand, we will formally define supervised learning and introduce *backpropagation*, the most popular learning algorithm used in deep neural networks. Finally, we will highlight the limit of conventional computers with regards to learning deep neural networks and therefore motivate the approach of neuromorphic computing.

1.1 Artificial neural networks: from unheard to overhyped

Artificial Intelligence had long been envisioned, back to 1950, when Turing suggested that "what we want[ed] [was] a machine that can learn from experience" [1], introducing the very notorious notion of Turing test whose goal is to distinguish, out of a conversation sample, a machine from a human being. The first theoretical attempt to model a biological neuron dates back to 1943 and was laid out by Walter Pitts and Warren McCulloch [2]. Since their model directly takes inspiration from a biological neuron, the very first question we want to address here is: *what* is a neuron? The following description does not aim to be holistic and rather focuses on the most important features of a neuron to be taken into account *computationally speaking*: a real neuron is a huge machinery, which relies on extremely complex biophysical processes mediated by thousands of neurotransmitters!

This being said, the working logics of a single neuron can be depicted as follows. A neuron is an electrically excitable cell that communicates through electrical spikes with other cells through connections called synapses. A neuron may or may not transmit an electrical stimulation to the surrounding neurons it is connected to based on the following principle.

First, a neuron receives each electrical input from other neurons through *dendrites*. Second, a weighted sum of these electrical inputs is integrated within the *soma*, whose effect is to increase the *membrane potential* of the neuron. More precisely, the membrane potential is created by a difference of concentrations of ions Cl^-, K^+ and Na^+ between the interior and exterior of a biological membrane, thereby balancing at equilibrium the diffusive motion of ions across the membrane. Chemical gates called ion channels selectively allow ions to pass through the membrane and can be activated by neurotransmitters. Neurotransmitters that activate Na^+ ion channels will contribute to increase the membrane potential, thereby creating excitatory post-synaptic potentials (EPSPs) and conversely neurotransmitters activating Cl^- or K^+ create inhibitory post-synaptic potentials (IPSPs) which contribute to decrease the membrane potential. When ion channels of different types are simultaneously activated, these effects cumulate.

This description is sufficient to understand the model of Pitts and McCulloch. Let us denote $\{x_k\}_{k\in[1,N]}$ input neurons and $\hat{y}$ the output neuron under consideration. In their model, input neurons can only take binary values: $x_k \in \{0,1\}$. Among the input neurons, we assume there exists one inhibitor neuron denoted x_i such that if it is activated ($x_i = 1$), the output neuron is inhibited ($\hat{y} = 0$). Based on cumulated binary inputs, the value of the output neuron is also binary and defined as:

$$\hat{y} = \sigma\left(\sum_{k=1}^{N} x_k\right) = \begin{cases} 1 & \text{if} \quad \sum_{k=1}^{N} x_k > \nu \quad \text{and} \quad x_i = 0 \\ 0 & \text{otherwise,} \end{cases} \tag{1.1}$$

where ν is the activation threshold of the neuron. Generally speaking, the function σ appearing in Eq. (1.1) is called an *activation function*, as it decides whether the neuron is activated or not given incoming stimuli. In their paper, McCulloch and Pitts show that their neuron can emulate 'AND'/'OR'/'NOT' logical gates if ν is chosen properly.

However, one aspect that the McCulloch-Pitts neuron does not model is how neurons are connected to each other, and how strong should be such a connection, which calls for the notion of a *synapse*. The *weight* of a synapse encodes its strength: it can be seen as the conductivity of the synaptic cleft separating the axon of an input neuron and the dendrite of an output neuron. In the context of deep learning, the terms "weight" and "synapse" are often used equivalently. In the model we have just described, all connections between inputs neurons x_k and the output neuron $\hat{y}$ are implicitly taken to be uniformly equal to 1. Also, the logical functions the McCulloch-Pitts neuron can implement requires the adjustable parameters of the model to be hand-coded rather than automatically adjusted, or "learnt", from data: this aspect is crucial and will be further discussed when defining precisely the notion of "learning" in our context of study. So the question raised here boils down to: how should synaptic connections be modelled and how should they be changed given a stimuli

input? Donald Hebb was one of the first to suggest a famous heuristic about the way synaptic weights should be changed based on stimuli patterns [3]: *"When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased"*; also rephrased by Karla Shatz [4]: *"cells that fire together, wire together"*. More explicitly, if we change the neuron response described by Eq. (1.1) into:

$$\hat{y} = \sum_i w_i x_i, \tag{1.2}$$

where w_i denotes the weight of the synapse connecting the input neuron x_i to the output neuron $\hat{y}$. Assuming the data set $\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(M)}, y^{(M)})\}$ where $x^{(k)}$ and $y^{(k)}$ are binary, Hebb's rule prescribes to set the weight value of each synapse w_k as:

$$w_k = \frac{1}{M} \sum_{m=1}^{M} x_k^{(m)} y^{(m)}. \tag{1.3}$$

Instead, if the data samples of $\mathcal{D}$ are presented sequentially, the m^{th} synapse update Δw_k between two data samples reads:

$$\Delta w_k = \frac{1}{M} x_k^{(m)} y^{(m)}. \tag{1.4}$$

Eq. (1.4), which prescribes a synapse update between two data samples (or batch of samples in more realistic situations), is more generally called a *learning rule*. The expected effect of this particular learning rule is that the output neuron pattern $y^{(m)}$ becomes activated whenever the pattern $x^{(m)}$ is presented to the input neurons, so that $x^{(m)}$ and $y^{(m)}$ become "associated".

Along the lines of the first theoretical intuitions brought by the McCulloch-Pitts neuron and Hebb's rule, the real milestone that ushered in the deep learning era was the development of the *perceptron* by Frank Rosenblatt in 1957 at the Cornell Aeronautical Laboratory [5]. Rosenblatt relaxed some of the stringent assumptions of the McCulloch-Pitts neuron: no absolute inhibition rule applies and importantly the neurons can be real-valued or negative. Overall, Rosenblatt neuron's model reads:

$$\hat{y} = \sigma\left(\sum_{k=1}^{N} w_k x_k + b\right) = \begin{cases} 1 & \text{if} \quad \displaystyle\sum_{k=1}^{N} w_k x_k + b > 0 \\ 0 & \text{otherwise,} \end{cases} \tag{1.5}$$

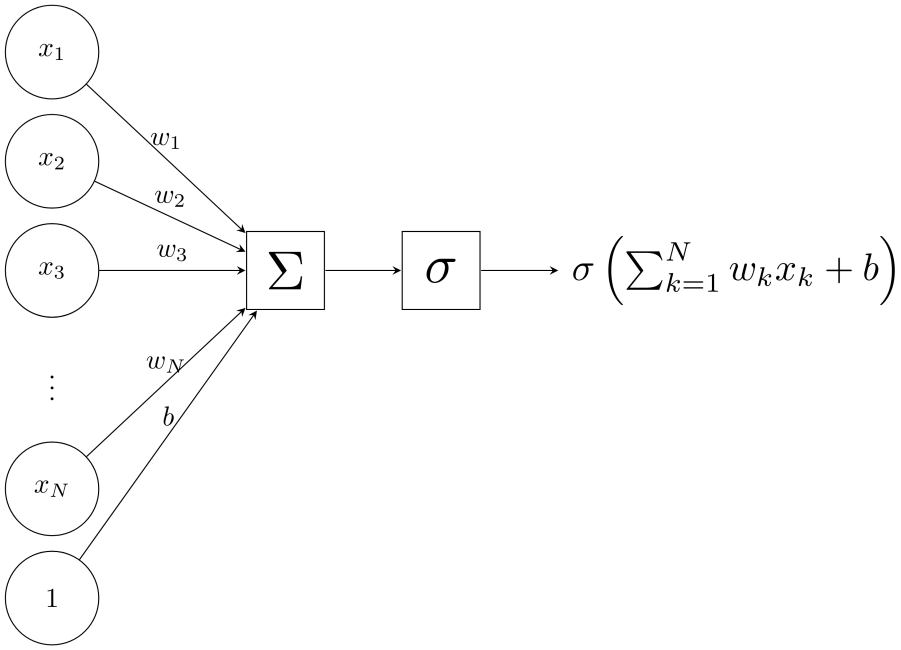

Figure 1.1: **Rosenblatt's perceptron**. By definition, the perceptron first performs a sum of the inputs $x_1, x_2, \cdots x_N$ weighted by the value of the synaptic weights (Σ node), then applies a non linearity (σ node), resulting in Eq. (1.5)

where again w_k stands for the weights of the synapses, and b is an adjustable parameter called a *bias* - which can be regarded as an extra synapse connected to a neuron that is constantly equal to one. In this case, the activation function σ is the Heaviside function. Rosenblatt's perceptron model is depicted on Fig. 1.1.

Along with this new neuron model, the major achievement of Rosenblatt has been to propose a supervised learning algorithm that enables to find the weights w_i and bias b that best account for a given data set. More precisely, let us consider a data set $\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(M)}, y^{(M)})\}$ where $x^{(m)}$ is a real-valued vector and $y^{(m)}$ its associated binary label ($y^{(m)} \in \{0, 1\}$). Let $\mathcal{D}_0 = \{x^{(m)} \in \mathcal{D} | y^{(m)} = 0\}$ and $\mathcal{D}_1 = \{x^{(m)} \in \mathcal{D} | y^{(m)} = 1\}$ be the two classes of the problem. The two classes $\mathcal{D}_0$ and $\mathcal{D}_1$ are said to be *linearly separable* if they can be separated by an hyperplane. With these definitions in hand, Rosenblatt's learning algorithm determines the weights of a perceptron defined per Eq. (1.5) so that the output of the perceptron correctly classifies two linearly separable classes: $\hat{y}(x^{(m)}) = y^{(m)} \quad \forall k \in [1, M]$. The algorithm proceeds intuitively. Given a sample $x^{(m)}$ presented to the input neurons, the perceptron outputs $\hat{y}(x^{(m)})$ given by Eq. (1.5), which is compared to the ground-truth label $y^{(m)}$. If $\hat{y}(x^{(m)}) < y^{(m)}$ (the output is too low), then any weight w_i connected to a positive (resp. negative) input $x_i^{(m)}$ should be increased (resp. decreased). Conversely if $\hat{y}(x^{(m)}) > y^{(m)}$ (the output is too high), then any weight w_i connected to a positive (resp. negative) input $x_i^{(m)}$ should be decreased (resp. increased). More formally, the weight update prescribed reads:

$$\Delta w_i \propto (y^{(m)} - \hat{y}^{(m)}) \cdot x_i^{(m)}. \tag{1.6}$$

In spite of the incredible success of the perceptron model which later inspired the Adaline algorithm [6], its downfall was in great part overshadowed by Eq. (1.5). Indeed, the boundary decision of the perceptron is given by the equation $\sum_{k=1}^{N} w_k x_k + b = 0$, which is that of an hyperplane. This amounts to say that Rosenblatt's perceptron model can only handle separable classification problems, in particular it cannot solve the XOR logical function which cannot be separated linearly. Also as the algorithm stands, it was unclear how it would extend to a *multi-layered* perceptron whose definition changes Eq. (1.5) into:

$$\hat{y} = \sigma(w^{N-1} \cdots \sigma(w^1 \cdot \sigma(w^0 \cdot x + b^0) + b^1) \cdots + b^{N-1}), \tag{1.7}$$

where the intermediate neuron values preceding the final output are called *hidden neurons*, as only the output value is actually observed. Since these values are "hidden", what should then be compared against so that the subsequent weight change on the adjacent synapses improves the output value? Moreover, the computation time required to learn even very simple functions with perceptron-based learning is very long. All these major limitations of the perceptron were pointed out by Marvin Minsky in 1969 [7], whose paper arguably downcasted the research endeavors of the then emerging neural network community and would freeze the funding of the field for the next ten years, a period also known as the AI winter.

It was not until the 90s that the neural network approach to AI really experienced a revival with the invention of the *backpropagation* training algorithm - the next section will be dedicated to describe this algorithm technically. In its Ph.D thesis drafted while at Harvard in 1974, Paul Werbos was the first to suggest the use of backpropagation to train neural networks [8]. It later inspired the development of various theoretical frameworks for backpropagation in neural networks, by David Parker [9] in his master thesis at the Massachusetts Institute Technology and Yann LeCun [10] independently, in 1985. Once the hardware resources available at this time enabled to run simulations fast enough, the definite kick-off of backpropagation in neural networks happened when, in 1987, Geoffrey Hinton and David E. Rumelhart experimentally demonstrated the relevance of the features extracted by hidden units when training a multilayer perceptron to learn non-linear tasks such as the XOR function [11], thereby addressing one of the fundamental objections of Minsky that had caused the demise of neural networks until then.

Another very significant breakthrough is the invention of *Convolutional Neural Networks*, a neural network architecture inspired from the primary visual cortex which was first known

as the Neocogitron [12] in 1980. Yann LeCun later applied backpropagation to train convolutional architectures on document recognition in 1998, a technique soon-to-be deployed commercially to read several million checks per day [13]. Finally, while the use backpropagation would enable to explore new neural network topologies such as *Recurrent Neural Networks* and *Long-Short Term Memory* [14] and variants to learn from temporal data, the tremendous hype that deep learning would definitely gain came from the AlexNet Convolutional Architecture [15] which won the ImageNet Large Scale Visual Recognition Challenge in 2012 and popularized the use of *Graphical Processing Units* (GPUs) to train deep neural networks.

1.2 Learning in neural networks

1.2.1 Definition of the problem

In this subsection, we introduce more formally the backpropagation learning algorithm mentionned previously. Let us define a neural network as a mapping $\mathcal{F}(\cdot;\theta) : x \to \hat{y}$, where x denotes the (data) input and $\hat{y}$ the output of the neural network. Typically, x can represent static data (for image classification for example) or be a stream of data $x = x_0, x_1, \cdots, x_T$ (for language processing for example), we will cover both cases in this section. $\theta = \{\theta_0, \theta_1, \cdots, \theta_{N-1}\}$ stand for the weights, and $\mathcal{F}$ reads as a composition of N functions F^n:

$$\mathcal{F} = F^{(N-1)} \circ F^{(N-2)} \circ \cdots \circ F^{(0)}, \tag{1.8}$$

where each function $F^{(n)}$ is parametrized by θ^n: $F^{(n)}(\cdot) = F^{(n)}(\cdot;\theta^n)$. Depending on the context, n can be a spatial or a temporal label: computation is either carried out *through space* (from one layer to another) or *through time* (from a time step to the next one) or both, depending on the task being solved. Typically, the $F^{(n)}$ function are the composition of a matrix multiplication (or a convolution) and of a non-linear function, or a non parametric operation as an average or max pooling. Also, although $F^{(n)}$ is often a deterministic function, it can also be stochastic, as it is the case in generative models like Boltzmann Machines as we will see later.

In this setting, we want to find the parameters, or "weights" θ which, given any input x of a given dataset, make the prediction of the neural network $\hat{y}(x)$ the closest to the target y associated with x. Generally, the target is a label (for instance, the target y provides the label 'cat' when the data input x is the picture of a cat) or the target can be the data itself, as it is the case in generative models like Boltzmann Machines covered in this thesis. In the former case, the learning setting is said to be "supervised" and in the latter case "unsupervised" -

intermediate situations where only a part of the data is labelled pertains to "semi-supervised" learning. For any purpose, the model builds an estimate of the target from the data denoted $\hat{y}(x)$ that is compared against the ground-truth target y, thereby motivating the following mathematical formulation of the problem:

$$\min_{\theta} \mathcal{L}(\hat{y}(x), y; \theta) \tag{1.9}$$

where we call $\mathcal{L}$ the *loss function* of the problem. In general, the minimum of $\mathcal{L}$ is not analytically tractable. So, in practice, we proceed numerically by using *gradient descent*: iteratively, for a given parameter θ, we compute the steep for the loss function $\mathcal{L}$ and update θ in the direction of decreasing loss, i.e.:

$$\theta \leftarrow \theta - \alpha \frac{\partial \mathcal{L}(x, \theta)}{\partial \theta}, \tag{1.10}$$

where α is a scalar coefficient called the *learning rate* which controls the "speed" of the weight update. Iterating Eq. (1.10) many times, we may reach an optimum θ^* such that $\frac{\partial \mathcal{L}(x,\theta^*)}{\partial \theta} \sim 0$ - see Fig. 1.2 for a cartoon illustration. Very importantly, θ^* may not be a global optimum since $\mathcal{L}$ is generally not a convex landscape. There is a vast literature of optimization techniques used for deep learning that enable to convey inertia to gradient descent so that local optima can be overshot, the most famous and widely used of those being the Adam optimizer [16]. With this terminology and notations in hand, **learning** from this prospective implies two components that are fundamental in the scope of this thesis:

- **Computing the gradient of the loss**: $\frac{\partial \mathcal{L}(x,\theta)}{\partial \theta}$.
- **Updating the weight value**: $\theta \leftarrow \theta - \alpha \frac{\partial \mathcal{L}(x,\theta)}{\partial \theta}$.

In order to go further in the details concerning the computation of the gradient, an important notion when dealing with neural networks is the notion of graphs. Eq. (1.8) can be seen as a directed graph where each leaf node represents input data or parameter, directed edges as operations, and their output nodes at the results of these operations. More often than not, the neural network as it is conventionally represented graphically is the computational graph at use. So from a mathematical viewpoint, we may use "neural network" or "computational graph" equivalently. In other contexts though, it is important to distinguish the neural network itself from the computation it carries out. To describe in more details the computation of the gradients of the loss, we will consider two particular topologies of interest in this thesis.

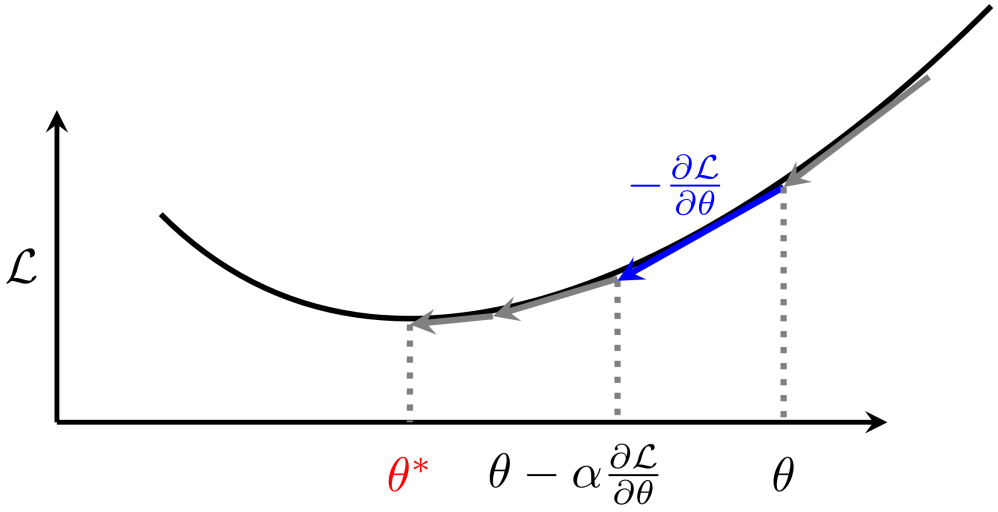

Figure 1.2: **Learning**. Learning proceeds iteratively by computing the gradient of the loss function $\frac{\partial \mathcal{L}}{\partial \theta}$ for a given θ and by subsequently updating the weight value $\theta \leftarrow \theta - \alpha \frac{\partial \mathcal{L}}{\partial \theta}$, until convergence to an optimum θ^*.

1.2.2 Backpropagation in a feedforward neural network

General derivation. Let us consider a special case where the topology of the computation in Eq. (1.8) is described as:

$$\begin{cases} s^1 &= F^{(0)}(x;\theta^0) \\ s^n &= F^{(n-1)}(s^{n-1};\theta^{n-1}) \qquad \forall n \in [2,N] \ , \\ \hat{y} &= F^{(N-1)}(s^{N-1};\theta^{N-1}), \end{cases} \tag{1.11}$$

where the intermediate values of the computation denoted s^n are called 'hidden layers'. In this particular case, the input data is presented once at the first computation of Eq. (1.11) and subsequent computations are only carried over hidden layers. The larger the number of operations N, the "deeper" the neural network.

Note from Eq. (1.11) that the model estimate of the target $\hat{y}$ is computed at the very end of the graph. This aspect is crucial and drives the intuition of the backpropagation algorithm whose goal is to compute the gradients of $\mathcal{L}$ with respect to each weight $\theta^0, \theta^1, \cdots, \theta^{N-1}$. However, $\mathcal{L}$ as it appears in Eq. (1.9) only depends explicitly on $\hat{y}$ and implicitly on the weights since $\hat{y}$ is a function of $\theta^0, \theta^1, \cdots, \theta^{N-1}$ through Eqs. (1.11). The intuition of backpropagation is to start with $\frac{\partial \mathcal{L}}{\partial \hat{y}}$ which is the simplest derivative to compute since $\mathcal{L}$ depends explicitly on $\hat{y}$. Then, we compute the derivative of the variables *preceding* $\hat{y}$ in the neural network by using the *chain-rule of differentiation*: this computation therefore propagates $\frac{\partial \mathcal{L}}{\partial \hat{y}}$ *backward* through the neural network, hence the name "backpropagation".

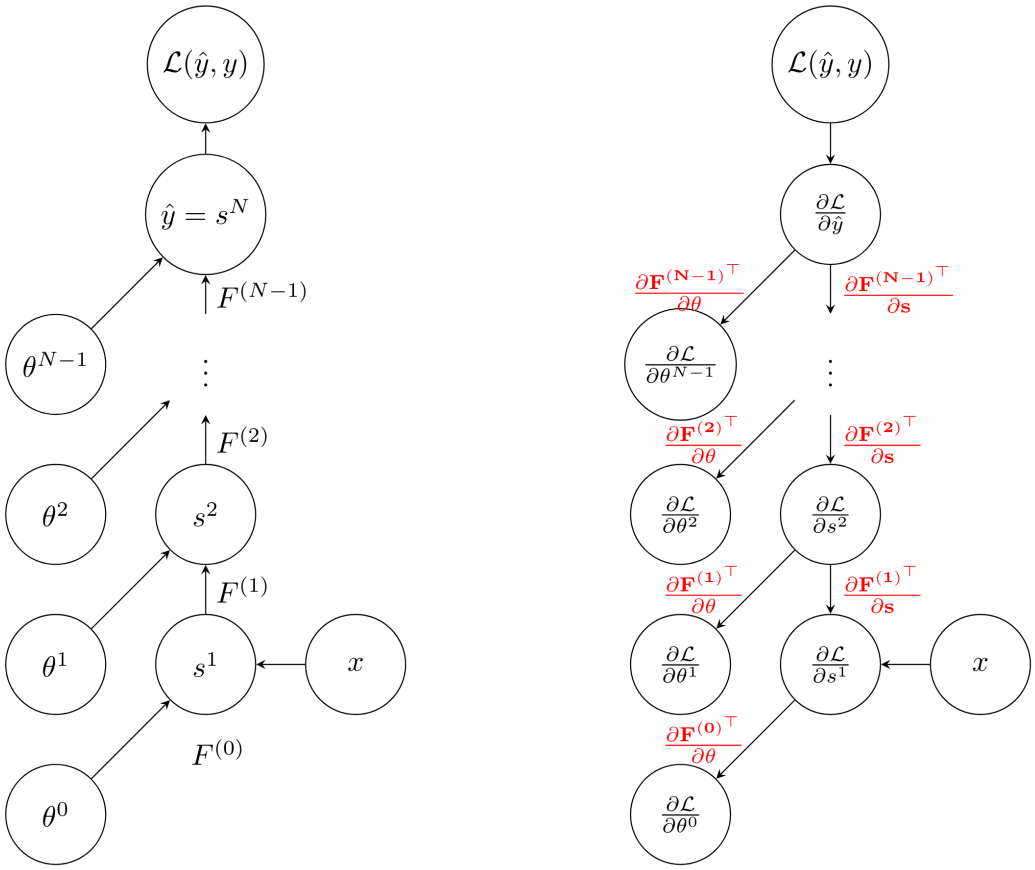

Figure 1.3: **Backpropagation in a feedforward architecture**. For any graph, each leaf node labels an input, an arrow a computation, and a child node the outcome of a computation. **Left**: forward pass, going from input x at the very bottom to $\mathcal{L}$ at the very top. At each forward computational step, $F^{(n)}$ takes θ^n and s^n as inputs and outputs s^{n+1}, until giving $\hat{y}$, and the subsequent loss $\mathcal{L}$. **Right**: backward pass, going backward from $\frac{\partial \mathcal{L}}{\partial \hat{y}}$ at the top to leaves $\frac{\partial \mathcal{L}}{\partial \theta^n}$. At each backward computational step, the Jacobian $\frac{\partial F^{(n)}{}^\top}{\partial s}$ takes $\frac{\partial \mathcal{L}}{\partial s^{n+1}}$ as an input and outputs $\frac{\partial \mathcal{L}}{\partial s^n}$. Similarly, the Jacobian $\frac{\partial F^{(n)}{}^\top}{\partial \theta}$ takes $\frac{\partial \mathcal{L}}{\partial s^{n+1}}$ as an input and outputs $\frac{\partial \mathcal{L}}{\partial \theta^n}$.

More concretely, let us compute $\frac{\partial \mathcal{L}}{\partial \theta^{N-1}}$. Since we have $\hat{y} = F^{(N-1)}(s^{N-1}; \theta^{N-1})$, we can easily compute $\frac{\partial \mathcal{L}}{\partial \theta^{N-1}}$ as a function of $\frac{\partial \mathcal{L}}{\partial \hat{y}}$. Namely, considering the i-th component of $\frac{\partial \mathcal{L}}{\partial \theta^{N-1}}$ and applying the chain-rule, we get:

$$\frac{\partial \mathcal{L}}{\partial \theta_i^{N-1}} = \sum_j \frac{\partial \hat{y}_j}{\partial \theta_i^{N-1}} \frac{\partial \mathcal{L}}{\partial \hat{y}_j},$$

which gives in a vectorized fashion, assuming the convention that $\frac{\partial \mathcal{L}}{\partial u}$ is a column vector for any column vector u:

$$\frac{\partial \mathcal{L}}{\partial \theta^{N-1}} = \left(\frac{\partial \hat{y}}{\partial \theta}\right)^\top \cdot \frac{\partial \mathcal{L}}{\partial \hat{y}}.$$

Using $\hat{y} = F^{(N-1)}(s^{N-1}; \theta^{N-1})$, we get:

$$\frac{\partial \mathcal{L}}{\partial \theta^{N-1}} = \left(\frac{\partial F^{(N-1)}}{\partial \theta^{N-1}}(s^{N-1}; \theta)\right)^\top \cdot \frac{\partial \mathcal{L}}{\partial \hat{y}}. \tag{1.12}$$

In order to compute $\frac{\partial \mathcal{L}}{\partial \theta^{N-2}}$, we first need to backpropagate $\frac{\partial \mathcal{L}}{\partial \hat{y}}$ to s^{N-1}, then to θ^{N-2}. Namely, we first compute $\frac{\partial \mathcal{L}}{\partial s^{N-1}}$:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial s^{N-1}} &= \left(\frac{\partial \hat{y}}{\partial s^{N-1}}\right)^\top \cdot \frac{\partial \mathcal{L}}{\partial \hat{y}} \\ &= \left(\frac{\partial F^{(N-1)}}{\partial s}(s^{N-1}; \theta^{N-1})\right)^\top \cdot \frac{\partial \mathcal{L}}{\partial \hat{y}},\end{aligned}$$

so that we can subsequently compute $\frac{\partial \mathcal{L}}{\partial \theta^{N-2}}$ as:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \theta^{N-2}} &= \left(\frac{\partial s^{N-1}}{\partial \theta^{N-2}}\right)^\top \cdot \frac{\partial \mathcal{L}}{\partial s^{N-1}} \\ &= \left(\frac{\partial F^{(N-2)}}{\partial \theta}(s^{N-2}; \theta^{N-2})\right)^\top \cdot \frac{\partial \mathcal{L}}{\partial s^{N-1}}.\end{aligned}$$

Backpropagation computation can be readily generalized for any variable in the network, proceeding recursively and backward from the output layer, with the following recursive equations:

$$\boxed{\begin{cases} \frac{\partial \mathcal{L}}{\partial s^N} &= \frac{\partial l}{\partial s}(\hat{y}, y) \\ \frac{\partial \mathcal{L}}{\partial s^{N-n}} &= \left(\frac{\partial F^{(N-n)}}{\partial s}(s^{N-n}; \theta^{N-n})\right)^\top \cdot \frac{\partial \mathcal{L}}{\partial s^{N-n+1}} \qquad \forall n > 1 \\ \frac{\partial \mathcal{L}}{\partial \theta^{N-n}} &= \left(\frac{\partial F^{(N-n)}}{\partial \theta}(s^{N-n}; \theta^{N-n})\right)^\top \cdot \frac{\partial \mathcal{L}}{\partial s^{N-n+1}} \qquad \forall n > 1 \end{cases}} \tag{1.13}$$

The computation of Eq. (1.13) can be conveniently depicted with a computational graph - see Fig. 1.3.

Example. For concreteness and as a particular case of Eq. (1.11), let us apply backpropagation the following neural network:

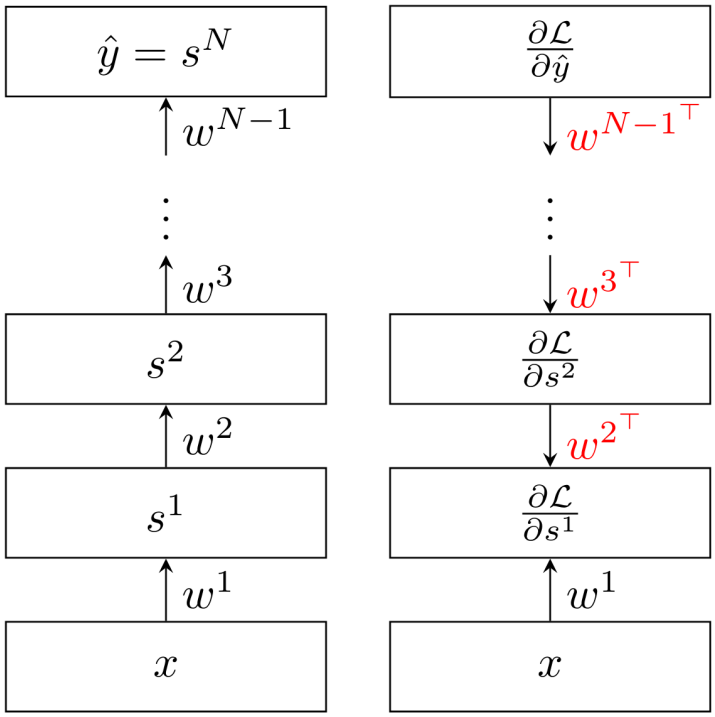

Figure 1.4: **Backpropagation in a feedforward neural network**. **Left**: forward pass, going from input x at the very bottom to $\hat{y}$ at the very top. s^n are commonly called "hidden layers". Each hidden layer value s^n is determined by the previous one s^{n-1}. **Right**: backward pass, going backward from $\frac{\partial \mathcal{L}}{\partial \hat{y}}$. At each backward computational step, the error signals are routed by ${w^n}^\top$, an important feature of backpropagation in neural networks.

$$\begin{cases} s^1 &= \sigma(w^0 \cdot x + b^0) \\ s^n &= \sigma(w^{n-1} \cdot s^{n-1} + b^{n-1}) \quad \forall n \in [2, N] \\ \hat{y} &= \sigma(w^{N-1} \cdot s^{N-1} + b^{N-1}). \end{cases} \tag{1.14}$$

Applying Eq. (1.13) to Eq. (1.14) yields:

$$\begin{cases} \frac{\partial \mathcal{L}}{\partial s^N} &= \frac{\partial l}{\partial \hat{y}}(\hat{y}, y) \\ \frac{\partial \mathcal{L}}{\partial s^{N-n}} &= {w^{N-n}}^\top \cdot \left(\sigma'(w^{N-n} \cdot s^{N-n}) \odot \frac{\partial \mathcal{L}}{\partial s^{N-n+1}}\right) \qquad \forall n > 1 \\ \frac{\partial \mathcal{L}}{\partial w^{N-n}} &= \left(\sigma'(w^{N-n} \cdot s^{N-n}) \odot \frac{\partial \mathcal{L}}{\partial s^{N-n+1}}\right) \cdot {s^{N-n}}^\top \qquad \forall n > 1, \end{cases} \tag{1.15}$$

where $\odot$ denotes element-wise (also known as Hadamard) product between two matrices. The computation of Eq. (1.15) is depicted on Fig. 1.4. For the remainder of this introduction part, the reader should bear in mind that Eq. (1.15) and Fig. 1.4 exhibit explicit features that will later be commented upon when dealing with biologically plausible approaches to learning in chapter 4 of this part.

1.2.3 Backpropagation *through time* in a recurrent neural network

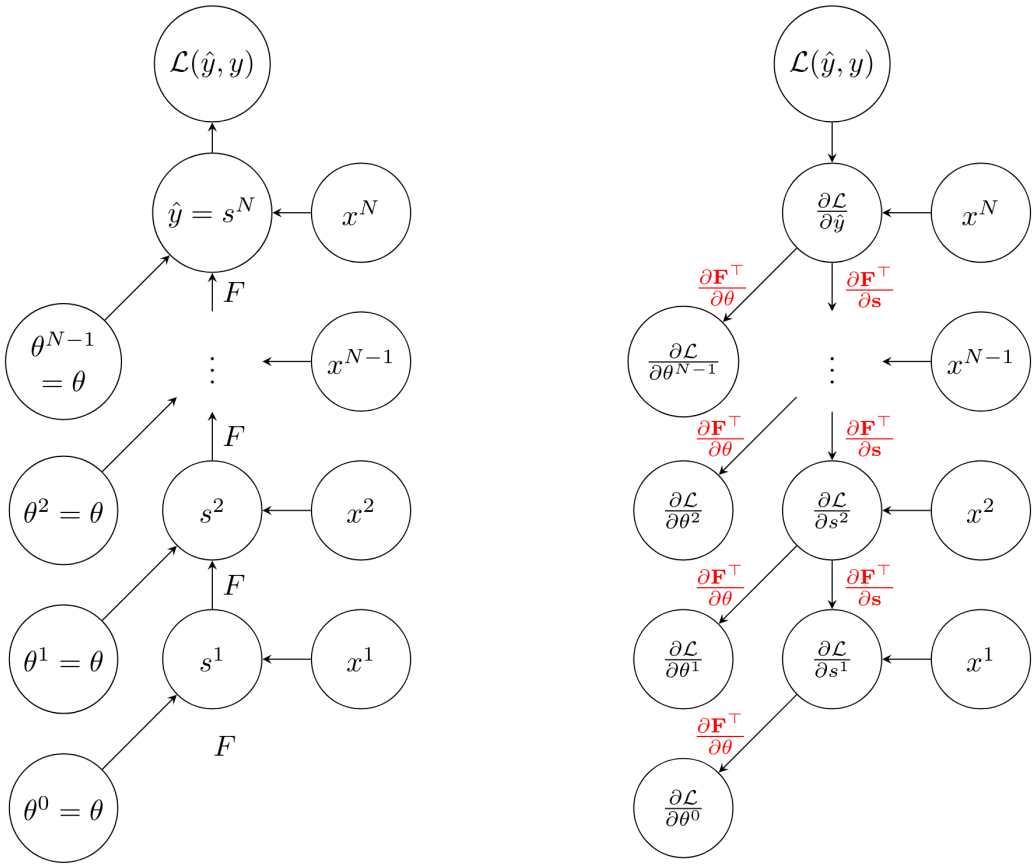

Figure 1.5: **Backpropagation through time in a recurrent architecture**. **Left**: forward pass. Differences with backpropagation in a feedforward architecture (Fig. 1.3): n labels time, all $F^{(n)}$ are the same with parameters θ^n shared across the whole graph and all equal to θ, and F takes x^n as an input at each computational step. **Right**: backward pass, going backward from $\frac{\partial \mathcal{L}}{\partial \hat{y}}$ at the top to leaves $\frac{\partial \mathcal{L}}{\partial \theta^n}$. Difference with backpropagation in a feedforward architecture (Fig. 1.3): because θ is shared across the whole forward computation, all derivatives $\frac{\partial \mathcal{L}}{\partial \theta^n}$ contribute to the same parameters θ and all add up to provide $\frac{\partial \mathcal{L}}{\partial \theta}$ (see Eq. (1.18))

General derivation. We now consider another neural network topology described by the following equations:

$$\left\{ \begin{array}{ll} s^1 & = F(x^1; \theta^0 = \theta) \\ s^n & = F(x^n, s^{n-1}; \theta^{n-1} = \theta) \qquad \forall n \in [2, N] \end{array} \right. . \tag{1.16}$$

Note the following important differences of Eqs. (1.16) with Eqs. (1.11):

- The data x is more specifically here a data stream $x^1, x^2, \cdots x^N$ where each x^n is fed at the n-th computational step.
- The label n is thereby better thought of as a temporal label.
- The transition functions F^n and associated parameters θ^n are all the same: $F^n(\cdot, \theta^n) = F(\cdot, \theta)$. An important consequence is that the parameters θ are shared across the computational graph, which comes into play when backpropagating derivatives through this type of graph.

All these features motivate to call this type of neural network a *recurrent neural network*. In general in a recurrent neural network, the goal of learning is to match a target y^n at every time step, typically for time series prediction or translation. So the model builds an estimate for y^n at each time step that we denote here $\hat{y}^n$. The loss for this kind of problem generally reads as:

$$\left\{ \begin{array}{l} \min\limits_{\theta} \mathcal{L}(\hat{y}, y; \theta) \\ \mathcal{L}(\hat{y}, y; \theta) = \sum\limits_{n} l(\hat{y}^n; y^n) \end{array} \right. , \tag{1.17}$$

where l is called a *cost* function. For simplicity here, we will restrict ourselves to consider the particular case where the loss only depends upon the last time step: $\mathcal{L}(\hat{y}, y; \theta) = l(\hat{y}^N; y^N)$, so that we subsequently drop the label N.

For the sake of gradient descent again, we want to compute $\frac{\partial \mathcal{L}}{\partial \theta}$ with $\mathcal{L} = l(\hat{y}(s^N), y)$, where the model estimate $\hat{y}$ is built from s^N at the last time step. A subtlety that we have mentioned before is that now, the parameter θ is shared across the whole computation: θ is used at each time step, as described per Eq. (1.16). Consequently, the effect of changing $\theta \to \theta + \delta\theta$ upon $\mathcal{L}$ depends on when this perturbation occurs. More specifically, when changing $\theta \to \theta + \delta\theta$ at time step n, the loss $\mathcal{L}$ evaluated at time step N is going to change by $\delta\theta^\top \cdot \frac{\partial \mathcal{L}}{\partial \theta^n}$, where $\frac{\partial \mathcal{L}}{\partial \theta^n}$ is a writing convention to express that the change is caused by a variation happening at time step n. With these notations, the "total" gradient $\frac{\partial \mathcal{L}}{\partial \theta}$ reads like:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial \theta^{N-1}} + \frac{\partial \mathcal{L}}{\partial \theta^{N-2}} + \cdots + \frac{\partial \mathcal{L}}{\partial \theta^0} \tag{1.18}$$

Finally, the computation the derivatives $\frac{\partial \mathcal{L}}{\partial \theta^{N-n}}$ exactly proceeds like Eq. (1.13), so that the whole computation of $\frac{\partial \mathcal{L}}{\partial \theta}$ is given by:

$$\boxed{\begin{cases} \frac{\partial \mathcal{L}}{\partial \theta} & = \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial \theta^{N-n}} \\ \frac{\partial \mathcal{L}}{\partial s^N} & = \frac{\partial l}{\partial s}(\hat{y}, y) \\ \frac{\partial \mathcal{L}}{\partial s^{N-n}} & = \left(\frac{\partial F}{\partial s}(x^{N-n+1}, s^{N-n}; \theta)\right)^\top \cdot \frac{\partial \mathcal{L}}{\partial s^{N-n+1}} \qquad \forall n > 1, \\ \frac{\partial \mathcal{L}}{\partial \theta^{N-n}} & = \left(\frac{\partial F}{\partial \theta}(x^{N-n+1}, s^{N-n}; \theta)\right)^\top \cdot \frac{\partial \mathcal{L}}{\partial s^{N-n+1}} \qquad \forall n > 1. \end{cases}} \tag{1.19}$$

Again, the computations of Eq. (1.19) can be conveniently depicted by the computational graph represented in Fig. 1.5.

Example. We consider the following simple recurrent neural network where each hidden state receives input from the previous one and the current data input, and an output is given based on this hidden state as:

$$\begin{cases} h_t & = \sigma([w^h, w^x] \cdot [h_{t-1}, x_t] + b^h) \quad \forall t \in [1, T] \\ o_t & = \sigma(w^0 \cdot h_t + b^o) \quad \forall t \in [1, T] \\ \hat{y} & = o_T, \end{cases} \tag{1.20}$$

where we use t as a label to emphasize to computation happens through time and $[u, v]$ stands for the concatenation of vectors u and v. Taking the loss to be $\mathcal{L} = l(o_T, y)$, applying Eq. (1.19) to Eq. (1.20) yields, for the parameters w^h:

$$\begin{cases} \frac{\partial \mathcal{L}}{\partial w^h} & = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial w^h_{T-t}} \\ \frac{\partial \mathcal{L}}{\partial h_T} & = w^{0\top} \cdot \left(\sigma'(w^o \cdot h_T + b^0) \odot \frac{\partial \mathcal{L}}{\partial o}\right) \\ \frac{\partial \mathcal{L}}{\partial h_{T-t}} & = w^{h\top} \cdot \left(\sigma'(w^h \cdot h_{T-t} + b^h) \odot \frac{\partial \mathcal{L}}{\partial h^{T-t+1}}\right) \quad \forall t \in [1, T] \\ \frac{\partial \mathcal{L}}{\partial w^h_{T-t}} & = \left(\sigma'(w^h \cdot h_{T-t} + b^h) \odot \frac{\partial \mathcal{L}}{\partial h^{T-t+1}}\right) \cdot h_{T-t}^\top \quad \forall t \in [1, T] \end{cases}, \tag{1.21}$$

for the parameters w^x:

$$\begin{cases} \frac{\partial \mathcal{L}}{\partial w^x} & = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial w^x_{T-t}} \\ \frac{\partial \mathcal{L}}{\partial h_T} & = w^{0\top} \cdot \left(\sigma'(w^o \cdot h_T + b^0) \odot \frac{\partial \mathcal{L}}{\partial o}\right) \\ \frac{\partial \mathcal{L}}{\partial h_{T-t}} & = w^{h\top} \cdot \left(\sigma'(w^h \cdot h_{T-t} + b^h) \odot \frac{\partial \mathcal{L}}{\partial h^{T-t+1}}\right) \quad \forall t \in [1, T] \\ \frac{\partial \mathcal{L}}{\partial w^x_{T-t}} & = \left(\sigma'(w^h \cdot h_{T-t} + b^h) \odot \frac{\partial \mathcal{L}}{\partial h^{T-t+1}}\right) \cdot x_{T-t}^\top \quad \forall t \in [1, T] \end{cases}, \tag{1.22}$$

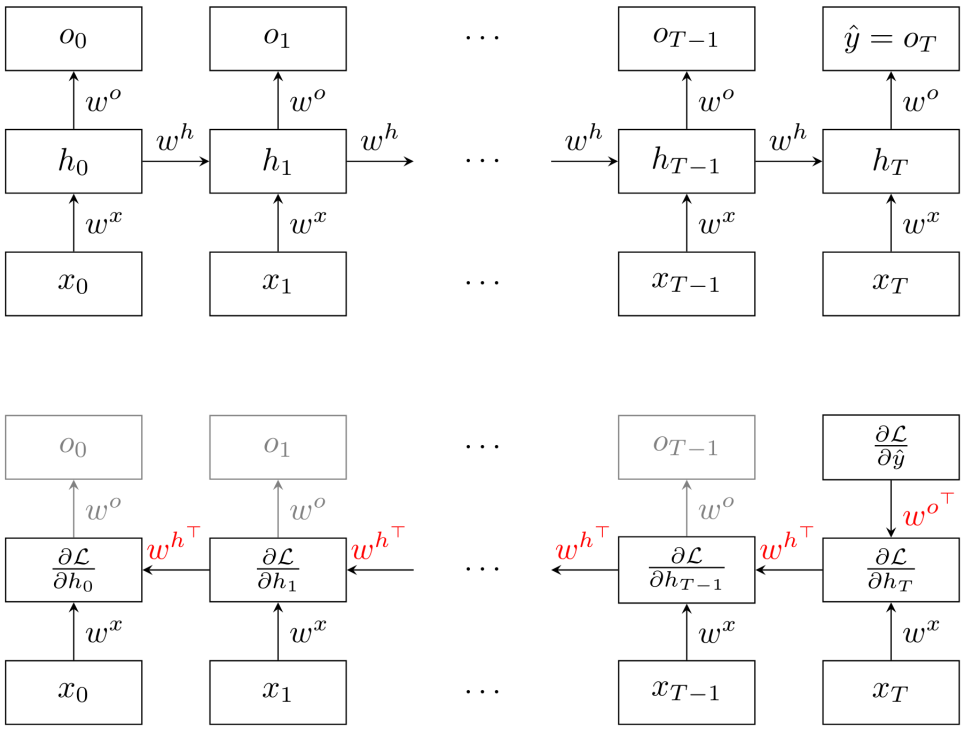

Figure 1.6: **Backpropagation in a recurrent neural network**. **Top**: forward pass, going from left to right. At time step t, each hidden layer value h_t is determined by the current input x_t and the previous hidden layer value h_{t-1}. Based on h_t, the neural network outputs o_t. **Bottom**: backward pass, going backward from $\frac{\partial \mathcal{L}}{\partial \hat{y}}$. Again, at each backward computational step, the error signals are routed by $w^{h^\top}$.

and finally for the parameters w^o:

$$\begin{cases} \frac{\partial \mathcal{L}}{\partial w^0} & = \frac{\partial \mathcal{L}}{\partial w_T^0} \\ \frac{\partial \mathcal{L}}{\partial w_T^0} & = \left(\sigma'(w^h \cdot h_{T-t} + b^h) \odot \frac{\partial \mathcal{L}}{\partial h^{T-t+1}}\right) \cdot h_T^\top \end{cases} . \tag{1.23}$$

Again, the computations of Eq. (1.21)-(1.23) are depicted in Fig. 1.6. Note that in Fig. 1.6, backpropagation goes explicitly backward in time, hence the name backpropagation *through time*.

1.3 The cost of learning on conventional computers

In the previous section, we have introduced the basic algorithmics of learning. How about the *hardware* that is used to run these algorithms in practice? In this section, we focus on this aspect and introduce two important limitations of today's widely used computer architectures.

1.3.1 The end of Moore's law

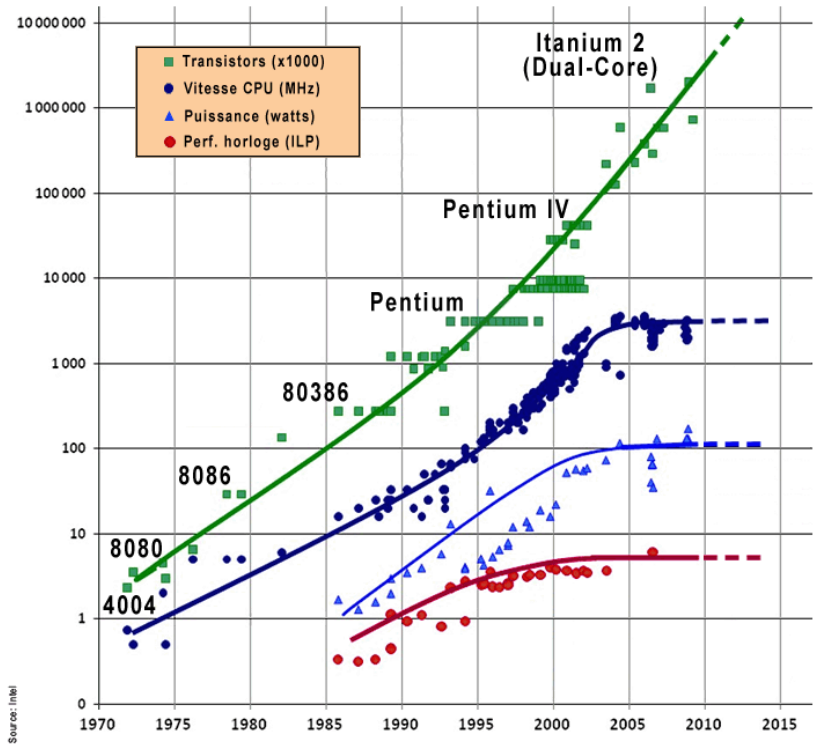

Figure 1.7: **The end of Moore's law**. Since 2005, as the number of transistors keeps increasing, physics limit clock frequency at around 4 GHz, thereby flattening speed and power curves from 2005 on.

Modern computing devices heavily rely on typical device and architecture paradigms, which happen to be fundamentally limited in certain ways today.

The Complementary Metal-oxide Field-Effect (CMOS) transistor is the building block of today's computer logic. A transistor is a three-terminal device (with a source, a gate and a drain) that can switch or amplify electrical signals. A current can only flow between the source and the gate depending on whether a voltage is applied to the gate, thereby producing a binary output value (current flows through the transistor, and is able to charge the output or not). Cascading multiple of these transistors enables to emulate incredibly complex logical functions, so that the more transistors can be fitted onto a chip, the more computationally efficient this chip can be. In 1974 Robert Dennard stated an hypothesis named after him as the *Dennard scaling* (also known as the MOSFET scaling) [17] that as transistors get smaller, their power density should stay constant, so that since the power use stays in proportion with area, both voltage and current scale would decrease with the gate length, and so would their prices. In 1975, Gordon Moore made the observation that the number of transistors on an integrated chip should double about every two years, an iconic "law" known as "Moore's law" which would hold for the next forty years. With now the transistor feature size dropping

below 10 nanometers, the end of this empirical law has become physically inevitable: the smaller the transistor, the higher the leakage and the most likely the bit flips due to thermal noise [18]. More explicitly, if we assume the two logical states of the transistor are separated by a voltage barrier ΔV (also called the logic threshold voltage) and that it operates at the maximal clock frequency, then it can be easily shown that the power dissipated during the transient phases when switching between logical states scales with the transistor feature size F as:

$$P \propto \frac{\Delta V^2}{F^2}. \tag{1.24}$$

Also, using Boltzmann statistics, the probability that the logical state of a transistor flips under thermal fluctuations, with k_b denoting the Boltzmann constant, is:

$$\mathbb{P}(\text{flip}) \propto \exp \frac{-C\Delta V^2}{2k_bT}. \tag{1.25}$$

Since C is an increasing function of F (with a linear or parabolic dependency, depending on the hypothesis made on the gate thickness), Eq. (1.25) prescribes to *increase* ΔV upon decreasing F to keep $\mathbb{P}(\text{flip})$ constant. Conversely, Eq. (1.24) constrains to *decrease* ΔV upon decreasing F to avoid increasing dissipation. Therefore, below a certain F, both constraints cannot be satisfied, so that Moore's law "dies": the speed of the processors can no longer be improved - see Fig. 1.7.

1.3.2 The von Neumann bottleneck

A fundamental architecture limitation also adds up to this conventional computing paradigm. Back in 1945, John von Neumann proposed a computer architecture, where logics and arithmetics would be performed in a Central Processing Unit (CPU) separately from memory read and write operations, both in space and time [19]. This architecture, known as the Princeton architecture or the *von Neumann* architecture, prevailed at this time over more complex proposals as the Harvard computer architecture by allowing independent CPU and memory designs with different device and energy requirements. Unfortunately, this physical separation between logic and memory also causes the overall computational inefficiency of this architecture. In a von Neumann architecture, CPU and memory communicate through a single bus that can therefore only access one unit at a time. Consequently, for tasks that typically require numerous memory access and simple logic, the data transfer rate (also called "throughput") is much lower than the rate at which the CPU can operate, so that the CPU constantly waits for data transfers, even more as CPUs have gotten faster. It was shown that an order magnitude more energy is needed to transfer and access the data in memory or in the CPU than the core logical operations operated on this data [20].

This phenomenon is famously known as the *von Neumann bottleneck*, a physical bottleneck which, as computer design has long built upon this architecture, even created inefficiencies at the programming language level, as John Backus puts it in its ACM Turing Prize lecture [21]: "*[T]he von Neumann bottleneck [...] is [...] a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it*".

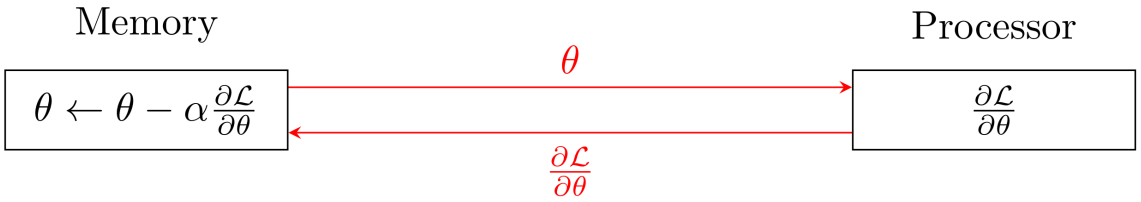

Figure 1.8: **The von Neumann bottleneck**. In von Neumann architectures, memory and computation are physically separated, which is particularly critical in the context of gradient descent for learning neural networks. At each learning step, parameters θ needs to be routed from memory to processors to compute gradients $\frac{\partial \mathcal{L}}{\partial \theta}$, which is subsequently routed back to memory to perform the parameter update $\theta \leftarrow \theta - \alpha \frac{\partial \mathcal{L}}{\partial \theta}$.

Both of these limitations — from the device and architectural prospective — are typically encountered in the context of learning as defined per section 1.2. Using the same notations, at a given learning iteration, θ is accessed from memory and processed in the computing unit to compute $\frac{\partial \mathcal{L}}{\partial \theta}$, which is then routed back to the memory to perform the parameter update $\theta \leftarrow \theta - \alpha \frac{\partial \mathcal{L}}{\partial \theta}$, so that the von Neumann bottleneck typically burdens gradient descent - see Fig. 1.8 for an illustration of this phenomenon. Also, as neural network models get bigger, computing efficiently $\frac{\partial \mathcal{L}}{\partial \theta}$ should not be taken for granted and is severely limited by the device constraints mentionned before. Although the use of Graphical Processing Units (GPUs, which were mentionned in section 1.1) can considerably accelerate gradient computation and the von Neumann bottleneck can be significantly mitigated for example with the use of cache memories or branch predictors, moving beyond these limitations requires a fundamental rethinking of the von Neumann architectures. In this purpose, *neuromorphic computing* is one approach to explore so-called "non-von Neumann architectures" for on-chip learning that take inspiration from the brain.

Chapter 2

An opportunity for neuromorphic engineering

Up to now, backpropagation has encountered a tremendous success, both in academia and industry, to train deep neural networks thanks to the use of GPUs with dedicated software frameworks like TensorFlow [22] or Pytorch [23], which employ automatic differentiation. However, designing even more powerful systems building on modern computing paradigms and conventional learning principles such as backpropagation has become fundamentally limited. In this chapter, we show how neuromorphic computing arises as a research opportunity in this context.

2.1 A brief history of neuromorphic engineering

Although there is no clear cut out definition today of "neuromorphic engineering", Carver Mead is considered as the main pioneer of the field, as he suggested there was something "fundamental to learn from the brain about a new and much more effective form of computation" [24]. Based on this inspiration, Mead fostered the development of Very Large Scale Integrated (VLSI) systems electronic circuits mimicking the brain and using MOSFET technology in the subthreshold regime. Operating in this regime requires smaller currents than in the standard digital regime and enables to emulate leaky integrate and fire (LIF) neurons, which depending on an internal analog value may spike or not. Using this CMOS-based LIF neuron as building block has helped achieve extremely energy-efficient systems that can handle simple pattern recognition with spike-based learning rules [25]. Although other significant CMOS-based analog neuromorphic circuits include the NeuroGrid chip developed at Stanford University [26], recent advances in industry and academia have mostly focused on the design digital circuits. SpiNNaker by the University of Manchester [27], TrueNorth by IBM Research [28] or Loihi by Intel [29] have been designed as fully CMOS, massively parallel neuromorphic architectures that separate spiking neurons and stored weights and operate

with an outstanding energy efficiency. While these chips are mostly of academic interest for prototyping spike-based learning and inference systems, the most ready-to-use optimized chip for deep learning applications comes from Google with their Tensor Processing Unit (TPU) ASIC chip [30].

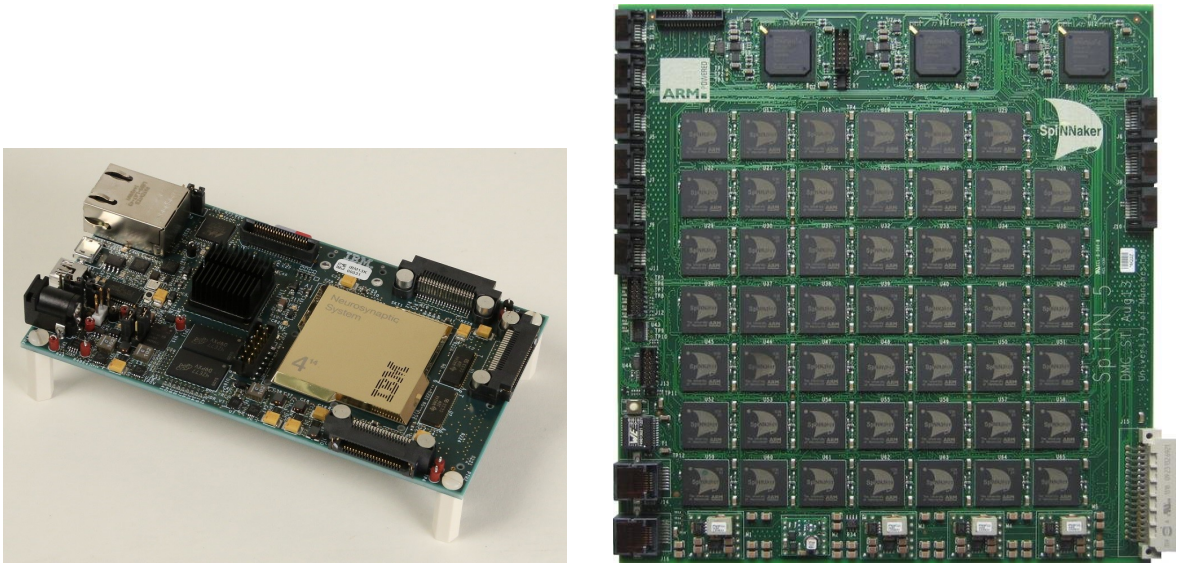

Figure 2.1: **Neuromorphic chips**. Left: IBM TrueNorth chip [28]. TrueNorth is made on 28-nanometer process technology. The processor contains 5.4 billion transistors and 4096 cores. Each core was provided with a task scheduler, SRAM-memory and a router. Right: SpiNNAker chip [27]. SpiNNAker is composed of 57,600 processors, each with 18 cores and 128 MB of mobile DDR SDRAM, totalling 1,036,800 cores and over 7 TB of RAM.

While all these systems are based upon well-established commercial technologies like CMOS, DRAM and SRAM, a new vision of neuromorphic computing has developed towards hybrid architectures incorporating emerging memory and logic nanodevices over the past decade. Among those, resistive memories — or *memristive* devices — are considered as compelling candidates for the design of massively parallel and energy efficient neuromorphic systems. Their nanoscale dimension, the low energy required to write and read their memory, their multiple bit-per-device capacity and the possibility to embed this technology into CMOS have incentivized proposals of neuromorphic building block circuitries [31]. A drawback of these technologies is that they are prone to imperfect programming due to the intrinsically unreliable physics at play. However, this aspect of memristive devices was perceived as an opportunity to model the biological features of synapses, and thereby build neuromorphic systems that would be fault-tolerant by design, using them as digital [32] or analog [33] memories. IBM Almaden recently suggested by hybrid software-hardware experiments that the use of non volatile memories as analogue synapses for training neural networks could deliver a computational power that would exceed those of most modern GPUs by two order of magnitudes [34]. The next section provides evidence about the huge potential of memristive

devices as artificial synapses candidates for on-chip learning.

2.2 The memristor as a promising building block for on-chip learning

Based on the fundamental relationships binding altogether current i, charge q, voltage v, magnetic flux linkage φ, Leon Chua theoretically investigated in 1971 the meaning and the implications of the relationship that could exist between q and φ, thereby suggesting, by symmetry arguments, the existence of a new two-terminal circuit element, albeit yet to be discovered, called a *memristor* [35] - as the contraction of "memory" and "resistor". More explicitly, the voltage drop of a charge-controlled memristor would be given at all time by:

$$\begin{cases} v(t) = M(q(t))i(t) \\ M = \dfrac{\partial \varphi}{\partial q} \\ v = \dfrac{\partial \varphi}{\partial t}, \end{cases} \tag{2.1}$$

so that the memristance M depends on the total integral of the past changes in charge, thence the memory effect. Chua showed that such a component could realize functions that none of the RLC-based circuits could on their own. In 2008, Hewlett-Packard (HP) laboratories claimed the discovery of the "missing" memristor in a very influential paper published in Nature [36], based on the definition of Chua and in the form of titanium dioxide switching cells. Many controversies arose after the paper was released: the memristor alledgedly unveiled by HP may not abide rigorously by Chua's definition and may rather belong to the broader class of resistance-switching devices, with solid arguments essentially based on thermodynamics [37, 38] and electromagnetism [39]. Whether a device is a "proper" memristor or not per Chua's 1971 definition therefore rather pertains to mathematics and was perceived as a rather unimportant matter for some device engineers [40]. In the scope of this thesis and for the sake of simplicity, we therefore make no distinction between a "memristor" and a "resitive memory".

A *resistive memory* is any two terminal physical device whose conductance can be modulated by current or voltage. There exists a wide variety of resistive memories, depending on the underlying physical mechanism causing the variation of conductance. Filamentary Redox-based resistive memories (ReRAM) are devices which can create conductive filaments between two electrodes through a transition metal oxide, depending on the voltage applied. Based on electrochemical reactions, the reduction of the anode create defect vacancies that are propagated to the cathode, and the conductance of the device grows as the filament gets thicker. Such a device typically uses metal-oxide films like HfOx or TiOx and combine them,

resulting in a simple, compact, CMOS-compatible technology, involving energies per synaptic operations that can be only sub pJ. Based on similar principles, the conductive-briding resistive memory (CB-RAM) exploits the electrochemical formation of conductive metallic filaments through an insulating solid electrolyte or oxide, also constituting a fast and low power technology. Ferroelectric resistive memories (FeRAM) are devices that use a ferroelectric layer instead of a dielectric layer to achieve non-volatility and which can achieve multiple conductance states. Phase-change memories (PCM) are based upon a different mechanism where the active part of the devices can either be in an amorphous (low conductance) or a crystalline (high conductance) state (or "phase", hence the name), switching between these two states through thermal activation by Joule effect. The material most widely used in PCMs is a germanium-antimony-tellurium alloy, which sits in an insulating middle layer and is also connected between an upper and lower electrode. Other resistive memories exploits magnetic effects, like the spin transfer torque (STT-RAM) memory in which a free magnetic layer can switch between two orientations with respect to a fixed layer, resulting in a very fast, current-controlled binary memory. A complete review on the use of non volatile memories for neuromorphic computing can be found in [33].

2.3 Bringing memory and computation the closest: crossbars

2.3.1 Kirchhoff laws for inference

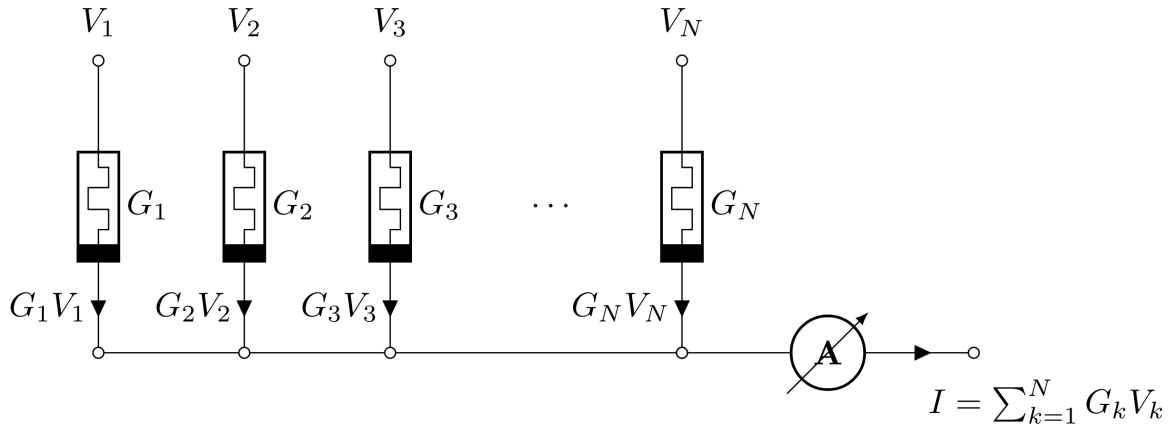

Figure 2.2: **Kichhoff laws for inference**. Each memristor carries a weight value as a conductance state G_n. With a voltage difference V_n, a current G_nV_n flows through each memristor by virtue of Ohm's law. Through Kirchhoff current law, all currents add up along the horizontal wire, so that the whole circuit implements the composition of a matrix multiplication.

Now that we have mentionned a few resistive memory technologies and put forward their advantages in terms of ultra low power consumption, let us account for their assets within

a neuromorphic system at a more abstract level. Let us assume a resistive memory whose conductance is denoted by G. When applying a voltage difference V to the device, Ohm's law states that the resulting current I flowing through the device is given by:

$$I = G \cdot V. \tag{2.2}$$

However innocuous it may seem, Eq. (2.2) is the corner-stone of resistive memory-based analog hardware neural network. Assume now N wires, each having a resistive memory G_i and undergoing a voltage drop V_i, all connected to the same output node. Then, applying the Kirchhoff current law along with Ohm's law yields the following expression measured at the output node:

$$I = \sum_{i=1}^{N} G_i \cdot V_i. \tag{2.3}$$

Algebraically speaking, the current flow through the N wires naturally performs the dot product $G^\top \cdot V$ - see Fig. 2.2 for an illustration.

The immediate generalization of this principle to perform matrix multiplication leads to the concept of *crossbar*. A cross-bar is an electrical circuit to build logical circuits based on memristors, then used in neuromorphic applications to build analog hardware neural networks. The most simple way to think of a cross-bar is an array of N horizontal wires and M vertical wires - Fig. 2.3 depicts a crossbar with $M = N = 3$. The horizontal wires are the input wires where currents are injected, the vertical wires are the output wires where the currents are collected. Each output horizontal wire i is connected to each of the input vertical wires j with a resistive memory of conductance G_{ij}. Therefore like Eq. (2.3), each output current I_i reads:

$$I_i = \sum_{j=1}^{N} G_{ij} \cdot V_j, \tag{2.4}$$

which goes to show that the crossbar defined as such performs the matrix multiplication $G \cdot V$. Adding extra circuitry (using an op-amp for instance) at the end of each output wire enables to implement a non-linearity σ, which changes Eq. (2.4) into:

$$I_i = \sigma\left(\sum_{j=1}^{N} G_{ij} \cdot V_j\right), \tag{2.5}$$

which brings us back to Rosenblatt's neuron model response Eq. (1.5). Eq. (2.5) gives us a flavor of the huge potential of memristive devices for neuromorphic applications: inference

is performed "for free" thanks to Kirchhoff's and Ohm's laws! These multiply-accumulate operations can be performed in parallel where the data is located in a locally analog fashion, thereby cutting power consumption by avoiding weight data transport [41].

2.3.2 A compelling case for memristor-based learning

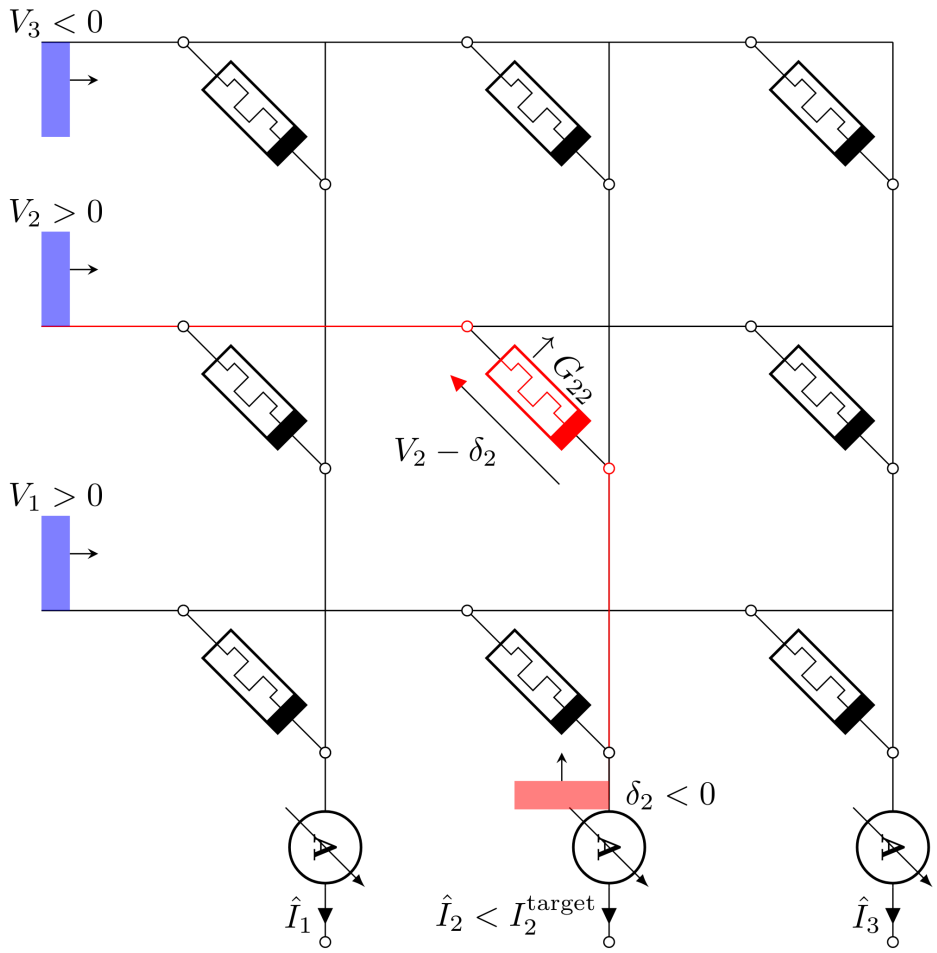

Figure 2.3: **Crossbar learning**. Inputs V_1, V_2 and V_3 are encoded as voltages (left end of horizontal wires) and outputs I_1, I_2 and I_3 as currents (bottom end of vertical wires). Each input j is connected to output i through a memristive device of conductance G_{ij}. On the cartoon, $\hat{I}_2 < I_2^{\text{target}}$, thus the conductance of devices connected to positive voltage entries should be increased, as it is the case for G_{22}. A negative error voltage pulse δ_2 is therefore sent along the second vertical wire so that the device of conductance G_{22} undergoes the voltage difference $V_{22} = V_2 - \delta_2 > V_\nu$, so that G_{22} is increased.

How about performing learning such a crossbar, that is finding the good conductance values G_{ij} to solve linearly separable classes? Let us push the analogy further with the perceptron model described earlier by considering a dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \cdots, (x^{(M)}, y^{(M)})\}$. In

a neuromorphic context, the input and the output values need to be physically encoded. So we assume here that the inputs V are encoded as input voltages, resulting in output currents $\hat{I}_i(V)$, which are either above or below a current threshold, thereby determining the class inferred by the perceptron. Learning consists in adjusting the conductance values of the resistive memories so that the output current $\hat{I}(V)$ matches the target current I^{target}: $\hat{I}_i(V) = I_i^{\text{target}} \quad \forall i$.

How does learning proceed physically? We assume each resistive memory can only be written if the voltage difference exceeds a voltage threshold V_ν: $\Delta G_{ij}(\Delta V_{ij}) = 0$ if $\Delta V_{ij} < V_\nu$ with ΔV_{ij} being the voltage difference applied to the device of conductance G_{ij}. Input voltages V_j are chosen below V_ν in absolute value, so that the resistive memories cannot be written without an additional voltage difference. With this in mind, let us assume voltage inputs V_j yielding a current $\hat{I}_i(V)$ that is compared against the ground-truth current I_i^{target}. If $\hat{I}_i(V) < I_i^{\text{target}}$ (the output current is too low), then any resistive memory connected to a positive voltage input V_j should have its conductance increased. In this case, an error voltage $\delta_i > 0$ is applied on the output wire of G_{ij}, where δ_i is taylored so that $\Delta V_{ij} = \delta_i - X_j > V_\nu$. Fig. (2.3) illustrates this learning procedure.The same logics applies in the 3 other situations, depending on whether the output current is too high or too low and the input voltage positive or negative, so that the learning rule reads heuristically *:

$$\Delta G_{ij} \approx \text{sign}((\hat{I}_i - I_i^{\text{target}}) \cdot X_j). \tag{2.6}$$

Note that, up to taking the sign, Eq. (2.6) is equivalent to the learning rule of the Rosenblatt's perceptron given by Eq. (1.6). We will come back on this in the next chapter.

While these working principles for memristor-based inference and learning on a crossbar are tractable in theory and in practice on a perceptron, implementing deeper architectures using the same principle by cascading crossbars remains a challenge. The next chapter describes in further details the fundamental difficulties inherent to memristor-based on chip learning.

*Not taking into account the memristor characteristic.

Chapter 3

Challenges of on-chip learning

The reasons why learning with resistive memories is extremely challenging are rooted into many essentially different aspects. A considerable range of difficulties to overcome stem from the technologies themselves, but also from the underlying algorithmics implemented. The fundamental trade-off that underlies this research could be phrased as: with regards to learning, how much can we give up on the energy spent on computational precision, memory, data routing and mathematical guarantees for the learning rule computation and the parameter update? We deal with each of these different aspects appearing in the literature in the next sections.

3.1 On and off-chip learning, analog and digital memories

One of the very first reason why resistive memory-based learning is hard comes from programming these devices: given an update prescribed by a parameter gradient value or any learning rule, how do we update the most accurately the conductance of a memristive device?

First, what a resistive device can encode as a memory in terms of bit capacity remains an open question. In the context of learning, resistive memories are often used or thought of as analog memories, when the conductance update can be gradual enough, thereby encoding a real value. Depending on the technology considered though, some resistive memories are better used as digital memories and therefore more suited for purely inferential engines [32], or can be combined with analog memories within mixed-precision architectures to achieve learning [42, 43].

Also, what is meant by "learning" should be clarified in the context of memristive technology-based learning. One approach is to compute the good value for the weights off-chip with software-based training simulations, then to map them onto a cross-bar as conductance values with elaborate voltage programming protocols [44, 45]. Another approach, which we want to described further in this section, is *on-chip* learning: the whole learning process is achieved on the chip, iterating inference (or "forward pass"), gradient computation and parameter update. In the next section, we focus on this approach.

3.2 Device programming

In spite of their compelling potential for neuromorphic applications, memristive devices exhibit undesirable features when it comes to updating their conductance in the purpose on-chip learning.

The first two most obvious difficulties directly arise from their nanoscale size: since the physical mechanisms governing the conductance updates operate at the atomic level as described in section 2.2, the conductance updates of resistive memories are inherently stochastic and vary a lot from a device to another one - see Fig. 3.1 where the device characteristic exhibits cycle-to-cycle variability. These two sources of stochasticity are often referred to as "cycle-to-cycle variability" and "device-to-device variability" respectively. For instance, ReRAM devices are particularly subject to this stochasticity since their conductance relies upon filament of atoms so that their formation or destruction obeys to a wide variety of physical parameters, entailing at the system level a significant spread of the minimal and maximal conductance values of the devices [46].

Another limitation of memristive devices is the asymmetry existing between potentiation (increase of conductance) and depression (decrease of conductance). PCM devices, described in subsection 2.2, are subject to such an asymmetry since their amorphization is abrupt while their crystallization can be gradual, so that only the conductance increase is well controlled [47]. One solution to this issue is to implement each synapse W with a pair of memristors of conductances G_+, G_- so that $W = G_+ - G_-$: G_+ (resp. G_-) should be potentiated to increase (resp. decrease) W. With this programming scheme however, devices are only programmed in potentiation so that they inevitably reach saturation, thereby requiring an occasional reset in conductance [47]. Similarly, hafnium-oxide RRAM exhibit a gradual conductance decrease (i.e. the dimensions of an existing conductive filament can be gradually reduced) and an abrupt conductance increase (i.e. a conductive filament connects for the first time the two electrodes of the devices), which also requires the use of two complementary devices per synapse with a programming scheme adapted accordingly [43].

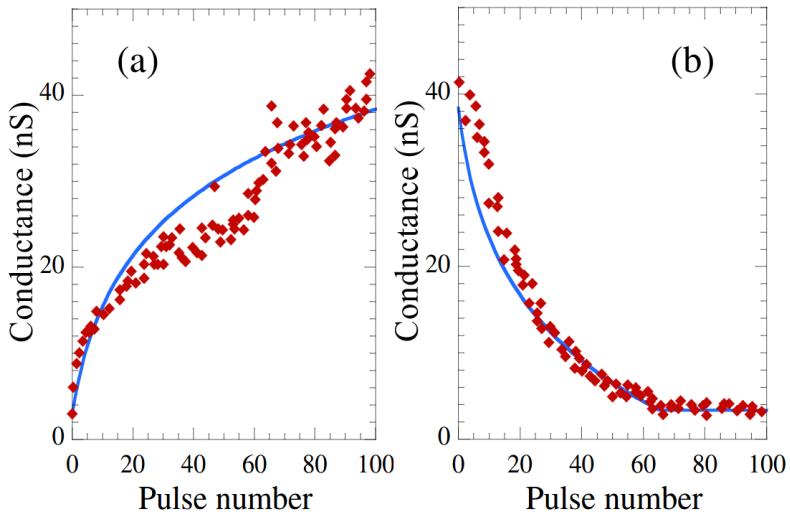

Figure 3.1: **Experimental memristor characteristic** (taken from [48]). The characteristic of filamentary Ag-Si RRAM is shown as the conductance state as a function of the number of programming voltage pulses applied, of 300μs each and $\sim \pm 3$V each, for potentiation (a) and depression (b). Red points are experimental and the blue curve is a fit. The memristor characteristic typically exhibits non-linearity, asymmetry between potentiation and depression and cycle-to-cycle variability.

The programming of memristive devices is also considerably limited by their finite conductance range $G_{\text{max}} - G_{\text{min}}$ [47]: the larger the conductance range, the more precise the programmation, the better the resulting performance at test time. One way to overcome this limitation is the use of a mixed architecture where imprecise analog conductance updates are cumulated in a digital unit with high precision [42]. Another approach consists in implementing each synapse with more than two memristive devices, yielding a larger conductance range and also a better training performance [49].

Most importantly, memristor-based learning is dramatically jeopardized by the non-linearity of the devices: the conductance update undergone by a device using constant programming conditions depends on its current conductance value - see Fig. 3.1 where the device characteristic exhibits non-linearity. For PCM devices as well as filamentary RRAM for instance, the conductance update for low conductance states is relatively high, and becomes smaller as the conductance value increases. Instead, linear devices whose conductance update does not depend on the current conductance value, would yield the best result on classification tasks [41, 50]. One first natural approach to overcome this limitation is to employ read-and-write programming schemes, where the conductance value of the device to program is first read, and the number of pulses applied are chosen accordingly [51, 52]. This solution however appears to be costly, with subsequent overhead on the peripheral circuitry around the crossbar [34, 53]. Another approach to mitigating non-linearity has been proposed by IBM Research in their milestone Nature paper, where each synapse is implemented with a pair of

PCM devices and a three transistors-one capacitor (3T-1C) capacitor [34]. The capacitor, whose charge-voltage characteristic is linear, accumulates the weight updates across training samples. Since the capacitor has a volatile memory, its charge is precisely transferred them onto the PCM devices as a conductance state every 8000 training samples, combined along with programming strategies to mitigate the variability of the capacitors. Also, recent work has demonstrated that the combined use of analog memories with digital operations is inherently more robust than more conventional fully analog approaches, without CMOS overhead [43].

3.3 Hardware-friendly learning rules

All the aforementioned approaches focus on the device engineering required to perform a conductance update in the most accurate way *given a target update*. Thereby implicitly, the computation of the gradient value of the loss function of interest - or of the learning used at all regardless of the notion of a loss function - is somewhat taken for granted. So far in the neuromophic computing literature, which learning rule to implement on a chip has taken two different paths.

3.3.1 Backpropagation?

One of the main approaches in neuromorphic computing to in-situ learning rule computation is to implement verbatim backpropagation. The most simple application of backpropagation is in a perceptron, where the learning rule is given by Eq. (1.6). In the on-chip learning protocol adapted for crossbars described in section 2.3, the learning rule Eq. (2.6) reads like the sign of Eq. (1.6). This learning rule presents significant advantages from a hardware prospective: it is local in space (the weight update solely requires the pre and post-synaptic neuron activities) and it takes the sign of the real gradient, which generally makes stochastic gradient descent in machine learning optimization more robust to noise [54, 55]. The learning rule given by Eq. (1.6) on a perceptron was coined the Delta Rule [56], or more generally on any optimization problem signSGD [54] and was successfully applied experimentally for the in-situ learning of 3x3 image patterns by a hardware perceptron [45].

So perceptron learning is one of the rare cases where backpropagation* readily applies, up to taking the sign of the gradient, in a hardware-friendly fashion. How we go about implementing backpropagation taking hardware constraints into account in deeper architectures remains one of the most challenging questions of on-chip learning. Most of IBM Research findings are carried out with hybrid hardware-software experiments where inference and gradient computation carried out ex-situ, with the resulting gradient value subsequently mapped

*Note that technically speaking, this is the Delta rule. Backpropagation refers to propagating error gradients across at least one hidden layer.

as a conductance update on the crossbar, with a strong focus on device engineering and slight algorithmic adaptation to hardware constraints [33, 34, 41, 47]. In particular, IBM Research has demonstrated equivalent accuracy to software-based techniques on CIFAR-10 with a convolutional architecture in the particular case of transfer learning where only the classifier is trained [34]. Along the same lines, Bennett et al advocate the use of extreme learning machine or NoProp [57], relaxing memory and energy budget, where only the classifier is learnt while the other weights are kept random and fixed. Random projections, which are of interest in many signal processing and machine learning applications, can also be implemented through light scattering [58] *. Similarly, Hirtzlin et al propose a robust in-situ learning scheme for Binarized Neural Networks mixing digital and analog operations, assuming implicitly the computation of the gradients applied to their RRAM devices [43]. Although the robustness of gradient descent to gradient binarization [54] or ternarization [59] can be seen as an opportunity in the scope of neuromorphic implementations, they still require high precision in the gradient computation.

Notwithstanding its gigantic popularity, it is widely acknowledged that standard backprop is biologically implausible, which is why it is believed not to account for learning in the brain [60]. Since most of neuromorphic computing research is driven towards biological inspiration to reproduce the outstanding robustness and energy-efficiency of the brain, backpropagation is also not hardware-friendly. The reasons to claim the biological unplausibility of backpropagation appear clearly in Eq. (1.15) and Fig. 1.4:

- the error signal $\frac{\partial l}{\partial s}(\hat{y}, y)$ is routed to upstream layers through the *transpose* of the forward weights ${w^{N-n}}^{\top}$: this is known as the *weight transport* problem.
- The gradients computed by backpropagation depend on the derivative of the activation function.
- The gradients computed by backpropagation also depend on the activations of the neurons during the forward phase, which therefore need to be stored during the (backward) gradient computation phase.
- Backpropagation computation is not carried in the circuitry of inference: inference and gradient computation are not "handled by the same system".
- Finally, the gradient $\frac{\partial \mathcal{L}}{\partial w^{N-n}}$ is *not local in space*: the resulting weight update to apply to w^{N-n} does not solely depend on the adjacent neurons s^{N-n} and s^{N-n+1} but also on the output layer $\hat{y}$.

Most significantly, the non-locality of backpropagation is a serious issue for on-chip learning. If we were to design a physical deep neural network on a chip out of cascaded crossbars with resistive memories, the conductance update of each device at each learning iteration

*This principle led to the development of an *Optical Processing Unit*, a hardware co-processor.

would require information that lies in the output layer of the neural network, possibly very far apart spatially, thereby creating another bottleneck! Therefore, neuromorphic researchers and neuroscientists endeavors converge to strive for efficient *local* learning rules, the former for hardware energy-efficiency and the latter to explain human intelligence.

3.3.2 Spike Timing Dependent Plasticity (STDP)

Another widely used learning rule in neuromorphic implementations of spiking neural networks is the *Spike Timing Dependent Plasticity* (STDP), which can be considered as a spiking version of Hebbian learning as defined in Eq. (1.3) [61]. The STDP rule prescribes to modulate the synapse update depending on the firing times of the pre-synaptic and post-synaptic neuron. If a pre-synaptic spike precedes a post-synaptic spike within a specific time-frame, these two spikes may somewhat be causally related so that the STDP rule increases the value of the synaptic weight. In the other way around, a pre-synaptic spike may not be causally related to a previous post-synaptic spike, and the synaptic weight value should therefore be decreased in such case. This STDP mechanism as described, however, is only an interpretation and its underlying causality remains questioned.

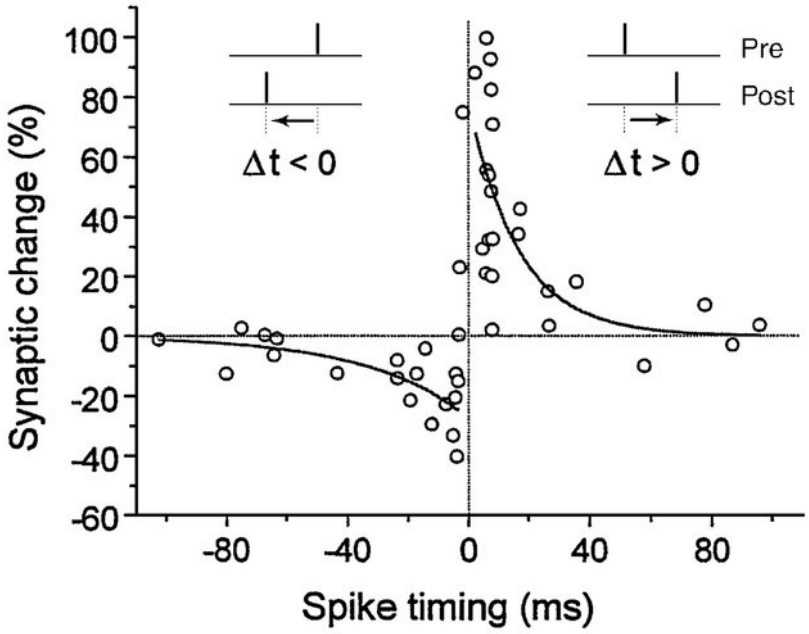

Figure 3.2: **Spike Timing Dependent Plasticity** [61]. In vitro experiments have shown the the rate of synaptic change is correlated with the relative timings o pre and post synaptic spikes. On average, whenever a pre synaptic spike precedes (resp. follows) a post-synaptic spike, the synapse gets potentiated (resp. depressed).

Very importantly, STDP has been particularly attractive to neuromorphic researchers for its locality in space and time: all the information a synapse needs to update its value at any time is carried by the current value of its adjacent neurons! The STDP rule can be realized on

a memristor [62]: if the pre and post-synaptic spikes have appropriate shapes, since the device is programmed by the voltage difference created by these spikes, the resulting conductance update can be correlated with the relative firing times of the pre and post-synaptic neurons - see Fig. 3.3 for an illustration. This memristor-based STDP has proven efficient in simulations for unsupervised learning of temporal correlations for car detection [63] or hand-written digit recognition [64] with 92% test accuracy*. Strikingly, simulations have shown that spiking memristor-based neural networks employing STDP exhibit an outstanding tolerance to device imperfections [48] – even more: they show in some situations that the non-linearity of the devices can be an asset! However, these demonstrations were carried out on a two-layers perceptron architecture and poorly scale to deeper architectures.

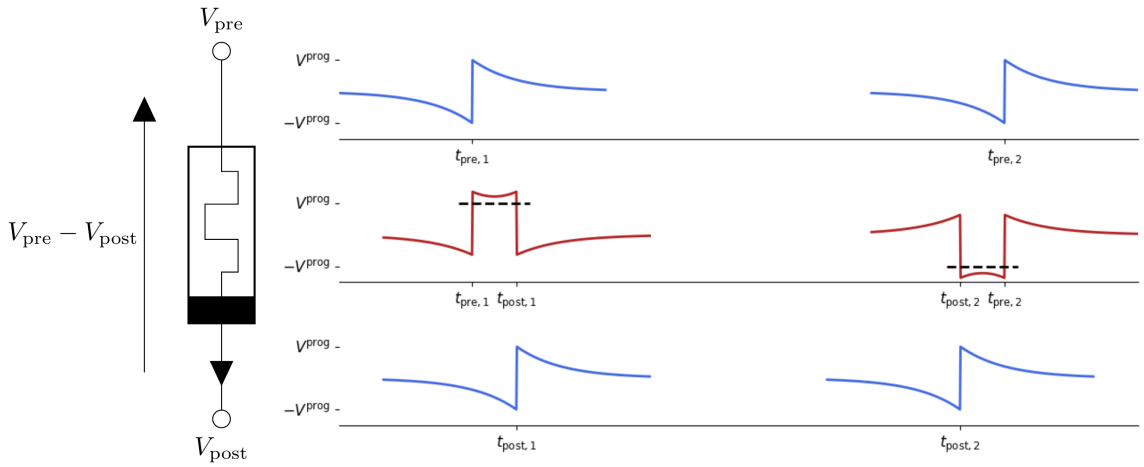

Figure 3.3: **Memristor-based Spike Timing Dependent Plasticity**. Pre and post synatic neurons fire at t_{pre} and t_{post} times respectively with specific pulse shapes. At all time, the memristive device undergoes the voltage difference created by the pre and post synaptic pulses. Upper panel, low panel and middle panel show pre-synaptic spikes, post synaptic spikes and voltage difference through time respectively. The pulse shapes are chosen so that whenever a pre (resp post) synaptic spike precedes a post (resp pre) synaptic spike within a time-frame, the voltage difference exceeds a positive (resp negative) threshold so that the device is subsequently potentiated (resp depressed), thereby emulating STDP. This principle is employed in [48].

Recent work proposes a reward-modulated version of STDP [65] where the sign of the STDP curve is modulated by the error signal produced at the output layer. When applying this version of STDP on the top layers of a spiking convolutional neural network and standard STDP elsewhere with latency coding, the authors report a test accuracy of 97.2% on MNIST, an improvement compared to then STDP state-of-the art, yet far from rate-based performance achieved with backpropagation. Part of the reason why STDP-based approaches may be limited is that the STDP rule itself is generally not derived as the gradient of an objective function in the context of classification but rather taken as a biologically plausible learning

*Which is, however, not as good as logistic regression.

heuristic. Only in a few works, some form of STDP-like learning rules were shown to arise from maximum likelihood approaches to reproduce arbitrary spiking patterns [66], but were only limited to small systems and simple pattern recognition.

Chapter 4

Towards better credit assignment for on-chip learning

4.1 What is credit assignment?

What we mean here by "credit assignment" is explained the best by Richards et al. in their recent proposal of a deep learning framework for neuroscience [60]:

> ***The concept of credit assignment refers to the problem of determining how much "credit" or "blame" a given neuron or synapse should get for a given outcome****. More specifically, it is a way of determining how each parameter in the system (for example, each synaptic weight) should change to ensure that [the objective function is optimized]. In its simplest form, the credit assignment problem refers to the difficulty of assigning credit in complex networks. Updating weights using the gradient of the objective function [...] has proven to be an excellent means of solving the credit assignment problem in ANNs. A question that systems neuroscience faces is whether the brain also approximates something like gradient-based methods.*

Credit assignment is of particular relevance in neuromorphic computing: once inputs and outputs of a given neuromorphic system are encoded, how should we change the physical observables of this system so that its output is improved with respect to a task, or namely how do we assign "credit" to these physical observables? In other words, how can the error signal be

physically encoded and even computed by the physics of the system itself? Backpropagation and STDP mentionned before are only two particular cases of how credit should be assigned to conductances of resistive devices, and this thinking extends well beyond to more complicated physical systems reported in the neuromorphic litterature. For instance, it was shown that the coupling physics of magnetic oscillators could be leveraged for the classification of vowels [67], with a learning scheme based on complex physics that it is worth describing to convey how critical credit assignment can be in neuromorphic systems. Input vowels are encoded as frequencies emitted by input magnetic oscillators that can couple in frequency with other magnetic oscillators. The classification output is decoded as synchronization patterns between oscillators, for instance two specific oscillators A and B should systematically be synchronized the vowel "a" is presented. If upon presenting the vowel "a" during training A and B are not synchronized, credit is assigned to the current injected in A and B so that their natural frequencies get closer, until A and B are synchronized.

However, the credit assignment schemes used in the neuromorphic litterature have yet to scale to bigger systems. The belief exposed in this thesis is that in order to generalize a credit assignment mechanism to bigger neuromorphic systems, it should preserve some theoretical guarantee that the objective function of interest is optimized. In the following sections, we present some alternatives to backpropagation that can approximate gradients of an objective function that exist in the machine learning and computational neuroscience literature, but have yet to be explored in neuromorphic computing, in order to motivate the objectives of this thesis.

4.2 Hopfield Networks & Contrastive Hebbian Learning

4.2.1 A brief history

Historically, since the McCulloch-Pitts neuron model was proposed in 1943, the research in neural networks split into two distinct, though permeable, quests: the development of the deep learning approach to AI we described earlier, and computational neuroscience to better understand the brain.

In 1982, at about the same time when backpropagation was developped, John Hopfield came up with a model of the human memory called content-addressable memories", or "associative memories", which are today remembered as Hopfield Networks [68]. These networks learn to remember input patterns so that after learning, if part of the input is removed, it can be recovered by sampling the model. Hopfield networks were first trained with Hebbian learning rules of the kind of Eq. (1.3)-(1.4), which feature *spatial locality*: the weight update on w_{ij} solely depends on the neurons x_i and x_j, contrary to backpropagation. Also, within the scope of this thesis, it is important to point out at this stage that the Hopfield network is

one of the first *energy-based* neural network model: each configuration of the neural network is associated to a scalar value depending on the values of the weights called an "energy". Learning consists in adjusting the weights, thereby deforming the energy landscape seen by the neurons, so that the energy minima correspond to the input patterns. When "sampling" the model from a corrupted input, the neurons descend the energy landscape until reaching a minima and should therefore recover the input patterns that have been learnt. In Hopfield networks, neurons are binary and evolve according to deterministic equations which descend the model energy function. Later in 1985, David Ackley and Geoffrey Hinton proposed the *Boltzmann Machine* model [69] taking its name and inspiration from the notorious theoretical physicist who pioneered statistical mechanics. In this model, input patterns should also be learnt as minima of an energy landscape, neurons assume binary values but are sampled *stochastically*: minima of the energy function correspond to modes of the model distribution. The introduction of stochasticity into energy-based models not only made this approach closer to biology but also helps neurons escape local minima, also called "spurious patterns", of the energy landscape.

Boltzmann machines, whose posterior distribution is intractable in general and therefore makes inference difficult, lead to the development of *Restricted Boltzmann Machines* (RBM), first known as "harmoniums" [70]. RBMs are two-layered Boltzmann Machines without internal connections within each layer, so with "restricted" connections. Owing to their simplified topology, inference is tractable in Restricted Boltzmann Machines and their learning can be achieved with *Contrastive Divergence* [71] which also prescribe a spatially local learning rule*. Later, *Deep Boltzmann Machines* extended Boltzmann Machine learning to stacks of Restricted Boltzmann Machines, employing variational inference to approximate the posterior distribution of the network †.

Without delving into technicalities, it is useful at this stage to convey intuition about energy-based learning to prelude this thesis. Contrastive Divergence, which is employed to train all variants of Boltzmann Machines previously mentionned, can be seen as a stochastic version of *Contrastive Hebbian Learning*, which was first introduced by Ackley and Hinton [69] and later formulated in a purely deterministic setting by Javier Movellan [72]. Let us assume a neural network (e.g. a Boltzmann Machine) with a global state variable s whose energy reads:

$$E = -\frac{1}{2} s^\top \cdot ws, \tag{4.1}$$

where w are the weights of the synapses connecting the neurons s. We also assume that the neurons evolve according to certain dynamics $t \to s_t$ towards decreasing values of the

*Part II will further detail Restricted Boltzmann Machine and Contrastive Divergence.

†Part III provides further details on Deep Boltzmann Machines and variational inference.

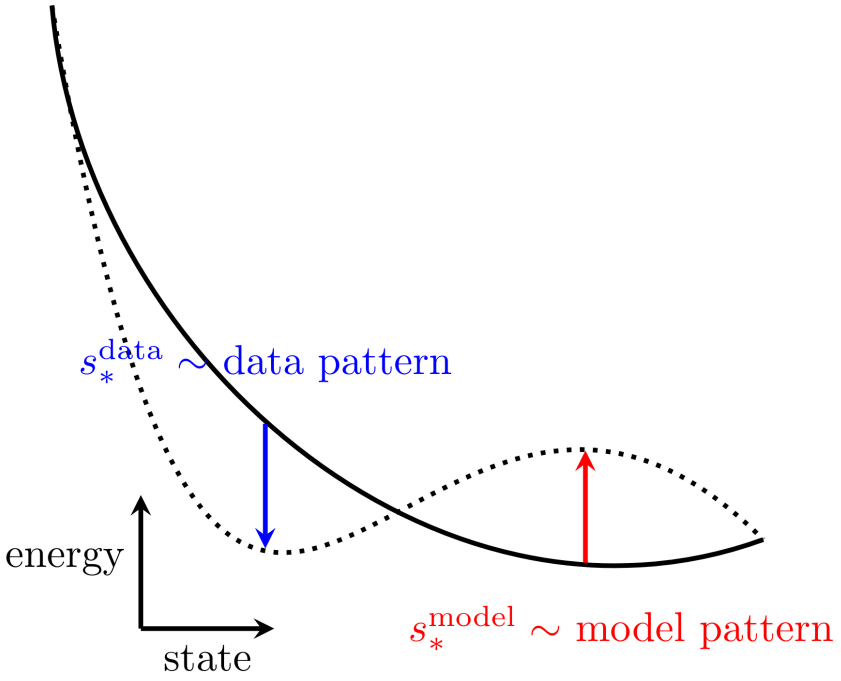

Figure 4.1: **Contrastive Hebbian Learning**. This figure illustrates Eq. (4.2). The plain and dotted lines represent the energy landscape seen by the neurons before and after learning respectively. By definition, neural dynamics evolve towards minima of the energy landscape, denoted here as s_*^{model} and called a 'model pattern'. At the beginning of learning, s_*^{model} does not correspond to configurations that account for the data that we call here "data patterns" (s_*^{data}): for instance for discriminative models the resulting output layer does not give the ground-truth target or for generative models the resulting visible layer does not correspond to the input data. The learning rule prescribed Eq. (4.2) amounts to reshape this energy landscape by increasing the energy of s_*^{model} ($-s_i^{*,\text{model}} s_j^{*,\text{model}}$ in Eq. (4.2)) and decreasing the energy of s_*^{data} ($+s_i^{*,\text{data}} s_j^{*,\text{data}}$ in Eq. (4.2)) so that the system subsequently evolves towards data patterns.

energy E that we denote s^*, typically the dynamics read $ds/dt = -\partial E/\partial s$. Using the same notations as before, we also define $\hat{y}$ as a subset of the neurons s that encodes the output of the neural network, e.g. $\hat{y}$, can be the visible layer of the neural network if we want to train a generative model, or conversely it can be the output layer of the neural network if we want to train a discriminative model. Again, let y be the target value for $\hat{y}$. The output neurons of the neural network $\hat{y}$ may evolve freely or not, depending on whether the target value is clamped on the output neurons, so that we enforce $\hat{y} = y$. The energy minimum that is reached depends on these two situations, so that we distinguish between $s^{*,\text{free}}$ and $s^{*,\text{clamped}}$. Heuristically, $s^{*,\text{clamped}}$ states therefore 'better' account for the ground-truth output y than $s^{*,\text{free}}$ states: $s^{*,\text{clamped}}$ states should therefore be minima for the energy. With this intuition in mind, the learning rule prescribed by Contrastive Hebbian Learning reads for the synapse w_{ij}:

$$\Delta w_{ij} \propto s_i^{*,\text{clamped}} s_j^{*,\text{clamped}} - s_i^{*,\text{free}} s_j^{*,\text{free}}. \tag{4.2}$$

Note from Eq. (4.1) that Eq. (4.2) amounts to increase the energy of $s^{*,\text{free}}$ and to decrease the energy of $s^{*,\text{clamped}}$. Therefore, upon iterating Eq. (4.2), the energy landscape minima correspond to the network states where $\hat{y} \sim y$.

4.2.2 Hardware implementations

One intermediate path towards reducing the gap between STDP-based and gradient-based approaches is to engineer STDP learning rules properly to that they descend the gradient of an objective function. In their event-based version of Contrastive Divergence, Neftci et al propose a version of STDP that is able to train a spiking Restricted Boltzmann Machines for discrimination and generation on MNIST, which is shown to approximate Contrastive Divergence that itself is an approximation of the gradient of the log-likelihood [73], achieving 91.9% test accuracy on MNIST, thereby close to the rate-based performance by less than 2%. This algorithm recently led to the first * experimental realization of a fully hardware spiking Restricted Boltzmann Machines [75] where synapses are made up of PCM devices and which can achieve 92% training accuracy over 100 MNIST samples.

The philosophy of the research presented in this thesis goes along the same direction: finding or building upon hardware friendly learning rules that preserve some theoretical guarantees.

4.3 Biologically plausible credit assignment

Along with Contrastive Hebbian Learning, there have been many attempts in the neuroscience and deep learning fields to account for how the brain might perform credit assignment in a way that is as efficient as backpropagation on standard learning benchmarks.

4.3.1 Reinforcement-based credit assignment

Many credit assignment mechanisms are based on reinforcement learning techniques. In this learning paradigm, an agent evolves in an environment which may reward or punish the agent depending on its state, as neuromodulators like dopamine do it in the brain. Therefore, learning consists for the agent in determining an optimal policy giving the best action to take

*Prior work proposed a hybrid hardware-software implementation of a Restricted Boltzmann Machine where neuron dynamics were emulated off-chip [74].

given its current state to maximize its cumulated reward over a trajectory in the environment. In most cases, the agent determines an optimal policy by building a map of reward estimates, also called Q-values. The agent should then find a balance between exploiting states of highest estimated reward and random exploration of states, which is also known as the exploitation-exploration trade-off.

Attention-Gated Reinforcement Learning (AGREL [76] or Q-AGREL in its most general version [77]) is a version of Reinforcement Learning applied to neural networks where during the forward phase, the prediction is seen as an action taken based on the output activations. Their values are interpreted as the model estimate of the rewards, or "Q-values", where the highest reward corresponds to the target output. During the backward phase, the network does not receive an explicit teacher signal depending on the target output, but gets rewarded or not depending on the predicted output. The synapse update is subsequently gated by a global reinforcement signal and the subset of neurons that were responsible for the prediction through an "attention" mechanism with specific feedback weights. Conversely, all the neurons are involved in backpropagation with the transpose feedforward weights. On average, the resulting weight updates approximates those provided by backpropagation.

Other techniques known as Node Perturbation and Weight Perturbation [78] consist in computing loss gradients by perturbing neurons ("nodes") or weights with noise and measuring the subsequent change in the loss value, also employing reinforcement learning techniques. However, in real neural circuits, it may not be possible to distinguish the injected noise from the intrinsic noise of the circuit, and therefore tell what caused the perturbation of the loss. Regression Discontinuity Design (RDD) overcomes this issue by inferring such causality using thresholding effects [79].

4.3.2 Credit assignment with generative models

Some attempts to propose biologically plausible learning models rely on the use of stochastic generative models.

In Difference Target Propagation (DTP), the error signal within each hidden layer is not a gradient but a target value [80]. While the target value for the output layer is simply the ground-truth target of the training sample, target values for hidden layers are computed by propagating upper target values with approximate inverses of the forward functions. These layer-wise inverse functions are approximated as auto-encoders where each of them learns to reconstruct a hidden layer from the upper hidden layer, without using the transpose of the forward weights to go backward.

Another approach is provided by the predictive coding framework [81] where the whole model is probabilistic with each layer being Gaussian conditionally on the previous layer. Maximizing the log-likelihood of the model given an input stimulus with respect to the neural states and the synapses yield local dynamics for the neurons and the synapses, where it is shown that in some limit, the synapse update is approximately the same as the one given by backpropagation. However, the error signal is routed by the transpose of the forward weights.

4.3.3 Credit assignment without weight transport

As emphasized in subsection 3.3.1, one biologically unplausible feature of backpropagation is the use of the transpose of the forward weights to route error signals back into the network. In the previously described approaches, Attention-Gated Reinforcement Learning (AGREL) and Target Propagation use distinct backward weights. Node Perturbation in its original version does not use specific backward weights, until recently where it is proposed to learn backward weights so that the error signals routed by these weights match the best the gradients computed by Node Perturbation [82].

An impactful paper showed that even when random weights are used as feedback weights in place of the transpose of the forward weights, learning occurs with a resulting performance close to backpropagation on benchmark visual tasks [83]. They also suggest that the underlying learning mechanism at stake relies upon the alignment of these backward weights with forward weights, a phenomenon known as Feedback Alignment, which later gave rise to Direct Feedback Alignment which uses feedback skip connections [84]. Bartunov et al showed that Feedback Alignment did not scale to complex visual tasks [85]. Xiao et al report a good peformance on ImageNet with a similar algorithm called Sign-Symmetry [86] where only the sign of the forward and backward weights should coincide, an hypothesis that they biologically justify. Recent work has shown that with two different mechanisms that improve the agreement between forward and backward weights, Feedback Alignment can be scaled up to hard visual tasks even better than Sign-Symmetry [87].

4.3.4 Assigning credit to apical dendritic compartments

In most conventional graphical presentations, neurons are represented as blobs in directed computational graphs so that, from a biologicaly prospective, they are mostly assimilated to their somas. So it seems, at first sight, that the neural dynamics are uniquely defined by feedforward equations of the kind of Eq. (1.14), while backpropagation is more of an artificial computation backward through the feedforward network. For biological soundness though, it is tempting to symmetricize the forward and the backward passes, so that the backpropagation of errors is itself part of the neural dynamics. In this case, using the same notations as section 1.2, each neuron voltage membrane s^n integrates sensory bottom-up

information $v_B^n \sim w_n \cdot s^{n-1}$ and top-down error signals $v_A^n \sim w_{n+1}{}^\top \cdot \frac{\partial \mathcal{L}}{\partial s^{n+1}}$. Therefore, each neuron needs two dendritic compartments to integrate the basal voltage v_B^n and the apical voltage v_A^n. Using the terminology given before, credit is assigned to apical dendritic voltage in this framework.

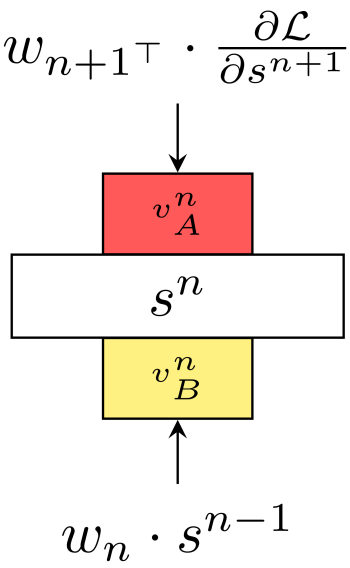

Figure 4.2: **Dendritic models**. Each neuronal layer of membrane potential s^n integrates bottom-up signals through basal compartments of voltage v_B^n and top-down signals through apical compartments of voltage v_A^n. Dendritic models and associated algorithms are designed so that basal and apical compartments integrate data information ($v_B^n \sim w_n \cdot s^{n-1}$) and error information ($v_A^n \sim w_{n+1}{}^\top \cdot \frac{\partial \mathcal{L}}{\partial s^{n+1}}$) respectively.

SpikeGrad [88] is an example of a bi-compartment model where integrate-and-fire neurons process bottom-up input spikes and error-discretized spikes in separate voltage compartments, with the exact same kind of dynamics for forward and backward passes. However, it requires to retain the neural activations of the forward passes to compute surrogate derivatives and gradients requested for the weight update and somehow remains close to standard backpropagation by requiring two phases, where each neuron processes successively bottom-up (data), then top-down (error) information. It is very likely though that in real biological systems, neurons can handle both kind of inputs at the same time, so that learning proceeds in one phase only! How a neuron can distinguish "pure" error top-down signals from self-generated top-down signals is still a very interesting open question.

In this purpose, Sacramento et al have proposed a dendritic micro-circuit which learns to cancel these intrinsic top-down inputs through the use of interneurons: whenever an external error signal appears, it cannot be explained away by the micro-circuit so that apical dendritic compartments perfectly encode an error signal [89]. Another study showed that the dendritic activity was reflected in the probability that a burst of spikes occurs, suggesting that top-

down error signals are encoded as bursts and bottom-up signals as single spikes [90].

4.3.5 Temporal credit assignment

Another approach to credit assignment is to compute error gradients *through time*. Biological neuronal networks are dynamical systems governed by temporal equations which, from the deep learning viewpoint, can be classified in the broader class of Recurrent Neural Networks (RNNs) where the computional graph is both deployed in space (from one layer to another) and time (from a time step to the next one). Applying backpropagation in such a neural network therefore amounts to go backward in time, which is why backpropagation in this context is more specifically called Backpropagation Through Time (BPTT) [91].

However, one of the goal of temporal credit assignment is to avoid going backward in time to compute error gradients but rather compute them in a forward-time fashion. Real Time Recurrent Leaning (RTRL) [92] is one of the earliest proposals of forward-time computation of the gradient provided by backpropagation through time, which precludes the need to store activations of the neurons at each time step of the forward phase. Recurrent Backpropagation (RBP) [93, 94] can be seen as one particular case of backpropagation through time when it is applied to convergent RNNs which reach a steady state: the error signal is backpropagated through this steady state so that the computation of the error jacobians does not require anything else but the value of this steady state.

Very recently and in the same spirit as RTRL, Eligibility Propagation (e-prop) [95] was proposed as a biologically plausible technique to approximate BPTT gradients in an online and forward-time fashion in spiking networks, using the combined trick of synaptic eligibility traces to capture long-term dependencies and approximating spikes by differentiable proxies [96]. In spite of the tremendous potential of e-prop with regards to neuromorphic computing, although the gradient computation happens online, it cannot be conducted by the system itself: from a hardware prospective, it would require an external circuitry and subsequent overhead.

Equilibrium Propagation [97], the central algorithm of this thesis, is a variant of Contrastive Hebbian Learning where the target y is weakly clamped to the output neurons $\hat{y}$ through a small nudging strength β. Albeit the simplicity of this modification of Contrastive Hebbian Learning, Equilibrium Propagation is endowed with strong theoretical guarantees for any network topology*. The core idea of Equilibrium Propagation is to first let the system reach a steady state given a static input on the visible layer. Then, the error in the output layer is encoded as a force which drives the whole system towards lower loss: because the system

*Part III is dedicated to introducing thoroughly Equilibrium Propagation and its variants.

initially does not move, subsequent motion encodes gradients. Heuristically if neurons are described by a variable s and the loss of interest is denoted L: $\dot{s} \sim \frac{\partial \mathcal{L}}{\partial s}$. Moreover, the learning rule prescribed by Equilibrium Propagation, which is rigorously shown to optimize the loss, is *spatially local*, an important feature for on-chip learning as we emphasized before. Therefore, the gradients of the loss are computed by the physics of the system itself: the system sustains both inference *and* gradient computation, with a local learning rule, without calling for the need of an external circuitry! Consequently, Equilibrium Propagation is both hardware friendly *and* mathematically justified, therefore extremely promising to scale on-chip learning to deep neural networks.

4.4 Main results of this thesis

The objective of this thesis is to address the two fundamental components of on-chip learning with two biologically plausible learning algorithms. We investigate *the conductance update* component of learning on Restricted Boltzmann Machines, and *the gradient computation* component with Equilibrium Propagation. Restricted Boltzmann Machine training and Equilibrium Propagation bear some resemblance that will be highlighted in part III. More precisely, the contributions of this thesis are the following:

- We first present an empirical study of the use of memristive devices in Restricted Boltzmann Machines. We come up with programming strategies to mitigate device imperfections such as non-linearity, device-to-device and cycle-to-cycle variability, and to facilitate hyperparameter tuning (part II).

- We reformulate Equilibrium Propagation in a discrete-time setting and demonstrate its equivalence with Backpropagation Through Time mathematically and numerically. We propose a convolutional model that is trainable with our discrete-time version of Equilibrium Propagation, achieving best performance on MNIST ever reported in the literature of this algorithm. Finally, we show that our new formulation of Equilibrium Propagation enables a simulation speed-up by a factor 5 to 8. These results can potentially help prototype faster hardware-friendly implementations of Equilibrium Propagation (part IV).

- We extend Equilibrium Propagation to the biologically plausible and hardware-friendly situation where the learning rule becomes local in time: synapses are treated as a dynamical system that evolve along with neurons during the second phase of the algorithm, a new version of the algorithm that we call Continual Equilibrium Propagation. We demonstrate the equivalence of Continual Equilibrium Propagation with Backpropagation Through Time, and extend the algorithm to the situation where the connections between neurons are asymmetric. Finally, we show numerically that the more a model satisfies the theorem before training, the best its resulting training performance. These

results can provide an engineering guidance to map Equilibrium Propagation onto neuromorphic chips (part V).

- Finally, we discuss ongoing projects and future directions of research for the implementation of Equilibrium Propagation on neuromorphic hardware.

Part II

Restricted Boltzmann Machines with memristors

Summary

One of the biggest stakes in nanoelectronics today is to meet the needs of Artificial Intelligence by designing hardware neural networks which, by fusing computation and memory, process and learn from data with limited energy. For this purpose, we have seen that memristive devices are excellent candidates to emulate synapses. A challenge, however, is to map existing learning algorithms onto a chip: for a physical implementation, a learning rule should ideally be tolerant to the typical intrinsic imperfections of such memristive devices, and local. Restricted Boltzmann Machines (RBM), for their local learning rule and inherent tolerance to stochasticity, comply with both of these constraints and constitute a highly attractive algorithm towards achieving memristor-based Deep Learning. On simulation grounds, this part gives insights into designing simple memristive devices programming protocols to train on chip Boltzmann Machines. Among other RBM-based neural networks, we advocate using a Discriminative RBM, with two hardware-oriented adaptations. We propose a pulse width selection scheme based on the sign of two successive weight updates, and show that it removes the constraint to precisely tune the initial programming pulse width as a hyperparameter. We also propose to evaluate the weight update requested by the algorithm across several samples and stochastic realizations. We show that this strategy brings a partial immunity against the most severe memristive device imperfections such as the non-linearity and the stochasticity of the conductance updates, as well as device-to-device variability.

Introduction

As we pointed it out in part I, fast progress in machine learning and big data processing make conventional electronics hardware unable to cope with it in the long run, and calls for breakthrough in artificial intelligence hardware design. Memristive devices are particularly exciting in this regard, as they can emulate synapses when arranged into crossbar arrays with interconnecting transistors acting as neurons [98–101] (see Fig. 2.3 of part I), which could lead to hardware neural networks with an outstanding energy efficiency.

As we mentionned it in subsection 3.3.1 of part I, such hardware neural networks can be trained ex situ: the synaptic weights are optimally determined on conventional central or graphical processing units, and then transferred onto memristive hardware [102, 103]. Nevertheless, the most exciting applications could come from systems with a capability of learning. However, such in situ learning comes with two major challenges: the programmation of memristive devices (section 3.2 of part I), and the learning rule implemented itself (section 3.3 of part I). For self-containedness of this part, we remind the main features of these two challenges.

First, programming the conductance of memristive device very precisely is difficult, due to well-known memristive device imperfections, such as non-linear conductance response, cycle-to-cycle and device-to-device variability. In the case of ex situ learning, this difficulty can be avoided by using complex tuning protocols. But in the case of in situ learning, such tuning protocols cannot be used as devices need to be reprogrammed repeatedly throughout learning.

The second challenge of in situ learning is the non-locality of most neural network learning rules. This is the case of backpropagation (see subsection 3.3.1 of part I) whose prescribed weight update does not solely depend on the pre- and post- synaptic neurons. Local learning rules can conversely be conveniently implemented on hardware with memristive devices which can be programmed by the voltage difference created by the pre- and post- synaptic neurons (see subsection 3.3.2 of part I). For this reason, although its theoretical implementation with

memristive devices has been extensively studied [104–110], most demonstrations of memristive in situ learning hardware is single layer, when backpropagation becomes local [102, 111].

In this part, we investigate the possibility to perform in situ learning circumventing these two challenges entirely. For this purpose, we propose implementing variations of Restricted Boltzmann Machines (RBMs) that allow in situ learning with a local learning rule, and where memristive device programming can be achieved in a very simple way. RBMs were introduced in the previous part (section 4.2 of part I) and have mostly found applications in pattern detection [112,113]. In software, RBMs often underperform with regards to the most sophisticated deterministic neural networks on benchmark data sets [114]. However, they appear extremely attractive with regards to our two challenges. They can indeed be trained with Contrastive Divergence [115], a spatially local learning rule. Also, their intrinsically stochastic nature suggests that they could be appropriate to learn in an approximate setting. Existing works on memristive RBMs [116–120] mainly focused on the CMOS circuitry to implement the neurons [116,118], matrix multiplication and summation [118], Gibbs sampling [120], neuron value centering when adding depth [119]. In our study, we perform simulations and propose methods for achieving in situ learning. We focus in particular on the impact of the conductance update physics, and of the tuning of hyperparameters, a critical question for in situ learning.

We first introduce a baseline memristor-based gradient descent algorithm taking only the sign of the gradient into account. We use this algorithm to train the three most encountered RBM-based architectures in the neuromorphic literature on the hand-written digit classification task (MNIST [121]) with typical values of the device parameters to identify the most relevant algorithm. In the second part of the paper, we show that reducing the variance of the gradient estimate provided by Contrastive Divergence improves the performance of the RBM with non-linear devices. Pointing out the necessity to hand-tune the pulse width in the baseline algorithm, we come up with a programming pulse width selection based on the sign of two consecutive weight updates inspired from Resilient Propagation [122–124] which enlarges the range of eligible pulse widths by up to two decades. Finally, we combine the two techniques introduced and analyze the effect of variability on the RBM. We conclude by lessons taught by these results. These results were published in Scientific Reports and the present part is adapted from this publication [125].

Chapter 1

Background

1.1 Restricted Boltzmann Machines

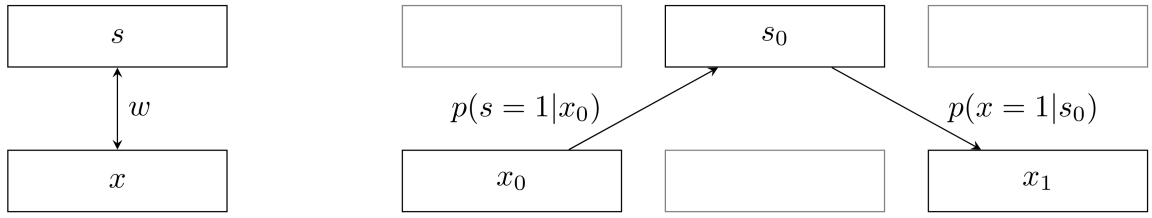

Figure 1.1: **Restricted Boltzmann Machine (RBM)**. Left: RBM toplogy, with a visible layer x and a hidden layer s bidirectionally connected through w. Right: Contrastive Divergence. Given data x_0 clamped to the visible layer, the hidden neurons are sampled from the Bernoulli distribution $s_0 \sim p(s = 1|x_0)$. Afterwards given the samples s_0, the visible layer is sampled from $x_1 \sim p(x = 1|s_0)$.

A Restricted Boltzmann Machine (RBM) is a stochastic neural network which learns to generate a data set. In such a network, the neural dynamics are governed by an energy landscape. After learning, the minima of the energy should correspond to the data set samples (see Fig. 4.1 on Contrastive Hebbian Learning in part I): neurons evolve towards a state that accounts for the data. With notations consistent with the introduction of this thesis, the data is presented to visible units, denoted by x and the other neurons, called hidden neurons and denoted by s, are correlated to the visible units through the weights w and evolve accordingly. Visible and hidden units may also be influenced by a constant input which we model by a bias, respectively b_{x} and b_{s}. Formally, we can write the energy associated to such a system as:

$$E(x, s; w, b_{\mathrm{x}}, b_{\mathrm{h}}) = -s^{\top} \cdot w \cdot x - b_{\mathrm{s}}^{\top} \cdot s - b_{\mathrm{x}}^{\top} \cdot x, \tag{1.1}$$

In practice, b_{x} and b_{s} are concatenated to w as an extra column and row, respectively, so that we absorb their definition into w without loss of generality. In accordance with the previous notations, we denote $\theta = \{w, b_{\mathrm{h}}, b_{\mathrm{v}}\}$. Neurons have binary values $\{0, 1\}$, which are samples of the joint distribution:

$$p(x, s; \theta) = \frac{\exp(-E(x, s; \theta))}{\sum_{\tilde{x}, \tilde{s}} \exp(-E(\tilde{x}, \tilde{s}; \theta))} \tag{1.2}$$

Running the neural dynamics amounts to sampling this distribution. Once the neural network is trained, such sampling is able to regenerate the data set. Learning is achieved by gradient ascent on the log-likelihood $\log p(x; \theta) = \sum_{\tilde{s}} p(x, \tilde{s}; \theta)$. Denoting $\mathcal{Z} = \sum_{\tilde{x}, \tilde{s}} \exp(-E(\tilde{x}, \tilde{s}; \theta))$, the computation reads:

$$\begin{aligned}
\Delta\theta &\propto \frac{\partial \log p(x; \theta)}{\partial \theta} \\
&= \underbrace{\frac{\mathcal{Z}}{\sum_{\tilde{s}} \exp(-E(x, \tilde{s}))}}_{=\frac{1}{p(x)}} \sum_{\tilde{s}} -\frac{\partial E}{\partial \theta}(x, \tilde{s}) \underbrace{\frac{\exp(-E(x, \tilde{s}))}{\mathcal{Z}}}_{=p(x, \tilde{s})} + \sum_{\tilde{x}, \tilde{s}} \frac{\partial E}{\partial \theta}(\tilde{x}, \tilde{s}) \underbrace{\frac{\exp(-E(\tilde{x}, \tilde{s}))}{\mathcal{Z}}}_{p(\tilde{x}, \tilde{s})} \\
&= \sum_{\tilde{s}} -\frac{\partial E}{\partial \theta}(x, \tilde{s}) p(\tilde{s}|x) + \sum_{\tilde{x}, \tilde{s}} \frac{\partial E}{\partial \theta}(\tilde{x}, \tilde{s}) p(\tilde{x}, \tilde{s}) \\
&= -\left\langle \frac{\partial E}{\partial \theta} \right\rangle_{\mathrm{data}} + \left\langle \frac{\partial E}{\partial \theta} \right\rangle_{\mathrm{model}}
\end{aligned}$$

where $\langle \cdot \rangle_{\mathrm{data}}$ and $\langle \cdot \rangle_{\mathrm{model}}$ denote a data average and a model average respectively. Taking the specific form of the energy given by Eq. (1.1), we finally get:

$$\begin{cases}
\Delta w_{ij} \propto \langle s_i x_j \rangle_{\mathrm{data}} - \langle s_i x_j \rangle_{\mathrm{model}} \\
\Delta b_{\mathrm{h,i}} \propto \langle s_i \rangle_{\mathrm{data}} - \langle s_i \rangle_{\mathrm{model}} \\
\Delta b_{\mathrm{v,j}} \propto \langle v_j \rangle_{\mathrm{data}} - \langle v_j \rangle_{\mathrm{model}}
\end{cases} \tag{1.3}$$

In Eqs. (1.3), computing the data statistics $\langle \cdot \rangle_{\mathrm{data}}$ is straightforward: the posterior $p(s|x)$ is a Bernoulli distribution that makes inference tractable. However, computing the model statistics $\langle \cdot \rangle_{\mathrm{model}}$, which boils down to sampling the joint distribution $p(x, s)$ is much more of a challenge. An approach to estimate the model statistics is provided by Constrastive Divergence [115]. More precisely, Contrastive Divergence provides a biased estimate of the gradient of the likelihood $\log p(x; \theta)$. The principle of this algorithm is to update the synaptic weights of the neural networks w_{ij} through:

$$\Delta w_{ij} \propto x_j(0)s_i(0) - x_j(1)s_i(1) \tag{1.4}$$

States "0" and "1" refer to the step of a "Gibbs chain", used to produce samples from the model. In step 0, the state of hidden neurons $s(0)$ is sampled based on the state of input neurons $x(0)$, clamped to a training example: $s_i(0) \sim p(s_i = 1|x_j(0))$ where '$\sim$' means 'is sampled from'. In step 1, the state of input neurons $x(1)$ is sampled based on the previous state of hidden neurons ($x_j(1) \sim p(x_j = 1|s_i(0))$), and the state of the hidden neurons $s(1)$ is sampled a second time based on the new state of the input neurons ($s_i(1) \sim p(s_i = 1|x_j(1))$). Note that $p(s|x) = \sigma(w \cdot x + b_s)$ and $p(v|s) = \sigma(w^\top \cdot s + b_x)$ with $\sigma(x) = 1/(1 + \exp(-x))$ so that the activation function used in a Restricted Boltzmann Machine is the usual sigmoid function - see the excellent introduction to Restricted Boltzmann Machines by Asja Fischer for derivation details [126]. The most distinctive feature of Contrastive Divergence is its spatial locality. Unlike the backpropagation rule use for conventional forms of neural networks, the update to synaptic weight w_{ij} only depends on information about the two neurons i and j to which the synapse is connected.

1.2 Memristor model used and associated algorithm

All the simulations presented in this paper have been carried out at a level which highlights the effects of the weight update physics and the learning rules it enables on the different neural network architectures introduced thereafter.

The following model [127] for the memristive devices was used:

$$\frac{dG(t)}{dt} = \begin{cases} C_p \exp\left(-\beta_p \frac{G(t)-G_{min}}{G_{max}-G_{min}}\right) & \text{(potentiation)} \\ -C_d \exp\left(-\beta_d \frac{G_{max}-G(t)}{G_{max}-G_{min}}\right) & \text{(depression)} \end{cases}, \tag{1.5}$$

applying Eq. (1.5) between t_0 and $t_0 + \Delta t$ yields the effective conductance update (whose explicit form is shown in the Methods):

$$G(t_0 + \Delta t) = G(t_0) + \int_{t_0}^{t_0+\Delta t} \frac{dG(t)}{dt} dt, \tag{1.6}$$

$G(t)$ denotes the conductance at time t of the device, with G_{max} and G_{min} being the maximal and minimal conductance, labels p and d referring to potentiation and depression respectively. Δt appearing in Eq. (1.6) defines the programming pulse width. Note that our memristor model implicitly takes into account the number of pulses applied to the device: it treats equally a programming pulse of width Δt or n programming pulses of width $\Delta t/n$. We also introduce Δt_{max} as the pulse width that is required to bring the conductance from G_{min}

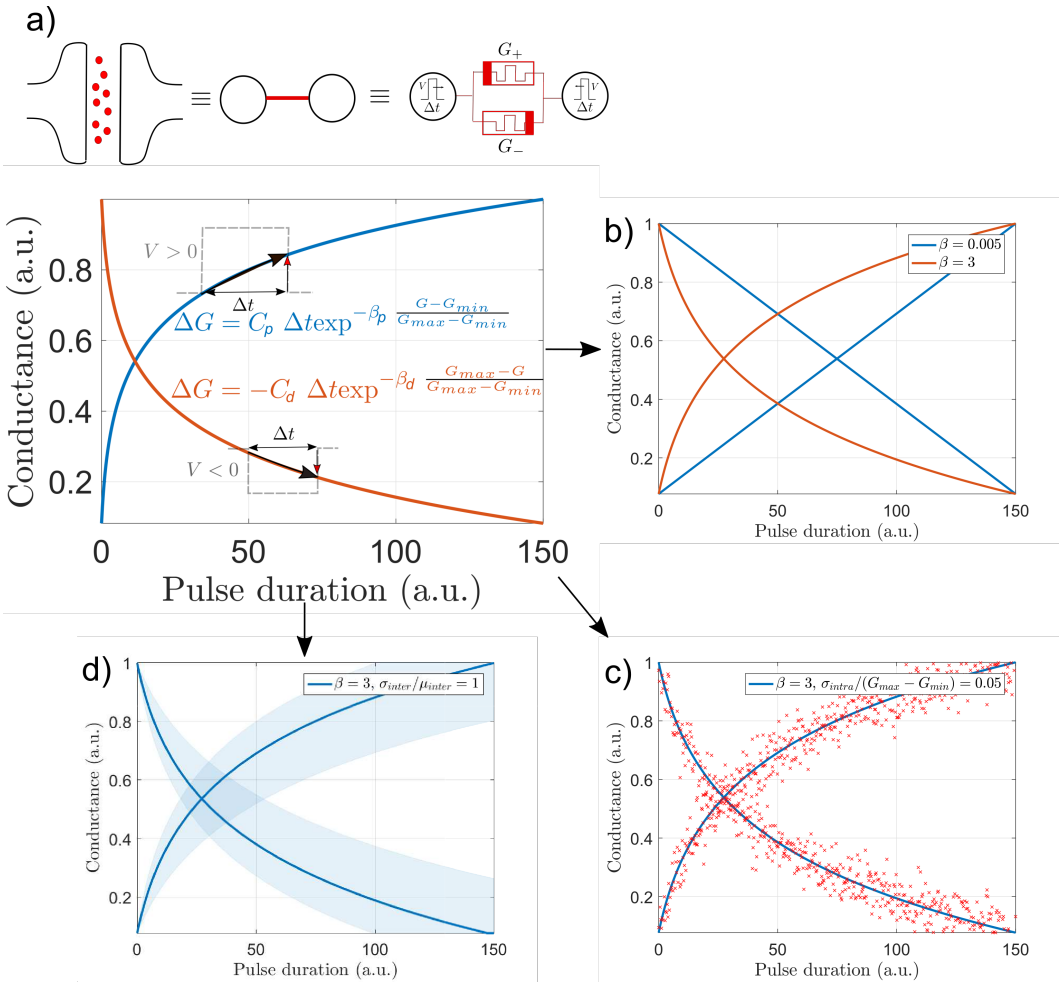

Figure 1.2: **Memristor model.** a) Each weight is implemented with two memristors, i.e. $W = G_+ - G_-$, conductance updates follow the memristor characteristic dictated by Eq.(1.5). b)-d) Illustration of the memristive imperfections taken into account: b) non-linearity (conductance dependent update), c) cycle-to-cycle variability (each red dot represent one stochastic realization of a conductance update), d) device-to-device variability (from left to right).

to G_{max}. C_p and C_d, which encode the amplitude of the voltage difference applied to the device, is fixed to ensure this last condition (see Methods for details). β_p and β_d model the dependence of the conductance update with the current conductance, namely the non-linearity of the device so that if $\beta_p = 0$, dG/dt is constant for potentiation. This model can be used to describe practical memristive devices [98, 127–129]. Our model is similar in form to existing model [130], as further discussed in Appendix 1.1.

In our work, we assume that each model parameter or weight w is carried by two memristive devices of conductance G_+ and G_-, so that $w = G_+ - G_-$ [104] - see Fig. 1.2. In the light of

these notations and for most of our simulations, we assumed that the non-linear parameter β and the multiplicative factor C were the same not only between two devices of the same synaptic pair, but also for potentiation and depression - in the absence of device variation, see below. Depending on the technology used, G_+ and G_- can only be increased [104, 131], which our simulation framework can handle. In most cases, a learning algorithm prescribes an update Δw, given by the gradient of a loss function for instance or a proxy (as it is the case for contrastive divergence)

$$w \leftarrow w + \alpha \Delta w, \tag{1.7}$$

where α and Δw are the learning rate and the weight update respectively. However, incrementing precisely $w = G_+ - G_-$ of the amount $\alpha \Delta w$ along the memristor characteristics Eq. (1.5) is extremely impractical: it requires to temporarily store the gradient value, read out the weight values and adjust the programming pulse width accordingly. So in this first section, we only take the sign of the gradient into account at each learning step and we apply identical pulses with width Δt according to a simple heuristic, described in Alg. 1: whenever the desired weight change Δw is positive (negative), we increase (decrease) G_+ and decrease (increase) G_- by applying a pulse of duration Δt. Note in Alg. 1 that the conductance update reads as $G_{ij} \leftarrow G_{ij} + f_{p,d}(G_{ij}, \Delta t)$ with f_p and f_d respectively denoting the effective potentiation and depression occuring over a pulse width Δt -see Appendix 1.1 for details.

Algorithm 1 Memristor based gradient descent algorithm (**Cst**)

Input: $\{w_{ij}\}$, $\{x^{(n)}\}$ (training set), Δt (pulse width)
Output: $\{w_{ij}\}$

```
for each sample x do
    for each weight w_ij do
        Compute Δw_ij
        if sign(Δw_ij) > 0 then
            Increase G_+,ij and decrease G_−,ij with Δt
        else
            Decrease G_+,ij and increase G_−,ij with Δt
        end if
        w_ij ← G_+,ij − G_−,ij
    end for
end for
```

The pulse width monitors the speed of learning, and therefore has to be tuned. Alg. 1 has been called the Manhattan Rule [122], Unregulated Step Descent [132] or Stochastic Sign Descent (SSD) [133]. We herein call it Cst to refer to the fact that the programming pulse width is constant throughout learning.

In this work, we focus on the impact of non-linearity, cycle-to-cycle variability and device-

to-device variability on the resulting performance. Non-linearity is parametrized by β in Eq. (1.5), we modeled cycle-to-cycle variability by adding a Gaussian noise to each conductance update, and device-to-device variability by a dispersion on the multiplicative factor C appearing in Eq. (1.5). In this case, one given device may not respond symmetrically to potentiation and depression, or devices of the same pair may not respond symmetrically to potentiation (see Appendix 1.1).

Chapter 2

Results

2.1 Resilience of RBM-based architectures trained with constant programming pulse width

The impact of RBM-based network topology has not been extensively investigated from a neuromorphic viewpoint [116, 117, 119]: a direct comparison of the influence of the position of the labels (i.e. placed in the visible layer or in a separate output layer) or of the depth of the network (i.e. stacking several RBMs) on the resulting performance with different device parameters has not yet been carried out. Our goal is to compare different RBM-based architectures on the same learning task in terms of their resilience to device imperfections. We now present the results obtained when training — under the Cst algorithm with typical device parameters — the three most encountered RBM-based architectures in the neuromorphic literature on the MNIST discrimination task (see Table 2.1 and Fig. 2.1).

- The first one is a simple Restricted Boltzmann Machine (RBM) topped by a softmax classifier [116, 119] ("RBM+softmax"), with labels placed at the end of the network as the output of a classifier. In this architecture, the connections between input and hidden neurons, and output and hidden neurons are learned independently.

- The second is a Discriminative Restricted Boltzmann Machine [117,134] taking as inputs both the picture and the associated label ("Discriminative RBM"). This architecture is expected to outperform the simple RBM, as the connections between input and hidden neurons, and output and hidden neurons are learned jointly.

- Finally, we simulate a Deep Belief Net consisting in a stack of two RBMs topped by a Discriminative RBM ("Deep Belief Net", or DBN) [119, 120]. As this architecture features three layers of hidden neurons, it is expected to be able to learn more difficult tasks than the other two architectures.

The three architectures are depicted in Fig. 2.1. A thorough description of the network hyperparameters and the methodology can be found in the Methods part. Throughout this part, in contrast with most studies on the multi-layer perceptron, neurons are encoded with binary values at train and test time, and not real values. Moreover, as we restrict our study to local learning rules, the Deep Belief Net has only been trained two-layer wise as a stack of independent RBMs (i.e. "greedy learning" [135]), with no additional joint training with backpropagation (i.e. "fine-tuning"). Apart from the softmax classifier, all the architectures are trained using Contrastive Divergence. If not stated otherwise, the mini-batch size is set to 100.

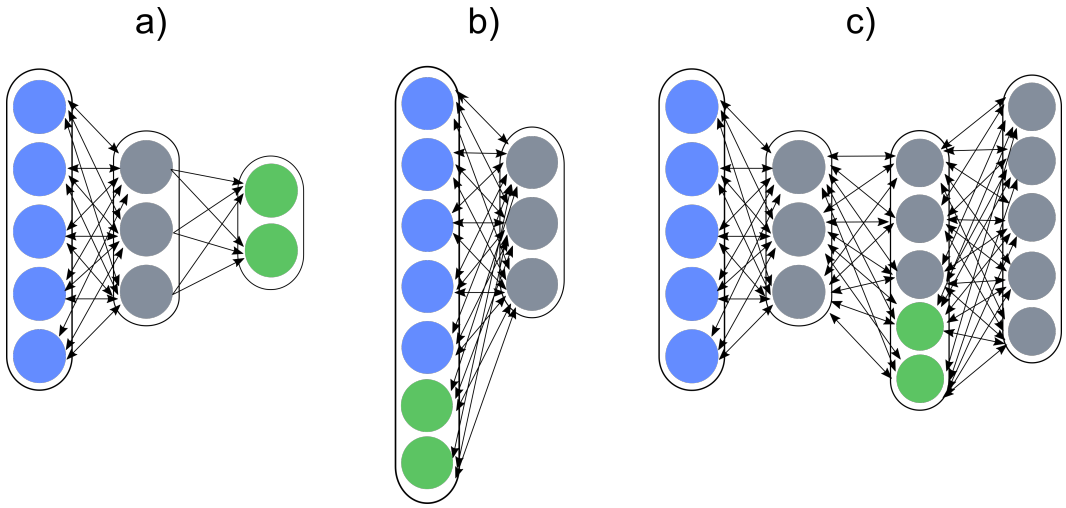

Figure 2.1: **Architectures under study**. a) a Restricted Boltzmann Machine topped by a softmax classifier ("RBM+softmax"), b) a Discriminative Restricted Boltzmann Machine ("Discriminative RBM"), c) a Deep Belief Net. Blue, grey and green filled circles stand for visible, hidden and label neurons respectively.

Table 2.1 lists the mean optimal performance over five trials of the three networks on the test set with typical device parameters. For each set of device parameters, we tuned the pulse width Δt until achieving the best performance: we denote the optimal pulse width for a given set of device parameters by Δt^*. To make sense of our simulation results, we also performed floating point standard gradient descent simulation results, referred to as "software-based" in Table 2.1. In this situation, as an example, a Discriminative RBM achieves over five trials $6.6 \pm 0.3\%$ test error with 300 hidden units. This error rate can be reduced by the use of larger neural networks. With 500 hidden units, the Discriminative RBM achieves $5.4 \pm 0.2\%$, and $3.6 \pm 0.2\%$ with 6,000 hidden units which, up to the choice of hyperparameters, is akin to state-of-the-art for this type of architecture [114].

In this non-memristive floating point software-based training, the Deep Belief Net outperforms the other two networks, as one would expect. When using memristors, the near-linear case ($\beta = 0.005$) yields the best results for the three architectures compared to the non-linear case ($\beta = 3$), as it has been extensively observed on multi-layer perceptrons [104, 109, 136].

		RBM+softmax	Discriminative RBM	Deep Belief Net
Topology		785-301-10	795-301	785-501-511-2001
Software-based	Test error	$7.0 \pm 0.5\%$	$6.6 \pm 0.3\%$	$5.7 \pm 0.1\%$
$\beta = 0.005$	Test error	$8.3 \pm 0.1\%$	$6.4 \pm 0.2\%$	$6.6 \pm 0.2\%$
	$\frac{\Delta t^*}{\Delta t_{\max}}$	$\frac{1}{1000}$	$\frac{1}{1000}$	$\frac{1}{150}$
$\beta = 3$	Test error	$17.3 \pm 0.3\%$	$15 \pm 0.1\%$	$28.4 \pm 0.3\%$
	$\frac{\Delta t^*}{\Delta t_{\max}}$	$\frac{1}{5000}$	$\frac{1}{5000}$	$\frac{1}{10000}$
Cycle-to-cycle variability	Test error	$15.1 \pm 0.4\%$	$11.9 \pm 0.5\%$	$9.2 \pm 0.3\%$
Device-to-device variability	Test error	$20.3 \pm 0.3\%$	$13.9 \pm 0.5\%$	$22.6 \pm 0.5\%$

Table 2.1: Test error rate achieved by the three architectures under study on MNIST with typical values of the device parameters in terms of non-linearity, cyle-to-cycle and device-to-device variability. Cycle-to-cycle variability is taken as $\frac{\sigma_{\text{intra}}}{(G_{max}-G_{min})} = 6 \cdot 10^{-3}$ with $\beta = 0.005$. Device-to-cycle variability is taken as $\left(\frac{\sigma}{\mu}\right)_{\text{inter}} = 1$ with $\beta = 0.005$. Each topology includes the bias. Each simulation was performed over 30 epochs with a mini-batch size of 100, we indicate the mean error rate and the variance over five trials.

Interestingly, the Discriminative RBM achieves the lowest test error rate. It is not surprising that the RBM topped by a classifier may not do as well as the Discriminative RBM, as nothing ensures the features extracted by the RBM to be discriminative [114]. By contrast, it is surprising at first sight that the benefits of depth with the Deep Belief Net are not observed as in the floating point software-based training: the Deep Belief Net performs similarly to the Discriminative RBM when using near-linear memristors. However, the shape of the features accounts for these discrepancies. In Fig. 2.2, we display a 5×5 grid of gray-scale pictures, each of which representing the values of the 784 weights connecting the visible layer to a given hidden unit: each picture represents what is seen by one hidden unit, thus giving a direct insight into the features extracted by this hidden unit from the data. As seen per Fig. 2.2, while the features learned by a standard RBM (i.e. with a proper gradient descent) are sharply defined stroke-like features [137], those learned by a memristive Discriminative RBM with the Cst algorithm are coarser. This may explain why stacking several memristive RBMs may not help for subtle features extraction and subsequently improved performance.

This adds up to the fact we did not fine-tune the Deep Belief Net with backpropagation.

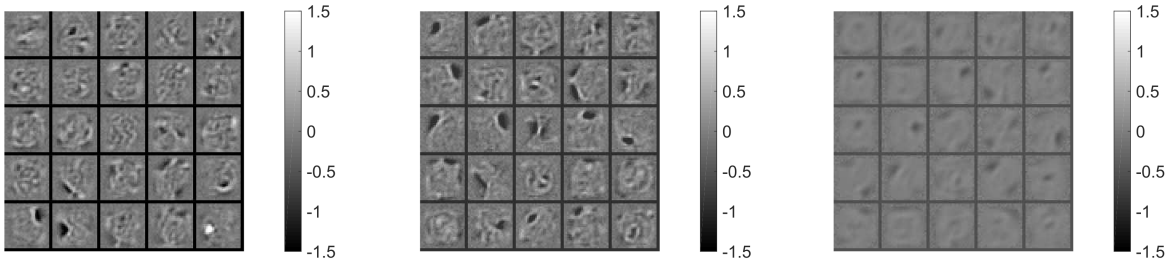

Figure 2.2: **Feature extraction**. Examples of hidden features extracted by, from left to right: standard RBM (trained with a learning rate of 0.05), a memristive Discriminative RBM (trained under Cst with $\beta = 0.005$, $\Delta t/\Delta t_{max} = 1/1000$), another memristive Discriminative RBM (trained under Cst with $\beta = 3$, $\Delta t/\Delta t_{max} = 1/5000$). Each gray-scale picture represents the values of the 784 weights connecting the visible layer to a given hidden unit.

In the non-linear case ($\beta = 3$), the RBM topped by a softmax and the Discriminative RBM test error rates jumps by $\sim 9\%$ compared to $\sim 22\%$ for the Deep Belief Net. We can account for this observation with the corruption of the extracted features which, as seen from Fig. 2.2, is even more pronounced than in the linear case. As these corrupted features are fed into the next RBM, this effect cumulates with depth. When passing extracted features from one RBM to the next one, one stochastic realization may consequently not be enough to transmit all the information contained by the features. Finally, the pulse width for which the networks are tuned at $\beta = 3$ is lower than in the $\beta = 0.005$ case: non-linearity drags the optimal pulse width to low values to accommodate the abrupt conductance update it triggers.

When taking cycle-to-cycle variability into account in the linear case ($\frac{\sigma_{\text{intra}}}{(G_{max}-G_{min})} = 6 \cdot 10^{-3}$, $\beta = 0.005$), the Deep Belief Net appears to be more resilient to the programming noise than the two other networks: its test error rate only jumps by $\sim 3\%$ against $\sim 6\%$ for the two other networks. This happens because the pulse width for which the first two networks are tuned ($\Delta t^*/\Delta t_{max} = 1/1000$) is lower than the one of the Deep Belief Net ($\Delta t^*/\Delta t_{max} = 1/150$), so that the deterministic component of conductance update better dominates the programming noise. This idea will be further developed in the last section.

The impact of device-to-device variability in the linear case ($\left(\frac{\sigma}{\mu}\right)_{\text{inter}} = 1$, $\beta = 0.005$) can also be interpreted in the light of the pulse width employed. As the coefficient carrying device-to-device variability comes in the memristor characteristic Eq. (1.5) as $C_{\pm}\Delta t$, the bigger Δt the bigger the effect of device-to-device variability, which may explain why the Deep Belief Net is less resilient in this regard than the Discriminative RBM: the test error rate achieved by the latter increases by $\sim 8\%$ compared to $\sim 16\%$ for the former. Although the RBM topped by a softmax and the Discriminative RBM use the same pulse width, the former network turns out to be less resilient. The Discriminative RBM optimizes the

joint probability of the inputs and labels so that device-to-device variability affects features extraction and classification consistently, which is not the case when RBM features are fed into an independent softmax classifier.

Overall, this comparative study reveals that the Discriminative RBM appears to be the best candidate architecture in terms of performance for typical values of the device parameters. We consequently focus our study in the rest of the part on the Discriminative RBM. Still, the best performance for a given set of realistic device parameters is not satisfactory enough, and it is achieved for a very narrow range of pulse widths around the optimum. In the next two subsections, we propose two intuitive solutions to deal with these two aspects respectively, and finally combine them in the last subsection.

2.2 Solutions mitigating device imperfections on the Discriminative RBM

2.2.1 Mitigating device non-linearity by reducing the variance of the gradient sign estimate

Gradient descent is inherently stochastic when dealing with a large data set. The first source of stochasticity comes from sampling a mini-batch of data drawn uniformly and independently from the data set and computing an approximate gradient over this mini-batch. A second source of stochasticity stems from Contrastive-Divergence itself, which relies on stochastic quantities, as seen in Eq. (1.4).

Most neuromorphic investigations on RBMs [116, 117, 119] exacerbate these two forms of stochasticity, as Contrastive Divergence is carried out sample by sample (that is with a mini-batch of size one) using one single stochastic realization per neuron. In this section, we investigate techniques to reduce the stochasticity. First, we sum Eq. 1.4 across several samples (i.e. mini-batches). Second, we sum it over multiple stochastic realizations (i.e. parallel Gibbs chains). This second strategy amounts to encoding neurons by their firing rate instead of a single spike, and is reminiscent of the rate-coded Contrastive Divergence of [138] or Event-driven Contrastive Divergence [139].

Fig. 2.3a) and 2.3b) show the test error rate as a function of the pulse width used for the Discriminative RBM trained under the Cst algorithm, in the linear and non-linear case, and with different mini-batch sizes and numbers of parallel Gibbs chains used for Contrastive Divergence.
In the linear case (Fig. 2.3a)), increasing the mini-batch size or the number of parallel Gibbs chains does not improve significantly the resulting performance. Conversely, when working

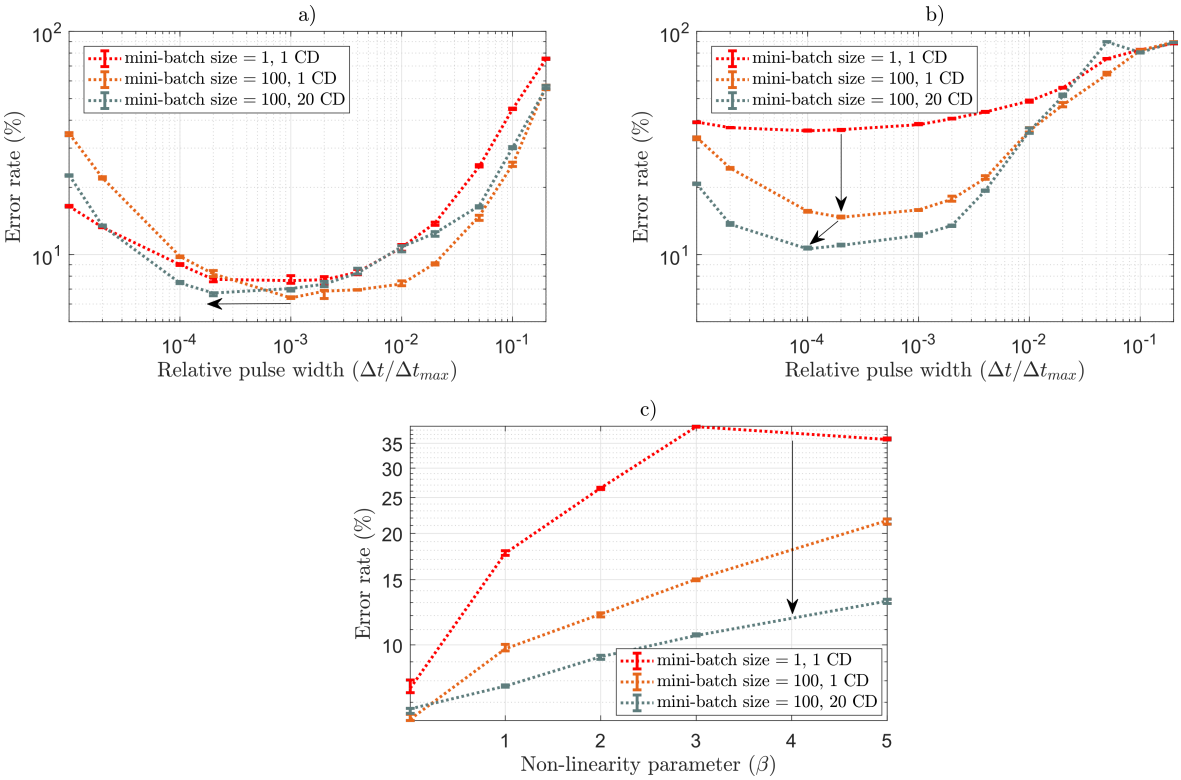

Figure 2.3: **Mitigating non-linearity.** a) Test error rate achieved by the Discriminative RBM as a function of the programming pulse width for different mini-batch sizes and different number of parallel Gibbs chains to evaluate the Contrastive Divergence term ('# CD') in the near-linear case ($\beta = 0.005$). b) Same Figure as a) in the non-linear case ($\beta = 3$).c) Optimal test error rate achieved by the Discriminative RBM for different values of β with different mini-batch sizes and different number of parallel Gibbs chains to evaluate the Contrastive Divergence term ('# CD'). For mini-batches size of 100, each simulation was ran over 30 epochs, 5 times per value of pulse width, error bars indicate median, first quartile and third quartile. For mini-batches of size 1, each simulation was ran over 50 epochs 5 times per value of pulse width to ensure convergence.

with non-linear devices (Fig. 2.3b)), decreasing the variance of the gradient estimate dramatically makes a difference. Decreasing the variance of the gradient estimate indeed helps the conductances to move into good directions, especially when the conductance increment is abrupt and uncontrolled in the non-linear case.

Moreover, the optimal pulse width is dragged towards smaller values when decreasing the variance of the sign of the gradient estimate: with a reduced variance and within a fixed number of epochs, the algorithm converges faster and subsequently selects a smaller learning rate. This could seem counter-intuitive, as in a standard gradient descent framework, the optimal mini-batch size is known to scale linearly with the learning rate [140]. However, this

analysis does not hold here upon only taking the sign of the gradient into account. Fig. 2.3c) shows for each value of β the best error rate achieved when using one or 20 parallel Gibbs chains, both with a mini-batch size of 100, supporting the above statement.

2.2.2 Facilitate pulse width tuning: Resilient Propagation (RProp)

Using a constant pulse width may not be optimal for several reasons. As the amplitude of the weight updates is directly monitored by the programming pulse width and amplitude, it has to be tuned with a hyperparameter selection by sweeping through different values. Also, when using identical pulses throughout learning, undesirably large weight updates may occur in conductance regions of high non-linearity, entailing weight dithering around optima [104]. Conversely with a pulse width that is too small, conductances may move too slowly for convergence to be achieved within a reasonable number of epochs. A natural solution is to drop the Manhattan rule by reading out the numerical value of the gradient itself and applying the number of pulses required [104, 109, 141], or emulating linearity with pulses consistent with the current conductance state [142, 143]. However, these solutions are very expensive in practice: they require reading the state of each memory device at each learning update. Here, we investigate a simpler to implement solution, which exploits information about neurons only. In a system, information about neurons is indeed much more readily available than information about the memory devices.

Interestingly, off-line conductance tuning protocols, gradually increasing the pulse width or voltage amplitude so long as we get closer to a conductance target or decreasing it otherwise [102,144], give some insights into appropriate on-line programming schemes. A mathematical generalization of this heuristic, consisting in increasing the learning rate so long as we get closer to an optimal model or decreasing it otherwise, is called Resilient Propagation [122–124] (RProp). Very recently, a RProp-like technique was proposed for training a memristive multi-layer perceptron and was named the "Local Gain Techniques" [145]. In this work, we take inspiration in an improved version of RProp with "weight back-tracking" (called RProp+ in [123]), which cancels conductance updates that overshot an optimal model, and subsequently reduces the pulse width.

A detailed description of the neuromorphic adaptation of RProp with weight back-tracking is presented in Alg. 2. Whenever the sign of the gradient remains the same, the pulse width is increased by a factor $\eta_+ > 1$ so long as it does not exceed the initial pulse width: $\Delta t_{ij} \leftarrow \min(\eta_+ \Delta t_{ij}, \Delta t(0))$. This condition emulates a learning rate decay from its initial value, as seen per Fig. 2.4. When a gradient sign flip is encountered, we cancel the last conductance change over the same pulse width, and decrease the pulse width for the next learning step by a factor $\eta_- < 1$. Note from Alg. 2 that we did not impose a minimal pulse width, we allow it to decay to zero. By construction, the pulse width is consequently bounded

Algorithm 2 Memristor based gradient descent algorithm (**RProp**)

Input: $\{w_{ij}\}$, $\{x^{(n)}\}$ (training set), $\{\Delta t(0)\}$ (initial programming pulse width)
Output: $\{w_{ij}\}$

for each sample x **do**
  for each weight w_{ij} **do**
    if $\Delta w_{ij}^{(n)} \Delta w_{ij}^{(n-1)} > 0$ **then**
      $\Delta t_{ij} \leftarrow \min(\eta_+ \Delta t_{ij}, \Delta t(0))$
      Adjust $G_{+,ij}$ and $G_{-,ij}$ according to $\text{sign}(\Delta w_{ij}^{(n)})$ with Δt_{ij}
    else if $\Delta w_{ij}^{(n)} \Delta w_{ij}^{(n-1)} < 0$ **then**
      Apply opposite conductance change over the same Δt_{ij}
      $\Delta t_{ij} \leftarrow \eta_- \Delta t_{ij}$ for the next learning step
      Set $\Delta w_{ij}^{(n)} = 0$
    else if $\Delta w_{ij}^{(n)} \Delta w_{ij}^{(n-1)} = 0$ **then**
      Adjust $G_{+,ij}$ and $G_{-,ij}$ according to $\text{sign}(\Delta w_{ij}^{(n)})$ with Δt_{ij}
    end if
    $w_{ij}^{(n+1)} \leftarrow G_{+,ij}^{(n+1)} - G_{-,ij}^{(n+1)}$
  end for
end for

by initial pulse width and zero: $\Delta t_{max} = \Delta t(0)$, $\Delta t_{min} = 0$. This weight-backtracking is meant to avoid penalizing twice the algorithm by overshooting a local optimum and not going back far enough to cancel the wrong conductance move, and is handled by the third logic case $\Delta W^{(n)} \Delta W^{(n+1)} = 0$ (see Alg. 2). In addition, the pulse width is bounded by the initial pulse width.

$$\Delta w_{ij}^{(n)} = N_{+,ij}^{(n)} - N_{-,ij}^{(n)}$$

$n-1$	$N_{+,ij}^{(n-1)} > N_{-,ij}^{(n-1)}$	$N_{+,ij}^{(n-1)} < N_{-,ij}^{(n-1)}$	$N_{+,ij}^{(n-1)} > N_{-,ij}^{(n-1)}$	$N_{+,ij}^{(n-1)} < N_{-,ij}^{(n-1)}$
n	$N_{+,ij}^{(n)} > N_{-,ij}^{(n)}$	$N_{+,ij}^{(n)} < N_{-,ij}^{(n)}$	$N_{+,ij}^{(n)} < N_{-,ij}^{(n)}$	$N_{+,ij}^{(n)} > N_{-,ij}^{(n)}$
Case	$\Delta w_{ij}^{(n-1)} \Delta w_{ij}^{(n)} > 0$		$\Delta w_{ij}^{(n-1)} \Delta w_{ij}^{(n)} < 0$	

Table 2.2: RProp table of truth for any mini-batch size and number of parallel Gibbs chains. In the five remaining cases: $\Delta W_{ij}^{(n-1)} \Delta W_{ij}^{(n)} = 0$. The notations are defined on the body text.

As seen in Table 2.2, in spite of the apparent complexity of the RProp, it can be handled easily when applied to Contrastive Divergence. In Table 2.2, we denote $N_{+,ij}^{(n)}$ the positive term of Contrastive Divergence, i.e. $v_j(0)h_i(0)$, and $N_{+,ij}^{(n)}$ the negative term, i.e. $v_j(1)h_i(1)$), summed across mini-batches and parallel Gibbs chains. The relative importance of $N_{+,ij}$ and $N_{+,ij}$ between two consecutive learning steps $n-1$ and n can be classified in the nine logic cases depicted in Table 2.2. From these nine cases, we can deduce the sign of the factor

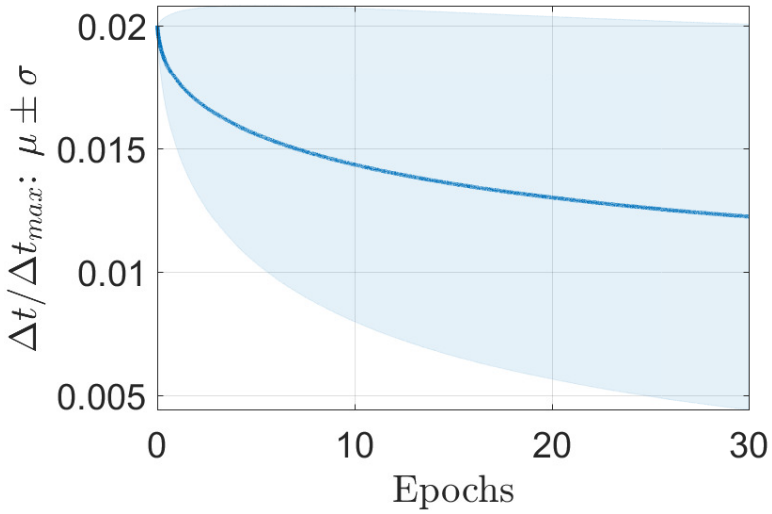

Figure 2.4: **RProp**. Typical time trace of the $\Delta t_{ij}/\Delta t_{max}$ statistics in terms of the mean value (blue plain line) and standard deviation (shaded blue region around the line).

$\Delta W_{ij}^{(n-1)} \Delta W_{ij}^{(n)}$, which is essential for RProp. This shows that our RProp-type rule can be implemented with knowledge about the neurons only, and relatively simple logics.

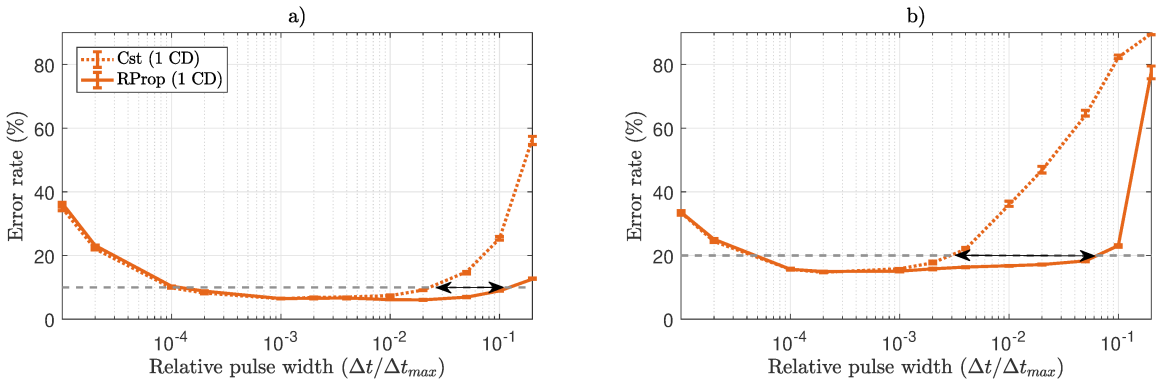

Figure 2.5: **RProp**. a) Test error rate achieved by the Discriminative RBM as a function of the programming pulse width when trained with Cst and RProp driven pulse widths for $\beta = 0.005$. b) Same as a) with $\beta = 3$. Grey dashed lines indicate 10% and 20% on the left and right panel respectively. Each simulation was ran over 30 epochs with a mini-batch size of 100, 5 times per value of pulse width, error bars indicate median, first quartile and third quartile.

Fig. 2.5 shows the comparative performance of the Discriminative RBM trained with Cst and our RProp rule, for varying initial pulse widths. In the linear case ($\beta = 0.005$), RProp allows achieving a test error that is lower than 10% for $\Delta t/\Delta t_{max} \in [10^{-4}, \sim 10^{-1}]$, compared to $\Delta t/\Delta t_{max} \in [10^{-4}, \sim 2.10^{-2}]$ when using the Cst algorithm. Similarly in the

non-linear case ($\beta = 3$), RProp allows achieving a test error that is lower than 20% for $\Delta t/\Delta t_{max} \in [\sim 5.10^{-5}, \sim 7.10^{-2}]$, compared to $\Delta t/\Delta t_{max} \in [\sim 5.10^{-5}, \sim 3.10^{-3}]$ when using the Cst algorithm. In this regard, RProp manages to extend the range of eligible pulse widths.

2.2.3 Resilience to cycle-to-cycle variability

In this subsection, we investigate the resilience of the Discriminative RBM to cycle-to-cycle variability and device-to-device variability using the two techniques introduced above. We restrict our study to the linear case ($\beta = 0.005$) to ensure that our results are not biased by non linearity.

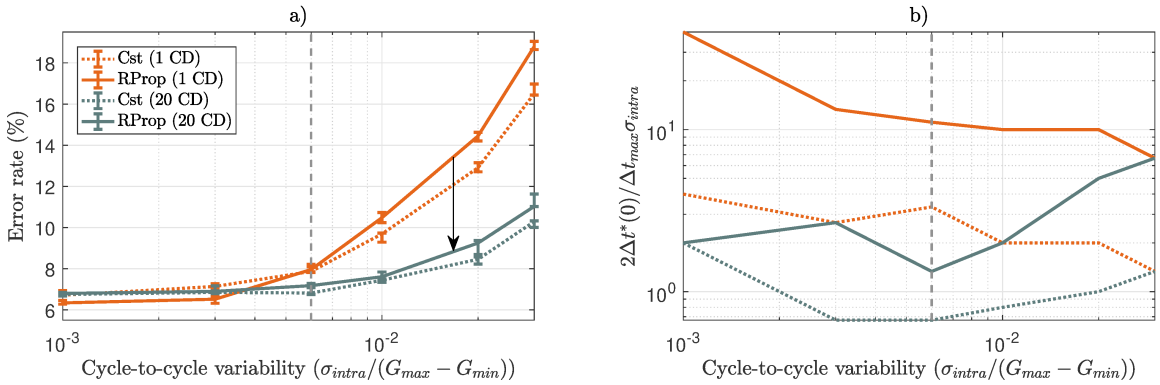

Figure 2.6: **Resilience to cycle-to-cycle variability**. a) Test error rate achieved by the Discriminative RBM as a function of cycle-to-cycle variability for every combination of the pulse width programming scheme (Cst, RProp) and number of parallel Gibbs chains used to evaluate Contrastive Divergence (# CD). b) Optimal conductance increment-to-noise ratio as a function of cycle-to-cycle variability associated with each curve of the left panel. When using 20 Gibbs chains (blue curves), from $\sigma_{intra}/(G_{max} - G_{min}) = 6.10^{-3}$ onwards (vertical gray dashed line) the conductance update overcomes the noise increase, accounting for the improved performance compared to the use of a single Gibbs chain (orange curves), regardless of the programming scheme. Each simulation was ran over 30 epochs with a mini-batch size of 100, 5 times per value of pulse width, error bars indicate median, first quartile and third quartile.

We present in Fig. 2.6a) the impact of cycle-to-cycle variability upon the performance of the Discriminative RBM trained under the four possible combinations of the training techniques studied before. As mentioned above, using longer programming pulses may be preferable in the presence of cycle-to-cycle variability. Therefore, we tuned learning for the best pulse width for each given noise intensity.
We first observe that using Cst or RProp only weakly changes the resilience to cycle-to-cycle variability. By contrast, using multiple Gibbs chains improves the performance by $\sim 6\%$ with the maximal amount of cycle-to-cycle variability. This result might seem initially surprising, as the systems with 20 Gibbs chains are tuned at a smaller pulse width than the systems

with one Gibbs chain at low noise, thus intuitively more sensitive to noise.
To facilitate the cycle-to-cycle variability analysis, we present the conductance increment-to-noise ratio in the linear case $2C\Delta t^*(0)/(\sigma_{intra}(G_{max} - G_{min})) = 2\Delta t(0)/(\Delta t_{max}\sigma_{intra})$, computed for each level of noise and optimal pulse width $\Delta t^*(0)$, in Fig. 2.6b). When using a single Gibbs chain, the conductance increment-to-noise ratio steadily decreases when noise increases: the noise increase dominates the conductance update. By contrast, when using 20 Gibbs chains, this parameter increases with noise from $\sigma_{intra}/(G_{max} - G_{min}) = 6.10^{-3}$ onwards: the conductance update starts to overcome the noise increase. This value corresponds to the level of noise for which a clear difference appears between on and 20 Gibbs chains in Fig. 2.6a), so that our analysis in terms of the conductance increment-to-noise ratio relevantly accounts for this discrepancy.
All of these observations boil down to how much the pulse width can be increased to absorb noise: what matters is not the pulse width for which the algorithms are tuned in the absence of noise, but how much it can be increased from there to absorb noise. Consequently, it is precisely because the systems using 20 Gibbs chains are tuned at a smaller pulse width than (Cst, 1 CD), and itself smaller than (RProp, 1 CD) that the former are more resilient to noise than the latter.

2.2.4 Resilience to device-to-device variability

While analyzing the impact of cycle-to-cycle variability on the performance involves many phenomena, understanding the impact of device-to-device variability for the four different schemes is straightforward and similar to the analysis carried out in Table 2.1: the larger the pulse width, the bigger the impact of device-to-device variability on the performance. Fig. 2.7 shows that the schemes using 20 parallel Gibbs chains are more robust to device-to-device variability than the scheme with Cst driven pulse widths and 1 Gibbs chain, itself more robust than its RProp counterpart, which is directly accounted by their respective pulse widths. Table 2.3 summarizes the results obtained with all the possible combinations of the techniques used and for typical values of the device parameters.

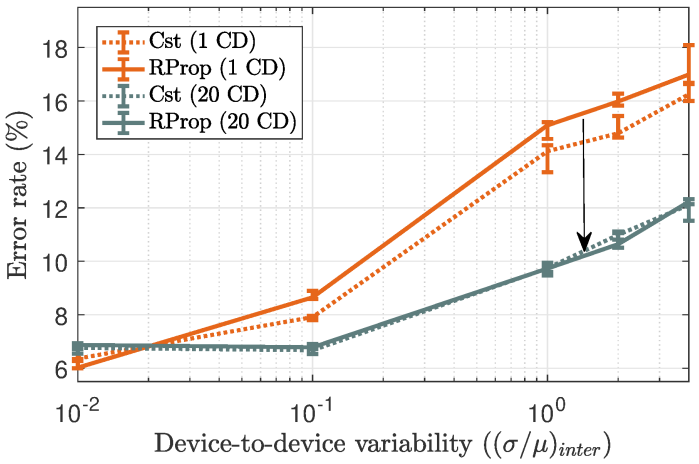

Figure 2.7: Resilience to device-to-device variability. Test error rate achieved by the Discriminative RBM as a function of device-to-device variability for every combination of the pulse width programming scheme (Cst, RProp) and the number of parallel Gibbs chains (# CD). Each simulation was ran over 30 epochs with a mini-batch size of 100, 5 times per value of pulse width, error bars indicate median, first quartile and third quartile.

		1 CD, Cst	1 CD, RProp	20 CD, Cst	20 CD, RProp
$\beta = 0.005$	Test error	$6.4 \pm 0.1\%$	$6 \pm 0.2\%$	$6.7 \pm 0.1\%$	$6.7 \pm 0.2\%$
	$\frac{\Delta t^*}{\Delta t_{\max}}$	$\frac{1}{1000}$	$\frac{1}{50}$	$\frac{1}{5000}$	$\frac{1}{5000}$
	$\frac{\Delta t}{\Delta t_{\max}}$ s.t. Test error $< 10\%$	$[10^{-4}, 2.10^{-2}]$	$[10^{-4}, 10^{-1}]$	$[5.10^{-5}, 10^{-2}]$	$[5.10^{-5}, 5.10^{-2}]$
$\beta = 3$	Test error	$14.7 \pm 0.2\%$	$14.9 \pm 0.2\%$	$10.7 \pm 0.2\%$	$10.5 \pm 0.2\%$
	$\frac{\Delta t^*}{\Delta t_{\max}}$	$\frac{1}{5000}$	$\frac{1}{5000}$	$\frac{1}{10000}$	$\frac{1}{10000}$
	$\frac{\Delta t}{\Delta t_{\max}}$ s.t. Test error $< 20\%$	$[5.10^{-5}, 3.10^{-3}]$	$[5.10^{-5}, 7.10^{-2}]$	$[10^{-5}, 4.10^{-3}]$	$[10^{-5}, 2.10^{-2}]$
Cycle-to-cycle variability	Test error	$16.8 \pm 0.7\%$	$18.8 \pm 0.3\%$	$10.2 \pm 0.2\%$	$11.2 \pm 0.1\%$
Device-to-device variability	Test error	$13.9 \pm 0.6\%$	$14.9 \pm 0.4\%$	$9.6 \pm 0.3\%$	$10.8 \pm 0.3\%$

Table 2.3: Summary of the results obtained on the Discriminative RBM. Cycle-to-cycle variability is taken to be $\frac{\sigma_{\text{intra}}}{(G_{max}-G_{min})} = 6 \cdot 10^{-3}$ with $\beta = 0.005$. Device-to-device variability is taken to be $\left(\frac{\sigma}{\mu}\right)_{\text{inter}} = 1$ with $\beta = 0.005$. Each simulation was ran over 30 epochs with a mini-batch size of 100, 5 times per value of pulse width, error bars indicate median, first quartile and third quartile.

Discussion

To design hardware-friendly learning rules that are both local and resilient to imprecise programming of memristive devices, we first studied the three most encountered RBM-based neural networks in the neuromorphic literature in terms of their performance on the MNIST discrimination task, when trained under our baseline memristor-based gradient descent algorithm (Cst). With typical values of non-linearity, cycle-to-cycle and device-to-device variabilities, the Discriminative RBM outperforms the two other architectures. Using one bit of information at each learning step (i.e. the sign of the gradient) with one bit per neuron (i.e. stochastically sampled binary neurons) while achieving a classification performance akin to software-based simulations, the Discriminative RBM trained under Contrastive Divergence appears to be a good candidate for in situ learning. Also, the choice of the pulse width is critical with respect to the device imperfections. While hand-tuning the programming width as a hyperparameter selects an optimal value that is never predictable in advance, we can understand how the weight update physics influence it. Increasing non-linearity or device-to-device variability, with regards to an ideal device, favors pulse widths that are shorter to avoid abrupt conductance changes. Conversely increasing cycle-to-cycle variability selects pulse widths that are longer to overshadow the programming noise with respect to the amplitude of the conductance update.

More importantly, and surprisingly at first sight, the Deep Belief Net does not perform better than the Discriminative RBM. On the one hand, the inefficiency of depth in our specific training and inference setting is due to the coarsened feature extraction abilities of RBMs upon using memristive devices. In the best case (near-linear) the stack of RBMs is not useful, in the worst case (non-linear), learning is dramatically jeopardized when passing corrupted features into downstream RBMs. On the other hand, this inefficiency also stems from not fine-tuning the stack of RBMs with backpropagation, as per our choice to solely focus on local learning rules.

For advanced applications, Discriminative RBMs could nonetheless be used within deep neural networks to learn complex tasks, if the transfer learning approach is used. This approach consists in importing upstreams weights previously trained on software for feature extraction on a particular kind of data, and training in situ only the last layers on similar but

more specific data [34]. This strategy has been proven in various contexts, and allows training neural networks on new tasks with relatively modest amounts of data, if the neural network has been previously trained on a different but yet similar task with important amounts of data [146]. Therefore, our simplified technique for training should not be considered for training deep neural networks in their entirety, but for adapting them to new situations, for example in embedded environments.

In the second section of this part, we showed that decreasing the variance of Contrastive Divergence by summing it across samples (i.e. mini-batches) and stochastic realizations of neurons (i.e. parallel Gibbs chains) considerably improved the performance of Discriminative RBMs, when trained with realistic memristive devices. This trend was seen in terms of non-linearity, cycle-to-cycle and device-to-device variability. Decreasing the variance of Contrastive Divergence indeed makes the algorithm more immune to non-linearity: it is no longer penalized by abrupt conductance changes along wrong directions. And as this technique selects a smaller programming pulse width, it can smoothen out the discrepancies due to variability sources. Interestingly, our findings on the Discriminative RBM shed new light on the impact of the device imperfections by entangling them all around the choice of the programming pulse width, which has to be tediously tuned when it is fixed throughout learning (Cst). This statement pushed us to investigate the use of RProp driven pulse widths. By taking into account the sign of the gradient between two consecutive learning steps, this technique enables to enlarge the range of sensible pulse widths by up to two decades without affecting the resilience to the device imperfections. From Fig. 2.6, Fig. 2.7 and Table 2.3, we acknowledge that the use RProp may not always yield the best error rate when tuned at its optimal pulse width. However, we see from Fig. 2.5 that RProp outperforms Cst when taking the whole range of pulse widths into account. Also, while we explicitly studied the combination of RProp with the use of multiple Gibbs chains with regards to variability sources, we want to stress here its impact upon non-linearity effects as it appears in Table 2.3: (20 CD, Cst) and (20 CD, RProp) at $\beta = 3$ achieve the same optimal performance and the only effect of RProp is to enlarge the range of pulse widths achieving a test error that is lower than 20%. Thus this technique is of definite practical interest as it reduces the need to tune hyperparameters, a major concern for learning in embedded contexts.

Our choices regarding device modeling were guided by the existing literature. For instance, putting device-to-device variability into the multiplicative parameter C appearing in Eq. (1.5) is inspired by device measurements [129]. Our work would not apply directly to Phase Change Memory devices which exhibit a strong asymmetry between potentiation and depression. However, our results would be applicable directly to pairs of Phase Change Memories associated in 2-PCM structures [104]. We also bear in mind that the use of mini-batches calls for the design of elaborate memory devices. A promising path to accomplish on-chip mini-batch gradient descent could be inspired by recent works [34], combining a volatile and a non-volatile memory that would be updated within and across mini-batches respectively.

Overall, these results suggest the possibility to achieve on-chip learning with memristive learning with Discriminative Restricted Boltzmann machines, using a local learning rule and very simple device programming, and highlights strategies to make the learning process viable even with highly imperfect devices. More generally, our results highlight that the methods for making learning functional with imperfect analog hardware can differ from the techniques used for standard machine learning in software, suggesting the high need for hardware and learning algorithm codevelopment.

Part III

Introduction to Equilibrium Propagation

Summary

In this part, we introduce *Equilibrium Propagation*, one of the central algorithms of this PhD thesis. We first formally introduce the algorithm as formalized by Scellier [97]. Then, we connect Equilibrium Propagation with Boltzmann Machines and we present intuitions about the algorithm. Afterwards, we present the generalization of Equilibrium Propagation to vector field dynamics [147], where the requirement of an energy function for the system dynamics is lifted. Finally, we mention the equivalence established between Equilibrium Propagation and Recurrent Backpropagation [148], a result which inspired one of the main contributions of this thesis presented in part IV.

Chapter 1

Equilibrium Propagation

1.1 A heuristic view

Before delving into the theory of Equilibrium Propagation, it is worth conveying a heuristic view of this algorithm. Let us consider a neural network with a visible layer x, one hidden layer s^1 and an output layer $\hat{y}$. We assume that this neural network is recurrent: simply speaking, the neural network evolves through time as a dynamical system — we introduced recurrent neural networks in Eq. (1.16) of subsection 1.2.3 of part I. We furthermore assume that the dynamics of the neural network derive from an energy potential: the neural network dynamics subsequently evolve towards minima of the energy function. More explicitely, if we denote $s = (s, \hat{y})^\top$ the global variable that labels the state of the hidden and output neurons altogether, θ the set of all the synapse weight values of the system and E the energy function of the system, the dynamics are defined as:

$$\frac{ds}{dt} = -\frac{\partial E}{\partial s}(x, s; \theta). \tag{1.1}$$

It should be noted in Eq. (1.1) that the energy function E explicitly depends on the input x, the neurons s and the weight values of the synapses θ. By definition, the system reaches equilibrium when $\frac{ds}{dt} = 0$ and we denote s_* the steady state achieving this equilibrium — therefore $\hat{y}_*$ and s^1_* denote the steady states of the output and hidden layers respectively.

We are in a supervised training context where the goal of learning is to map an input x to a target y. More precisely here, the goal of learning is to adjust the weights θ so that, upon clamping an input x to the visible layer and letting the system evolve along the dynamics given by Eq. (1.1) until reaching equilibrium, the resulting steady state configuration of the output layer $\hat{y}_*$ is the closest to the ground-truth target y — see Fig. 1.1 for a cartoon.

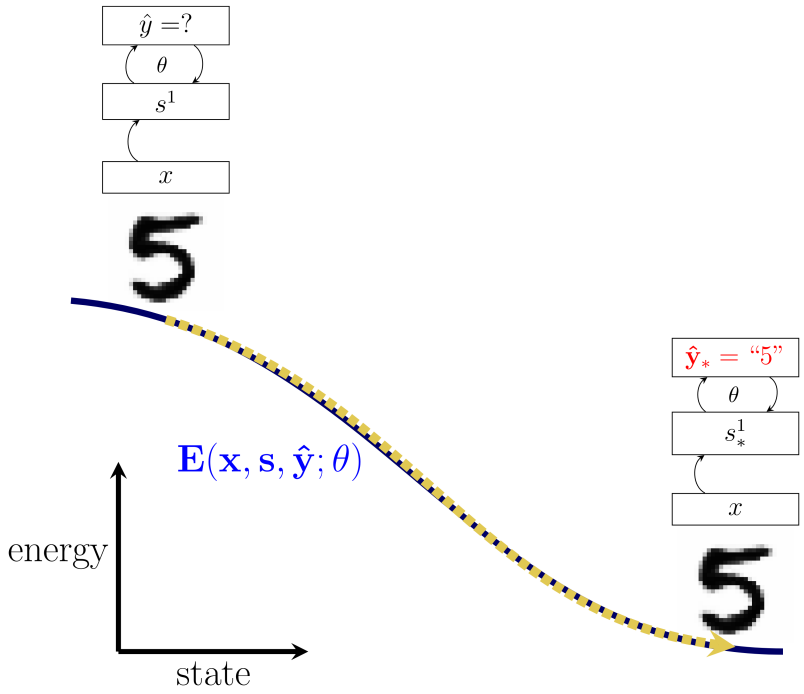

Figure 1.1: **A heuristic view of Equilibrium Propagation** [97]. We consider a recurrent neural network with a visible layer x, one hidden layer s^1 and an output layer $\hat{y}$ whose dynamics are governed by Eq. (1.1) with a given energy function E: the network evolves towards minima of energy. The energy landscape seen by the neurons is plotted in dark blue. In the Equilibrium Propagation setting, the goal of learning is to adjust the weight values of the synapses θ so that upon presenting an input x to the visible layer of the system (e.g. a MNIST sample, as depicted here), the system evolves towards a minimum of the energy function so that the resulting steady state of the output layer $(\hat{y}_*)$ is the closest to the ground-truth target y. The trajectory of the systemv from its initial state until reaching equilibrium is plotted with a dashed yellow arrow.

Since this system falls into the category of recurrent neural networks, there is at this stage a natural intuition that upon defining the appropriate loss function, backpropagation through time could be used for the purpose of this learning objective, as we showed in subsection 1.2.3 of part I. However, instead of backpropagating error gradients through the computational graph, Equilibrium Propagation enables the system *itself* to compute the error gradients *through time*. In the next sections, we explain why and how.

1.2 Theory

We now formally introduce Equilibrium Propagation as it is presented in [97]. Again, we consider a system that is described by a state variable s, parameters θ and associated energy

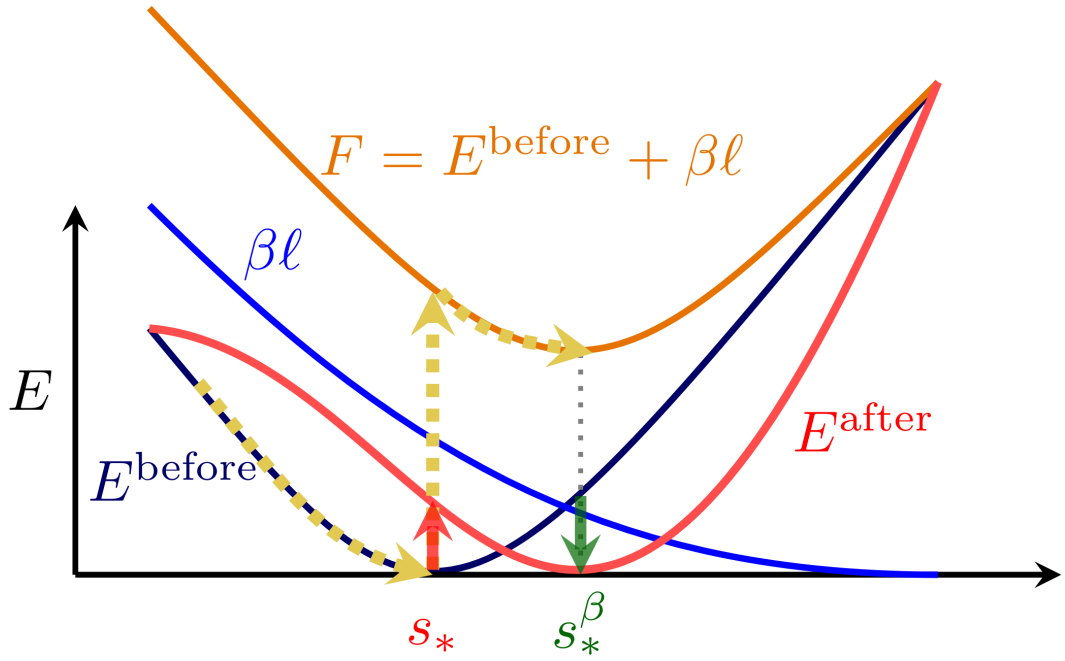

Figure 1.2: **Equilibrium Propagation** [97]. The system trajectory is depicted with yellow dotted arrows. During the first phase, the system descends the energy landscape before the weight update (E^{before}, in dark blue), until reaching the first steady state s_* (in red). During the second phase, the elastic contribution $\beta\ell$ (bright blue) is added to E^{before}, so that the system subsequently evolves along the free energy landscape F (orange), until reaching the second steady state s_*^β (in green). The effect of the learning rule Eq. (1.5) is to increase the energy of s_* (red upward arrow) and to decrease the energy of s_*^β (green downward arrow), resulting in a new energy landscape (E^{after} in red).

function $E(x, s; \theta)$ where x is an input clamped to the visible layer, which models "external world". Generally, E is a Hopfield energy, as we shall see on the concrete example presented later in this chapter. We also define a cost function ℓ that tells how "good" a network configuration is with respect to a target y, and the loss function $\mathcal{L}$ is the cost function evaluated at equilibrium: $\mathcal{L} = \ell(s_*, y)$ with $\frac{\partial E}{\partial s}(s_*) = 0$, where s_* is called the *free steady state*. The goal of learning is to adjust the parameters θ so that the cost function at equilibrium is minimal, or more formally:

$$\begin{cases} \min_\theta \ell(s_*(\theta), y) \\ \text{subject to} \quad \frac{\partial E}{\partial s}(s_*) = 0 \end{cases}. \tag{1.2}$$

When learning is achieved, the output of the system should spontaneously relax towards y upon presenting x, namely: $\hat{y}_* \sim y$. Before stating their main result, Scellier and Bengio also introduce a *free energy* function $\mathcal{F}$ defined as:

$$\mathcal{F}(x, s, y; \theta, \beta) = E(x, s; \theta) + \beta \ell(s, y), \tag{1.3}$$

where β is called the *nudging strength*. Physically, the free energy is the landscape seen by the neurons when they are nudged towards y with strength β: the term $\beta\ell(s, y)$ in Eq. (1.3) can be seen a elastic energy potential of stiffness β that can shift the steady state of a system. For instance, if we were to think of a pendulum under the influence of gravity, E in Eq. (1.3) would model gravity energy potential and $\beta\ell$ the elastic energy potential of a spring that would pull the pendulum updwards.

We call the *nudged steady state* s_*^β the configuration of the network which minimizes $\mathcal{F}$: $\frac{\partial \mathcal{F}}{\partial s}(s_*^\beta) = 0$. Scellier and Bengio showed that, as expected, s_*^β has a lower cost than s_* so that, in this regard, s_*^β a "better" state than s_*.

The main result of [97] is that the gradient of the loss function is given by:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \lim_{\beta \to 0} \frac{1}{\beta} \left(\frac{\partial \mathcal{F}}{\partial \theta}(x, s, y; \theta, \beta) - \frac{\partial \mathcal{F}}{\partial \theta}(x, s, y; \theta, \beta = 0) \right). \tag{1.4}$$

In particular, when the cost function ℓ does not depend on θ, as it is the case when we use a mean-squared error (i.e. $\ell(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$), the learning rule reads:

$$\boxed{\frac{\partial \mathcal{L}}{\partial \theta} = \lim_{\beta \to 0} \frac{1}{\beta} \left(\frac{\partial E}{\partial \theta}(x, s, y; \theta, \beta) - \frac{\partial E}{\partial \theta}(x, s, y; \theta, \beta = 0) \right)} \tag{1.5}$$

The interpretation of Eq. (1.5) is very intuitive. When going down the loss function $\theta \leftarrow \theta - \alpha \frac{\partial \mathcal{L}}{\partial \theta}$, the first term of Eq. (1.5) contributes to *decreasing* the energy of s_*^β which is a "good" state while the second term contributes to *increasing* the energy of s_* which is a "bad" state. The energy landscape seen by the neurons is iteratively deformed until minima of the energy correspond to data configurations. This interpretation exactly follows the rationale of Contrastive Hebbian Learning as illustrated on Fig. 4.1. Finally, note that Eq. (1.5) holds for any network topology and does not specify any particular dynamics, so long as the system under consideration can minimize an energy function E and the associated free energy function $\mathcal{F}$.

1.3 Algorithm

Let us now describe the algorithmic implementation of Equilibrium Propagation. To perform the weight update prescribed by Eq. (1.5), we need to get the free and nudged steady states s_*

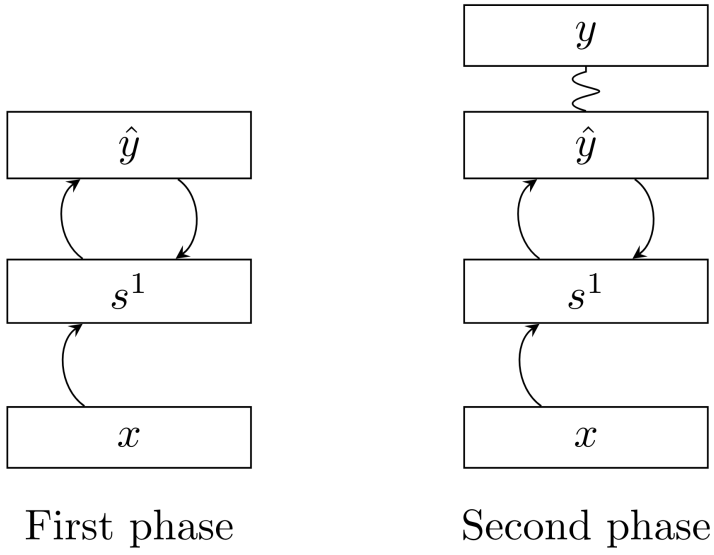

Figure 1.3: **Implementation of Equilibrium Propagation** [97]. We consider a fully connected architecture with a visible layer taking input x, hidden layer s and output layer $\hat{y}$. During the first phase, the system evolves on its own, under the influence of x, until reaching the first steady state s_*. During the second phase, the output layer $\hat{y}$ is elastically nudged towards the ground-truth target y, until reaching the second steady state s_*^β. The learning rule Eq. (1.5) is subsequently applied.

and s_*^β respectively for a given input x. For this purpose, Equilibrium Propagation proceeds in two phases. During the first phase, we execute the following dynamics:

$$\frac{ds}{dt} = -\frac{\partial E}{\partial s}(x, s; \theta), \tag{1.6}$$

until reaching the free steady state s_* since by definition, Eq. (1.6) minimizes E. Then, during a second phase, we execute the following dynamics:

$$\frac{ds}{dt} = -\frac{\partial E}{\partial s}(x, s; \theta) - \beta\frac{\partial \ell}{\partial s}, \tag{1.7}$$

until reaching s_*^β, since Eq. (1.7) minimizes $\mathcal{F}$. Then, along with s_*^β and s_*, we simply perform the weight update given by Eq. (1.5). The whole algorithm is summarized by Alg. 3.

Algorithm 3 Equilibrium Propagation [97]

Input: x (input data), θ (parameters), y (target)
Output: θ

while s is not converged **do**
    $s \leftarrow s - dt\frac{\partial E}{\partial s}$
end while
$s_* \leftarrow s$
while s is not converged **do**
    $s \leftarrow s - dt\frac{\partial E}{\partial s} - dt\beta\frac{\partial \ell}{\partial s}(s, y)$
end while
$s_*^\beta \leftarrow s$
$\theta \leftarrow \theta - \frac{1}{\beta}\left(\frac{\partial E}{\partial \theta}(s_*^\beta) - \frac{\partial E}{\partial \theta}(s_*)\right)$

1.4 Neural network model trained by Equilibrium Propagation

Although Eq. (1.5) holds for any neural network so long as the system under consideration can minimize an energy function E, the neural networks trained by Equilibrium Propagation in practice are recurrent neural networks typically described by dynamics of the kind of Eq. (1.6). More precisely, these RNNs belong to the class of *convergent* RNNs which, by definition, converge to a steady state given a static input. Depending on the literature, these networks are equivalently called "continuous Hopfield networks" and their dynamics are sometimes abusively said to be Leaky-Integrate and Fire (LIF), although we only consider here rate-based and not spiking models.

1.5 Example

Let us now apply this theory to a two layer neural network, defined by the following energy:

$$E(x, s^1, \hat{y}; w_1, w_2) = \frac{1}{2}\|s^1\|^2 + \frac{1}{2}\|\hat{y}\|^2 - \sigma(s^{1^\top}) \cdot w_1 \cdot \sigma(x) - \sigma(\hat{y}^\top) \cdot w_2 \cdot \sigma(s^1), \tag{1.8}$$

where σ denotes the activation function. Again, the goal of learning is that the free steady state of the output layer $\hat{y}_*$ coincides the best with a given target y. Typically, we choose the cost function as a mean squared error function:

$$\ell(\hat{y}, y) = \frac{1}{2}\|\hat{y} - y\|^2 \tag{1.9}$$

Applying Eq. (1.6) with this energy gives the following differential equations for the first phase of Equilibrium Propagation:

$$\begin{cases} \frac{d\hat{y}}{dt} = -\hat{y} + \sigma'(\hat{y}) \odot w_2 \cdot \sigma(s^1) \\ \frac{ds^1}{dt} = -s^1 + \sigma'(s^1) \odot \left(w_1 \cdot \sigma(x) + w_2^\top \cdot \sigma(\hat{y}) \right) \end{cases} . \tag{1.10}$$

During the second phase, the system satisfies the following equations:

$$\begin{cases} \frac{d\hat{y}}{dt} = -\hat{y} + \sigma'(\hat{y}) \odot w_2 \cdot \sigma(s^1) + \beta(y - \hat{y}) \\ \frac{ds^1}{dt} = -s^1 + \sigma'(s^1) \odot \left(w_1 \cdot \sigma(x) + w_2^\top \cdot \sigma(\hat{y}) \right) \end{cases} , \tag{1.11}$$

where the output layer is elastically nudged towards y and $\odot$ denotes Hadamard (element-wise) product. Having the free and nudged steady states of all the neurons, the weight update given by Eq. (1.5) reads in this particular case:

$$\begin{cases} \Delta w_2 = \frac{1}{\beta} \left(\sigma(\hat{y}_*^\beta) \cdot \sigma(s_*^{1,\beta})^\top - \sigma(\hat{y}_*) \cdot \sigma(s_*^1)^\top \right) \\ \Delta w_1 = \frac{1}{\beta} \left(\sigma(s_*^{1,\beta}) \cdot \sigma(x)^\top - \sigma(s_*^1) \cdot \sigma(x)^\top \right) \end{cases} \tag{1.12}$$

1.6 Intuitions about Equilibrium Propagation

1.6.1 Going deeper with Boltzmann Machines?

Variational inference in Deep Boltzmann Machines. In the work presented previously, we stack RBMs and perform greedy learning to train deeper architectures with a local learning rule, which is theoretically justified by the fact that this procedure increases a lower bound on the log-likehood $p(x; \theta)$ [135]. Increasing further this lower bound implies taking hidden layer interactions into account. Again, the hardest part in Boltzmann Machine learning is to sample the posterior distribution of the model. As highlighted in part I, inference is tractable in a *Restricted* Boltzmann Machines because the *restriction* of the synaptic connections make the model distribution tractable and easy to sample from. For Deep Boltzmann Machines [149], variational inference is used to approximate the model posterior distribution. The central idea of variational inference is to approximate the true posterior distribution by an *approximate* posterior distribution that maximizes the log-likelihood of the model *.The derivation presented in this subsection goes along the lines of [150].

*Note that this setting where both the parameters and posterior are unknown and one is needed to compute the other falls into *Expectation-Maximization*. During the expectation phase, the posterior is approximated given parameter values, and during the maximization phase the parameter gradients are computed given posterior values.

For the sake of concreteness, let us consider a Deep Boltzmann Machine with two hidden layers defined by the energy:

$$E(x, s^1, s^2; w_1, w_2) = -{s^1}^\top \cdot w_1 \cdot x - {s^2}^\top \cdot w_2 \cdot s^1, \tag{1.13}$$

with $p(x, s^1, s^2; \theta) = \exp(-E(x, s^1, s^2; \theta))/\mathcal{Z}$, $\theta = \{w_1, w_2\}$ and $\mathcal{Z}$ the partition function. More precisely, variational inference aims to approximate $p(s^1, s^2|x; \theta)$ with an approximate family of distributions $q(s|x) = q(s^1, s^2|x)$ which are chosen so that they maximize log-likelihood of the model. For any distribution q, it can be shown with the Jenssen inequality that $p(x; \theta)$ can be lower-bounded by $\mathcal{L}_b(x, q)$ as:

$$\log p(x; \theta) \geq \mathcal{L}_b(x, q; \theta) \tag{1.14}$$

where $\mathcal{L}_b(x, q; \theta) = p(x; \theta) - KL(q(s|x)||p(s|x; \theta))$, with KL denotes the Kullback-Leibler divergence of two distributions: $KL(q||p) = \sum_h q(h) \log \frac{q(h)}{p(h)}$. Therefore, the lower-bound is tight when $KL(q(s|x)||p(s|x; \theta)) = 0$, that is $q(s|x) = p(s|x; \theta)$.

$\mathcal{L}_b(x, q; \theta)$ can be computed explicitly by rewriting it as $\mathcal{L}_b(x, q; \theta) = \mathcal{H}(q) - \langle \log p(x, s; \theta) \rangle_q$ where $\mathcal{H}(q) = \sum_h q(h|v) \log q(h|v)$ denotes the entropy of q. For the Deep Boltzmann Machine defined per Eq. (1.13, $\mathcal{L}_b(x, q; \theta)$ therefore rewrites:

$$\mathcal{L}_b(x, q; \theta) = \mathcal{H}(q) + \langle {s^1}^\top \cdot w_1 \cdot x \rangle_q + \langle {s^2}^\top \cdot w_2 \cdot s^1 \rangle_q - \log \mathcal{Z}, \tag{1.15}$$

where $\log \mathcal{Z}$ does not depend upon q. To go further in the determination of approximate posteriors, we need an explicit prior on their form. *Mean-field inference* is one particular case of variational inference where a simple prior is assumed.

Mean-field inference. Mean-field inference is a particular case of variational inference where we choose factorial distributions to approximate q:

$$q(s^1, s^2|x) = \prod_i q^1(s_i^1|x) \prod_i q^2(s_i^2|x), \tag{1.16}$$

where the probability of each hidden unit is a Bernoulli distribution, i.e. $q(s_i^1 = 1|v) = \lambda_i^1$, $q(s_i^2 = 1|v) = \lambda_i^2$. In this case we can easily show that Eq. (1.15) rewrites:

$$\begin{aligned}\mathcal{L}_b(x, q; \theta) = {}& \lambda^{1,\top} \cdot \log \lambda^1 + (1 - \lambda^1)^\top \cdot \log(1 - \lambda^1) + \lambda^{2,\top} \cdot \log \lambda^2 + (1 - \lambda^2)^\top \cdot \log(1 - \lambda^2) \\ & + {\lambda^1}^\top \cdot w_1 \cdot x + {\lambda^2}^\top \cdot w_2 \cdot \lambda^1 + \log \mathcal{Z}.\end{aligned} \tag{1.17}$$

Mean-field inference finds λ_i values that maximize $\mathcal{L}_b$ by cancelling its gradient component-wise: $\frac{\partial \mathcal{L}(x,q;\theta)}{\partial \lambda_i} = 0$. We iteratively run the resulting equation for each layer i until we satisfy a convergence criterion. We have:

$$\begin{aligned} &\left\{ \begin{array}{l} \frac{\partial \mathcal{L}}{\partial \lambda^1} = 0 \\ \frac{\partial \mathcal{L}}{\partial \lambda^2} = 0 \end{array} \right. , \\ &\Leftrightarrow \left\{ \begin{array}{l} \lambda^1 = \sigma(w_1 \cdot x + w_2^\top \cdot \lambda^2) \\ \lambda^2 = \sigma(w_2 \cdot \lambda^1) \end{array} \right. . \end{aligned} \tag{1.18}$$

In Deep Boltzmann Machine training, mean-field inference enables getting the data statistics of Contrastive Divergence, while the model statistics are again obtained by Gibbs sampling, as in RBMs [149]. This whole procedure therefore allows to estimate gradients of the log-likelihood of Deep Boltzmann Machines, and therefore training.

1.6.2 Neural computation: going down the energy

Assuming the neuron are described, at any moment in *time*, by their approximate posterior distribution λ_t, mean-field inference prescribes that their dynamics should read:

$$\left\{ \begin{array}{l} \lambda^1_{t+1} = \sigma(w_1 \cdot x + w_2^\top \cdot \lambda^2_t) \\ \lambda^2_{t+1} = \sigma(w_2 \cdot \lambda^1_t), \end{array} \right. \tag{1.19}$$

until reaching a steady state. By construction of mean-field inference, Eq. (1.19) make the system evolve towards states of higher probability. We insist on Eq. (1.19) because very similar dynamical equations will appear later in this thesis, and according to the same principle.

Taking a step back on the particular Boltzmann Machine setting, a central hypothesis in neural computation is that the neurons evolve towards configurations of higher probability under the current "model of the world" associated with the parameters of the model, i.e. the value of the synapses at time scales such that they can be considered to be static [151]. Writing the configuration of the neurons as s, this hypothesis can be written heuristically as:

$$\frac{ds}{dt} \approx \frac{\partial \log p(x, s; \theta)}{\partial s}. \tag{1.20}$$

Assuming an energy model like a Boltzmann Machine: $p(x, s; \theta) \propto \exp(-E(x, s; \theta))$, Eq. (1.20) rewrites:

$$\frac{ds}{dt} \approx -\frac{\partial E(x, s; \theta)}{\partial s}. \tag{1.21}$$

Therefore, evolving towards more probable configurations under the posterior $p(s|x;\theta)$ and eventually sampling from $p(s|x;\theta)$ amounts for the neurons to descend the energy function E *. Equilibrium Propagation builds on this central assumption.

1.6.3 Key ingredients of Equilibrium Propagation

Computing through time, encoding the error as a force and bidirectional integration. From either a neuromorphic or neuroscience prospective, one fundamental limitation of backpropagation is that it prescribes a learning rule which is not local in space (see subsection 3.3.1). Another obstacle of backpropagation is that it proceeds *backward*: as emphasized in the introduction, it propagates error backward through the computational graph that is used for inference (see Fig. 1.3 in part I). However, it is very likely that the brain computes error signals out of its own biophysics in a *forward-time* fashion. From a neuromorphic prospective, a "dream chip" would use the same circuitry, more precisely in our context of study the same crossbar for inference and gradient computation, out of the sole circuit physics. The idea of performing both inference *and* gradient computation on the same circuitry motivates the following ingredients for Equilibrium Propagation:

- The first error signal in the output layer should be encoded *physically*. Pretty much like dopamine acts a reward signal upon neural dynamics in the brain, the original error signal $\frac{\partial \ell}{\partial \hat{y}}$ should act as a force on the output layer.
- Somehow, by standard backpropagation or any other learning scheme, the error signal should be routed back into the network. With our constraints, we want error signals to propagate *out of the network dynamics*. Therefore, we consider *recurrent neural network* where computation happens *through time*.
- Another consequence is that each neuron should be able integrate both *bottom-up* (data) information and *top-down* (error) information. At the theoretical level, this involves the existence of *bidirectional* connections.

In this way, error signals can propagate across time and layers, so that at some point in time, the learning rule can also become local, which was one of the most fundamental point to address.

*This statement is rigorous if we add additive noise to Eq. (1.20) and Eq. (1.21), which is similar to Stochastic Gradient Langevin Dynamics in the context of Bayesian learning [152]. This detail is not essential here and is omitted for readability.

The requirement of equilibrium from a topological prospective. Still, the last three ingredients are not sufficient to complete Equilibrium Propagation credit assignment scheme. Error signals propagate through time from "above" and the data is sent from "below". With a biological terminology, we call the input connections to the neuron integrating bottom-up signals *basal* dendrites and those integrating top-down signals *apical* dendrites. Whenever a neural network model allows for bidirectional integration, it implicitly assume the existence of such dendritic compartments — see Fig. 1.4. However, it is misleading to think of the top-down input arriving to each neuron as a pure error signal: without further assumption about the state of the system or the topology of the neuronal circuit employed, top-down signals are a mix of error *and* data information and there is no way that each neuron could disentangle these two components.

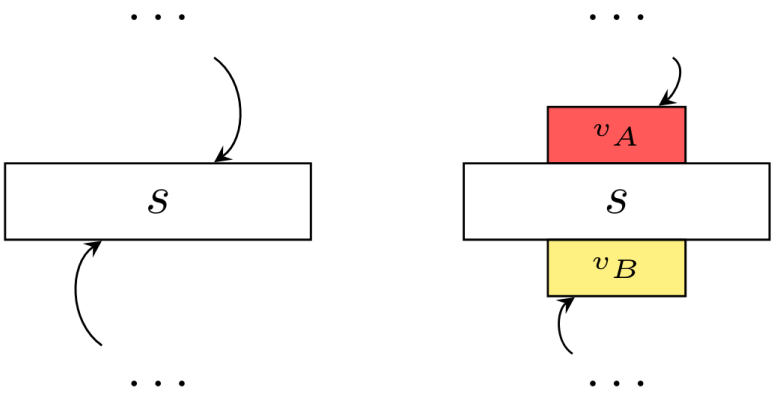

Figure 1.4: **Dendritic compartments**. Equilibrium Propagation applies to neural networks who can integrate signals bidirectionally: roughly speaking, from below and from above. When a layer receives two inputs (top-down and bottom-up inputs), it implicitly assumes dendritic compartments: basal compartments (of voltage v_B) and apical compartments (of voltage v_A) integrate bottom-up and top-down signals respectively. We invoke this biological terminology for completeness, although Equilibrium Propagation formalism and application do not require these concepts.

The way Equilibrium Propagation proceeds in this purpose is contained in its name: the requirement of *equilibrium*. Let us consider again the layered architecture depicted on Fig. 1.3, and assume that we are in the second phase of Equilibrium Propagation (leftmost part of Fig. 1.5). At this stage, the system is under the influence of the input and of the nudging of the output layer. More precisely, the layer s^1 integrates both "self-generated" top-down inputs (blue arrows) and pure error signals (red arrows). During the second phase, s^1 only integrates self-generated top-down inputs. "Substracting" these two situations, we are only left out with the red arrows: only the error contribution so that the layer s^1 only integrates the error signal coming from the output layer - see Fig. 1.5. This "substraction" corresponds to a temporal variation of the system, which goes to show that temporal variations of the system can encode error signals.

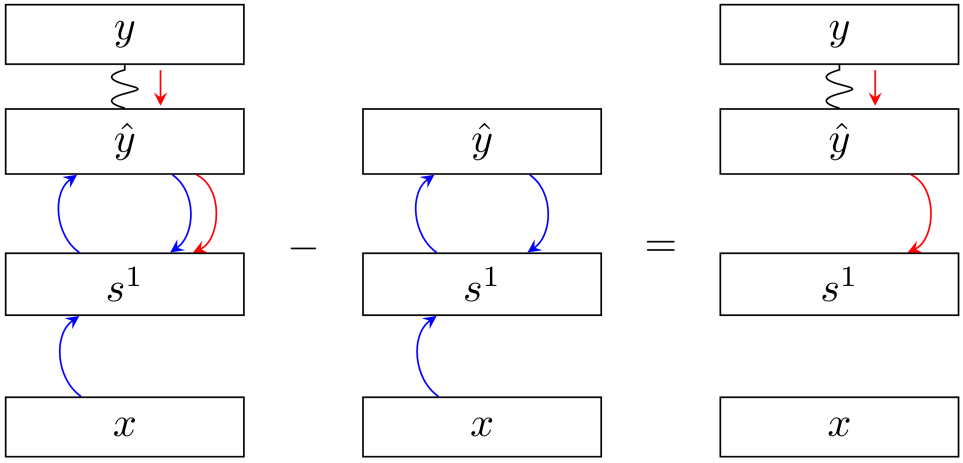

Figure 1.5: **Equilibrium Propagation from a topological prospective**. During the second phase, the layer s^1 integrates both self-generated top-down inputs (blue arrows) and the error signal coming from the output layer (red arrows). Conversely during the second phase, s^1 only integrates the self-generated top-down inputs. Substracting the two phases, we are only left out with the red arrows. This simple reasoning conveys that the temporal variations of the system during the second phase of Equilibrium Propagation may encode error signals.

The requirement of equilibrium from a dynamical prospective. We can convey more explicitly why the requirement of equilibrium is essential. Let us assume the network state is described by a variable s. For a layered architecture of the kind described before in this thesis, s represents the concatenation of all the layers: $s = (s^1, s^2, \cdots, s^N = \hat{y})^\top$. Equilibrium Propagation assumes that s evolves as:

$$\frac{ds}{dt} = -\frac{\partial E}{\partial s}(x, s; \theta), \tag{1.22}$$

where x denotes an input clamped to the visible layer and E the energy function of the system. Since Eq. (1.22) derives from a potential, after running the dynamics for a sufficiently long time, the system reaches a steady state s_* such that $\dot{s} = 0$, that is $\frac{\partial E}{\partial s}(s_*) = 0$. Once equilibrium is achieved, assume a nudging $-\frac{\partial \ell}{\partial \hat{y}}$ is applied on the output layer: output neurons are pulled towards directions of decreasing cost, that is towards y. Since the system is initially at rest, subsequent motion of the system is solely due to this error signal: in Equilibrium Propagation, **temporal variations of the system encodes error signals**. Therefore, the initial perturbation of the equilibrium undergone by the output layer propagates across layers, hence the name "Equilibrium Propagation". More explicitly, Fischer and Bengio [151] propose the following heuristic. If the output layer is nudged by $-\frac{\partial \ell}{\partial s}$, its resulting temporal variation reads:

$$\dot{\hat{y}} = -\frac{\partial \ell}{\partial \hat{y}}. \tag{1.23}$$

The resulting variation in the previous layer s^{N-1} is:

$$\begin{aligned}
\dot{s}^{N-1} &= \frac{\partial s^{N-1}}{\partial \hat{y}} \cdot \dot{\hat{y}} \\
&\propto \frac{\partial^2 E}{\partial s^{N-1} \partial \hat{y}} \cdot \dot{\hat{y}} \\
&= \left(\frac{\partial^2 E}{\partial \hat{y} \partial s^{N-1}} \right)^\top \cdot \dot{\hat{y}} \\
&\propto -\left(\frac{\partial \hat{y}}{\partial s^{N-1}} \right)^\top \cdot \frac{\partial \ell}{\partial \hat{y}} \\
&\propto -\frac{\partial \ell}{\partial s^{N-1}},
\end{aligned} \tag{1.24}$$

where we have used that $\dot{\hat{y}} = -\partial E / \partial \hat{y}$ and $\dot{s}^{N-1} = -\partial E / \partial s^{N-1}$. This reasoning extends to previous layers: $\dot{s}^n \sim -\frac{\partial \ell}{\partial s^n}$ - see Fig. 1.6.

Importantly, note that the requirement of an energy function for the network dynamics has two distinct roles. First, it ensures the existence of an equilibrium, as we pointed it before. Second and in a more subtle way, it also ensures that the Jacobian of the dynamics is symmetric: $\frac{\partial s^{N-1}}{\partial \hat{y}} = \left(\frac{\partial \hat{y}}{\partial s^{N-1}} \right)^\top$. This is a condition for the temporal variations of the system to carry error gradients, or more precisely for the equivalence of Equilibrium Propagation with Recurrent Backpropagation and Backpropagation Through Time, as we shall see later in this thesis.

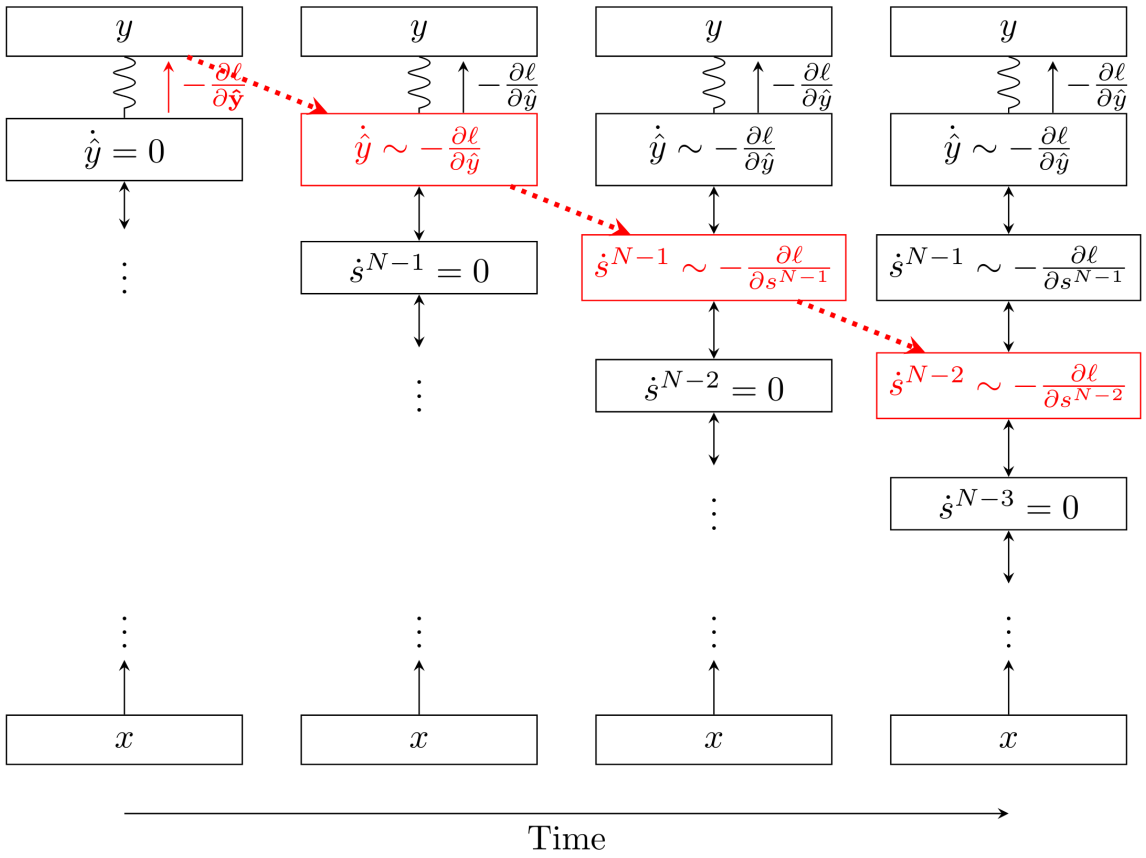

Figure 1.6: **Intuition of Equilibrium Propagation**. Equilibrium Propagation applies to recurrent neural networks so that neural computation occurs throughout time. Assuming the neural network is initially at equilibrium ($\dot{s}^n = 0$), the output layer $\hat{y}$ undergoes the perturbation $\frac{\partial \ell}{\partial \hat{y}}$. Through reciprocal connections, this perturbation propagates across time (from left to right) and layers (from top to bottom), with $\dot{s}^n \sim \frac{\partial \ell}{\partial s^n}$.

Chapter 2

Why is Equilibrium Propagation hardware-friendly?

2.1 Link between Equilibrium Propagation and Spike Timing Dependent Plasticity

Pre-synaptic activity times the variation of the post-synaptic activity. Following their preliminary intuition of Equilibrium Propagation, Bengio and Fischer [151] asked the following question: given that the temporal variations of the membrane potential of the neurons can encode error gradients after a perturbation from equilibrium as we shown earlier, what would it take to carry out gradient descent on synapses? They simply noted that, denoting again $\mathcal{L}$ the loss function, using Eq. (1.24) and assuming $\frac{\partial s^{n+1}}{\partial w_n} = \sigma(s^n)$:

$$\begin{aligned}\Delta w_n &= -\frac{\partial \mathcal{L}}{\partial w_n} \\ &= -\left(\frac{\partial s^{n+1}}{\partial w_n}\right)^\top \cdot \frac{\partial \mathcal{L}}{\partial s^{n+1}} \\ &\sim \sigma(s^n) \cdot \dot{s}^{n+1}, \end{aligned} \tag{2.1}$$

which gives, element-wise:

$$\Delta w_{n,ij} \sim \sigma(s_j^n)\dot{s}_i^{n+1} \tag{2.2}$$

Putting Eq. (2.2) into words, the synaptic update should be proportional to the pre-synaptic activity times the change of post-synaptic activity.

Relation to Spike Timing Dependent Plasticity. Very interestingly, the learning rule prescribed by Eq. (2.2) was shown to resemble STDP [153]. To convey this, let us assume that each neuron s^n spike at a rate proportional to $\sigma(s^n)$: $\xi_j^n \sim \sigma(s_j^n) \in \{0, 1\}$, where $\xi_j^n = 1$ means that the neuron s_j^n has spiked. We also apply Eq. (2.2) in an event-based fashion, whenever the presynaptic neuron s^n spikes:

$$\Delta w_{n,ij} \sim \xi_j^n \dot{s}_i^{n+1} \tag{2.3}$$

Let us moreover assume that when the presynaptic neuron spikes ($\xi_j^n = 1$) at t_j^n, the post-synaptic activity is rising ($\dot{s}_i^{n+1} > 0$), so that Eq. (2.3) gives $\Delta w_{n,ij} > 0$. Appropriate correlation of Eq. (2.3) with STDP as defined by Fig. (3.2) requires $t_j^n < t_i^{n+1}$ (the post synaptic neuron spikes after the pre synaptic neuron). Considering a temporal window of length Δt preceding the pre-synaptic spike and another one of the same length following it, we have :$\mathbb{P}(\text{pre spikes after post}) = \int_{t_j^n - \Delta t}^{t_j^n} \sigma(s_i^{n+1}) dt < \int_{t_j^n}^{t_j^n + \Delta t} \sigma(s_i^{n+1}) dt = \mathbb{P}(\text{post spikes after pre})$. The same reasonning applying when the post-synaptic activity is decreasing ($\dot{s}_i^{n+1} < 0$), the learning rule Eq. (2.3) correlates on average with STDP, as shown by Fig. (2.1).

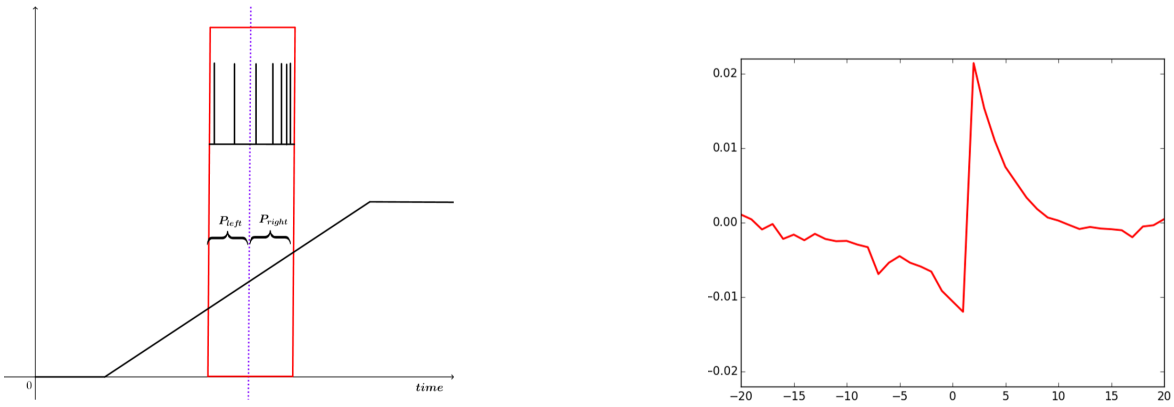

Figure 2.1: **Rate-based STDP** (taken from [153]). Left: when the presynaptic neuron spikes ($\xi_j^n = 1$) at t_i^n (vertical dotted line) and the post-synaptic activity is rising ($\dot{s}_i^{n+1} > 0$), the post-synaptic neuron will most likely spike after the pre-synaptic neuron. Right: Eq. (2.3) yields STDP-like correlations similar to Fig. 3.2. The x-axis denotes the temporal delay between the post-synaptic and pre-synaptic firing times and the y-axis the associated weight update.

Connection to Equilibrium Propagation. Now we have yet to clarify how this rate-based version of STDP defined by Eq. (2.2) is related to the exact learning rule that Equilibrium Propagation prescribes (Eq. (1.5)). Changing slightly Eq. (2.2) into:

$$\Delta w_{n,ij} \sim \sigma(s_j^n) \frac{d\sigma(s_i^{n+1})}{dt}, \tag{2.4}$$

and bearing in mind that the bidirectional connections used in the networks trained by Equilibrium Propagation should account for both pre-synaptic to post-synaptic and post-synaptic to pre-synaptic pressures [97], cumulating the resulting total weight change $\Delta w_{n,ij}$ prescribed by Eq. (2.4) over the second phase (from the free steady state until the nudged steady state) yields:

$$\begin{aligned}\Delta w_{n,ij} &\sim \int_{s_*}^{s_*^\beta} \left(\sigma(s_j^n) \frac{d\sigma(s_i^{n+1})}{dt} + \sigma(s_i^{n+1}) \frac{d\sigma(s_j^n)}{dt} \right) = \int_{s_*}^{s_*^\beta} \frac{d}{dt} (\sigma(s_i^{n+1}) \sigma(s_j^n)) \\ &= \left(\sigma(s_{i,*}^{n+1,\beta}) \sigma(s_{j,*}^{n,\beta}) - \sigma(s_{i,*}^{n+1}) \sigma(s_{j,*}^n) \right), \end{aligned} \tag{2.5}$$

hence the connection between Eq. (2.2) and Equilibrium Propagation.

This relationship is especially of interest for neuromorphic researchers. As we pointed out earlier, STDP can be emulated with memristors in spiking neural networks, assuming the memristor is programmed by the voltage difference created by the pre and post synaptic spikes, where the spike forms are appropriately engineered to create arbitrary STDP-correlation (see Fig. 3.3). Therefore, there is some intuition behind Eq. (2.5) that Equilibrium Propagation could be implemented in a event-driven fashion with memristive devices.

2.2 Generalization of Equilibrium Propagation to Vector Field dynamics

2.2.1 Theory

One limitation of Equilibrium Propagation as it was presented so far is that it requires an energy function, or equivalently, symmetric synaptic connections. This assumption is neither biologically plausible, nor hardware-friendly.

To address this issue, a version of Equilibrium Propagation was proposed [147] where the dynamics of the neurons follow a *vector field* which does not necessarily derives from a potential, namely changing Eq. (1.6) into:

$$\frac{ds}{dt} = \mu(x, s; \theta), \tag{2.6}$$

and the dynamics in the second phase are changed into:

$$\frac{ds}{dt} = \mu(x, s; \theta) - \beta \frac{\partial \ell}{\partial s}. \tag{2.7}$$

Scellier et al. subsequently propose the following learning rule:

$$\Delta\theta = \lim_{\beta \to 0} \frac{1}{\beta} \left(\frac{\partial \mu}{\partial \theta}(s_*) \right)^\top \cdot \left(s_*^\beta - s_* \right). \tag{2.8}$$

Note that when $\mu = -\frac{\partial E}{\partial s}$, Eq. (2.8) and Eq. (1.5) coincide.

2.2.2 Example

Let us consider again the same example that we used to illustrate Equilbrium Propagation. In this context, having model dynamics which do not derive from an energy function amounts to use asymmetric connection between neurons. Before, w_n sustained bi-directional propagation from s^n to s^{n+1}. Now, we denotes $w_{n,n+1}$ the synapses connecting s^{n+1} towards s^n and $w_{n+1,n}$ the reverse synaptic connections. So the dynamics of system during the first phase have to be changed from Eq. (1.10) to:

$$\begin{cases} \frac{d\hat{y}}{dt} = \mu^2 = -\hat{y} + w_{2,1} \cdot \sigma(s^1) \\ \frac{ds^1}{dt} = \mu^1 = -s^1 + w_{1,0} \cdot \sigma(x) + w_{1,2} \cdot \sigma(\hat{y}) \end{cases}, \tag{2.9}$$

and the dynamics of the second phase from Eq. (1.11) to:

$$\begin{cases} \frac{d\hat{y}}{dt} = -s^2 + w_{2,1} \cdot \sigma(s^1) + \beta(y - s^2) \\ \frac{ds^1}{dt} = -s^1 + w_{1,0} \cdot \sigma(x) + w_{1,2} \cdot \sigma(\hat{y}) \end{cases}. \tag{2.10}$$

Applying Eq. (2.8) to these dynamics yield the following parameters updates:

$$\begin{cases} \Delta w_{2,1} = \frac{1}{\beta} \rho(s_*^1) \cdot \left(\hat{y}_*^\beta - \hat{y}_* \right)^\top \\ \Delta w_{1,2} = \frac{1}{\beta} \rho(\hat{y}_*) \cdot \left(s^{1,\beta} - s_*^1 \right)^\top \\ \Delta w_{1,0} = \frac{1}{\beta} \rho(x) \cdot \left(s^{1,\beta} - s_*^1 \right)^\top \end{cases}. \tag{2.11}$$

2.3 Equivalence between Equilibrium Propagation and Recurrent Backpropagation

As we pointed out earlier, Bengio and Fischer heuristically showed that, in the first time steps of the second phase of Equilibrium Propagation, the temporal derivatives of the activations

encoded error signals - see Fig. (1.6). This intuition was first proved formally by Scellier [148] in terms of an equivalence between Equilibrium Propagation and *Recurrent Backpropagation*, an algorithm proposed by Almeida [154] and Pineda [155] which computes the gradients of $\mathcal{L} = \ell(s_*)$ using the same notations as before. To state formally his result, Scellier introduces the notion of *projected cost function*:

$$\mathcal{L}(u, t) = \ell(s(t, u; \theta)), \tag{2.12}$$

where $s(t, u; \theta)$, also called a *flow* in the theory of dynamical systems, denotes the state of the neurons at time step t when the system was initially at u: $s_0 = u$. In other words, $t \to \mathcal{L}(u, t)$ gives the value of the cost function all along the system trajectory when it starts from u. In the context of Equilibrium Propagation, the projected cost function of interest is $\mathcal{L}(s_*, t)$ where the system is initially at the free steady state ($s_0 = s_*$). Recurrent Backpropagation can compute $\mathcal{L}(s_*, t)$ iteratively for increasing t. Based on the way Recurrent Backpropagation carries out gradient computation through time, Scellier shows that the temporal variations of the neurons and of the derivative $\partial E/\partial \theta$ through the second phase of Equilibrium Propagation can compute the exact same gradients as those computed by Recurrent Propagation, namely:

$$\begin{cases} \lim_{\beta \to 0} \frac{1}{\beta} \frac{\partial s_t^\beta}{\partial t} = -\frac{\partial \mathcal{L}(s_*, t)}{\partial s} \\ \lim_{\beta \to 0} \frac{1}{\beta} \left(\frac{\partial E}{\partial \theta}(s_t^\beta; \theta) - \frac{\partial E}{\partial \theta}(s_*; \theta) \right) = -\frac{\partial \mathcal{L}(s_*, t)}{\partial \theta} \\ s_0^\beta = s_* \end{cases} . \tag{2.13}$$

Part of this thesis is dedicated to extending this result to an equivalence between Equilibrium Propagation and Backpropagation Through Time.

Part IV

Updates of Equilibrium Propagation Match gradients of BPTT in an RNN with a Static Input

Summary

In part III, we introduced Equilibrium Propagation as biologically inspired learning algorithm that can be used to train convergent recurrent neural networks, i.e. RNNs that are fed by a static input x and settle to a steady state (section 1.4). Training convergent RNNs consists in adjusting the weights until the steady state of output neurons coincides with a target y (see Eq. (1.2) in section 1.2 of part III). Convergent RNNs can also be trained with the more conventional Backpropagation Through Time (BPTT) algorithm (which was introduced and derived in subsubection 1.2.3 of part III). In its original formulation EP was described in the case of real-time neuronal dynamics (see Eq. (1.6) in section 1.3 of part III), which is computationally costly. In this chapter, we introduce a discrete-time version of EP with simplified equations and with reduced simulation time, bringing EP closer to practical machine learning tasks. We first prove theoretically, as well as numerically that the neural and weight updates of EP, computed by *forward-time* dynamics, are step-by-step equal to the ones obtained by BPTT, with gradients computed *backward in time*. The equality is strict when the transition function of the dynamics derives from a primitive function and the steady state is maintained long enough. We then show for more standard discrete-time neural network dynamics that the same property is approximately respected and we subsequently demonstrate training with EP with equivalent performance to BPTT. In particular, we define the first convolutional architecture trained with EP achieving $\sim 1\%$ test error on MNIST, which is the lowest error reported with EP. These results can guide the development of deep neural networks trained with EP. This chapter is adapted from our paper that was presented at NeurIPS 2019 as an oral contribution [156]. This work was done in collaboration with Benjamin Scellier, who carried out the derivation of theorem 4.

Introduction

The remarkable development of deep learning over the past years [157] has been fostered by the use of backpropagation [11] which stands as the most powerful algorithm to train neural networks. In spite of its success, the backpropagation algorithm is not biologically plausible [158], and its implementation on GPUs is energy-consuming [159]. As we mentionned in part I, hybrid hardware-software experiments carried out by IBM Almaden have recently demonstrated how physics and dynamics can be leveraged to achieve learning with energy efficiency. Hence the motivation to invent novel learning algorithms where both inference and learning could fully be achieved out of core physics.

Many biologically inspired learning algorithms have been proposed as alternatives to backpropagation to train neural networks. As we mentioned in the section 4.2 of part I, Contrastive Hebbian learning (CHL) has been successfully used to train recurrent neural networks (RNNs) with static input that converge to a steady state, such as Boltzmann machines and real-time Hopfield networks. Equilibrium Propagation that we thoroughly introduced in part III also belongs to the family of CHL algorithms to train RNNs with static input. Interestingly, EP also shares similar features with the backpropagation algorithm, and more specifically recurrent backpropagation (RBP): it was proved that neural computation in the second phase of EP is equivalent to gradient computation in RBP - see section 2.3 of part III.

Originally, EP was introduced in the context of leaky integrate-like dynamics for the neurons (see section 1.4 of the previous chapter). Computing their dynamics involves long simulation times, hence limiting EP training experiments to small neural networks. In this chapter, we propose a discrete-time formulation of EP. This formulation allows demonstrating an equivalence between EP and BPTT in specific conditions, simplifies equations and speeds up training, and extends EP to standard neural networks including convolutional ones. Specifically, the contributions of the present work are the following:

- We introduce a discrete-time formulation of EP (Section 2.1) of which the original real-time formulation can be seen as a particular case (Section 4.1).
- We show a step-by-step equality between the updates of EP and the gradients of BPTT when the dynamics converges to a steady state and the transition function of the RNN

derives from a primitive function (Theorem 4, Figure 1). We say that such an RNN has the property of 'gradient-descending updates' (or GDU property).

- We numerically demonstrate the GDU property on a small network, on fully connected layered and convolutional architectures. We show that the GDU property continues to hold approximately for more standard – *prototypical* – neural networks even if these networks do not exactly meet the requirements of Theorem 4.

- We validate our approach with training experiments on different network architectures using discrete-time EP, achieving similar performance than BPTT. We show that the number of iterations in the two phases of discrete-time EP can be reduced by a factor three to five compared to the original real-time EP, without loss of accuracy. This allows us training the first convolutional architecture with EP, reaching $\sim 1\%$ test error on MNIST, which is the lowest test error reported with EP. Our code is available on-line in Pytorch [*].

[*] `https://github.com/ernoult/updatesEPgradientsBPTT`

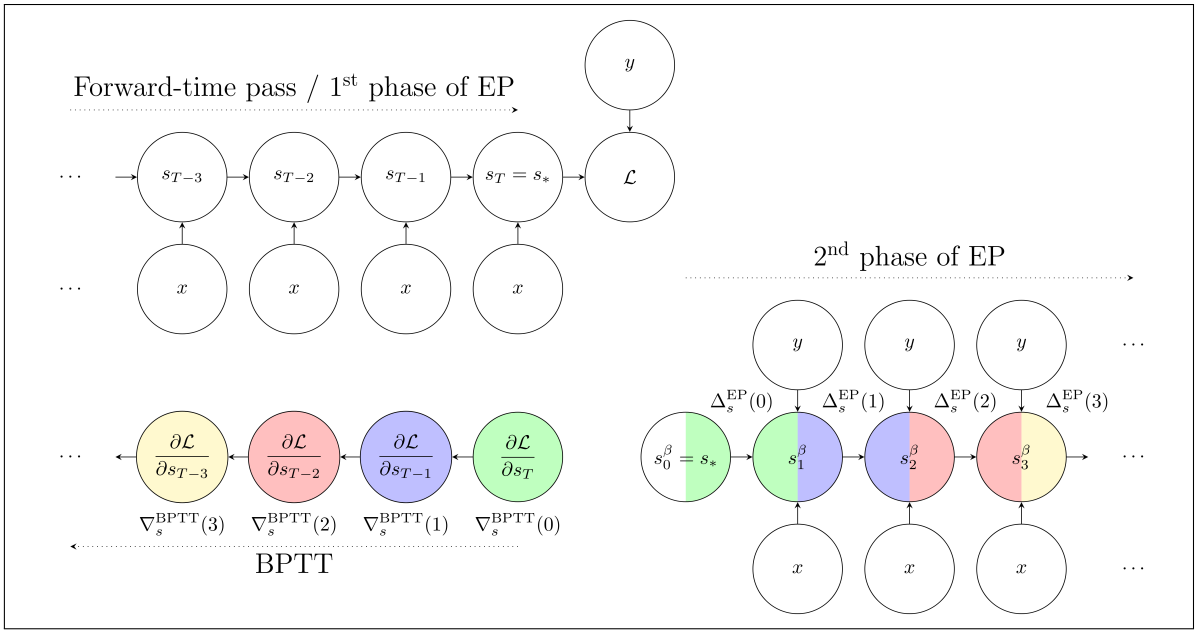

Figure 1: Illustration of the property of Gradient-Descending Updates (GDU property). **Top left.** Forward-time pass (or 'first phase') of an RNN with static input x and target y. The final state s_T is the steady state s_*. **Bottom left.** Backprop through time (BPTT). **Bottom right.** Second phase of equilibrium prop (EP). The starting state in the second phase is the final state of the first phase, i.e. the steady state s_*. **GDU Property (Theorem 4).** Step by step correspondence between the neural updates $\Delta_s^{\mathrm{EP}}(t)$ in the second phase of EP and the gradients $\nabla_s^{\mathrm{BPTT}}(t)$ of BPTT. Corresponding computations in EP and BPTT at timestep $t = 0$ (resp. $t = 1, 2, 3$) are colored in green (resp. blue, red, yellow). Forward-time computation in EP corresponds to backward-time computation in BPTT.

Chapter 1

Background

1.1 Convergent RNNs With Static Input

We consider the supervised setting where we want to predict a target y given an input x. The model is a dynamical system - such as a recurrent neural network (RNN) - parametrized by θ and evolving according to the dynamics:

$$s_{t+1} = F\left(x, s_t; \theta\right). \tag{1.1}$$

We call F the *transition function*. Note that Eq. (1.1) uses the same notations as in the introduction of this thesis (part I). The input of the RNN at each timestep is static, equal to x. Assuming convergence of the dynamics before time step T, we have $s_T = s_*$ where s_* is such that

$$s_* = F\left(x, s_*; \theta\right). \tag{1.2}$$

We call s_* the *steady state* (or fixed point, or equilibrium state) of the dynamical system. The number of timesteps T is a hyperparameter chosen large enough to ensure $s_T = s_*$. The goal of learning is to optimize the parameter θ to minimize the loss:

$$\mathcal{L}^* = \ell\left(s_*, y\right), \tag{1.3}$$

where the scalar function ℓ is called *cost function*. Several algorithms have been proposed to optimize the loss $\mathcal{L}^*$, including Recurrent Backpropagation (RBP) and Equilibrium Propagation (EP), as we saw in section 2.3 of the previous chapter. Thereafter, we reformulate with new notations Backpropagation Through Time (BPTT) which was introduced in subsection 1.2.3 of part I so as to enunciate the main theoretical result of this paper (Theorem 4).

1.2 Backpropagation Through Time (BPTT)

BPTT was introduced in subsection 1.2.3 of part I, and we use here the same notations, substituting the index n by t to emphasize that computation happens through time.

With frameworks such as Pytorch or Tensorflow implementing automatic differentiation, optimization by gradient descent using Backpropagation Through Time (BPTT) has become the standard method to train RNNs. In particular BPTT can be used for a convergent RNN such as the one that we study here. To this end, we consider the loss after T iterations (i.e. the cost of the final state s_T), denoted $\mathcal{L} = \ell(s_T, y)$, and we substitute $\mathcal{L}$ as a proxy * for the loss at the steady state $\mathcal{L}^*$. The gradients of $\mathcal{L}$ can be computed with BPTT. In order to state our Theorem 4 (chapter 3), we recall some of the inner working mechanisms of BPTT. Eq. (1.1) can be rewritten in the form $s_{t+1} = F(x, s_t, \theta_t = \theta)$, where θ_t denotes the parameter of the model at time step t, the value θ being shared across all time steps. This way of rewriting Eq. (1.1) enables us to define the partial derivative $\frac{\partial \mathcal{L}}{\partial \theta_t}$ as the sensitivity of the loss $\mathcal{L}$ with respect to θ_t when $\theta_1, \dots \theta_{t-1}, \theta_{t+1}, \dots \theta_T$ remain fixed (set to the value θ). With these notations, the gradient $\frac{\partial \mathcal{L}}{\partial \theta}$ reads as the sum:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial \theta_1} + \frac{\partial \mathcal{L}}{\partial \theta_2} + \cdots + \frac{\partial \mathcal{L}}{\partial \theta_T}. \tag{1.4}$$

This equation is the same as Eq. (1.18) of subsection 1.2.3 (part I), we simply recall it here for the completeness of this chapter. BPTT computes the "full" gradient $\frac{\partial \mathcal{L}}{\partial \theta}$ by computing the partial derivatives $\frac{\partial \mathcal{L}}{\partial s_t}$ and $\frac{\partial \mathcal{L}}{\partial \theta_t}$ iteratively and efficiently, backward in time, using the chain rule of differentiation. Subsequently, we denote the gradients that BPTT computes:

$$\forall t \in [0, T-1] : \begin{cases} \nabla_s^{\mathrm{BPTT}}(t) = \dfrac{\partial \mathcal{L}}{\partial s_{T-t}} \\ \nabla_\theta^{\mathrm{BPTT}}(t) = \dfrac{\partial \mathcal{L}}{\partial \theta_{T-t}}, \end{cases} \tag{1.5}$$

so that

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{t=0}^{T-1} \nabla_\theta^{\mathrm{BPTT}}(t). \tag{1.6}$$

The gradients $\nabla_s^{\mathrm{BPTT}}(t)$ and $\nabla_\theta^{\mathrm{BPTT}}(t)$ are the 'elementary gradients' computed as intermediary steps in BPTT in order to compute the 'full gradient' $\frac{\partial \mathcal{L}}{\partial \theta}$. We now reformulate Eq. (1.19) with this new set of notations.

*The difference between the loss $\mathcal{L}$ and the loss $\mathcal{L}^*$ is explained in Appendix 2.1.

Proposition 1 (Backpropagation Through Time)**.** *The gradients* $\nabla_s^{\mathrm{BPTT}}(t)$ *and* $\nabla_\theta^{\mathrm{BPTT}}(t)$ *can be computed using the recurrence relationship*

$$\nabla_s^{\mathrm{BPTT}}(0) = \frac{\partial \ell}{\partial s}\left(s_T, y\right), \tag{1.7}$$

$$\forall t = 1, 2, \ldots, T, \qquad \nabla_s^{\mathrm{BPTT}}(t) = \frac{\partial F}{\partial s}\left(x, s_{T-t}; \theta\right)^\top \cdot \nabla_s^{\mathrm{BPTT}}(t-1), \tag{1.8}$$

$$\forall t = 1, 2, \ldots, T, \qquad \nabla_\theta^{\mathrm{BPTT}}(t) = \frac{\partial F}{\partial \theta}\left(x, s_{T-t}; \theta\right)^\top \cdot \nabla_s^{\mathrm{BPTT}}(t-1). \tag{1.9}$$

Proof of Proposition 1. This is a direct application of the chain rule of differentiation, using the fact that $s_{t+1} = F\left(x, s_t, \theta\right)$ - exactly as we did for Eq. (1.19) in subsection 1.2.3 of part I □

Chapter 2

A discrete-time formulation of Equilibrium Propagation

2.1 Algorithm

In its original formulation, Equilibrium Propagation (EP) was introduced in the case of real-time dynamics - see section 1.4 of part III. The first theoretical contribution of this chapter is to adapt the theory of EP to discrete-time dynamics. EP is an alternative algorithm to compute the gradient of $\mathcal{L}^*$ in the particular case where the transition function F derives from a scalar function Φ, i.e. with F of the form $F(x, s, \theta) = \frac{\partial \Phi}{\partial s}(x, s; \theta)$. The algorithmics of our discrete-time version are exactly the same as the one of the original version of Equilibrium Propagation described in section 1.3 of part III. In this setting, the dynamics of Eq. (1.1) rewrites:

$$\forall t \in [0, T-1], \qquad s_{t+1} = \frac{\partial \Phi}{\partial s}(x, s_t; \theta). \tag{2.1}$$

This constitutes the first phase of EP. At the end of the first phase, we have reached steady state, i.e. $s_T = s_*$. In the second phase of EP, starting from the steady state s_*, an extra term $\beta \frac{\partial \ell}{\partial s}$ (where β is a positive scaling factor) is introduced in the dynamics of the neurons and acts as an external force nudging the system dynamics towards decreasing the cost function ℓ. Denoting $s_0^\beta, s_1^\beta, s_2^\beta, \dots$ the sequence of states in the second phase (which depends on the value of β), the dynamics is defined as

$$s_0^\beta = s_* \qquad \text{and} \qquad \forall t \geq 0, \quad s_{t+1}^\beta = \frac{\partial \Phi}{\partial s}\left(x, s_t^\beta; \theta\right) - \beta \, \frac{\partial \ell}{\partial s}\left(s_t^\beta, y\right). \tag{2.2}$$

The network eventually settles to a new steady state s_*^β. It was shown in [160] that the gradient of the loss $\mathcal{L}^*$ can be computed based on the two steady states s_* and s_*^β. More

specifically, * in the limit $\beta \to 0$,

$$\frac{1}{\beta}\left(\frac{\partial \Phi}{\partial \theta}\left(x, s_*^\beta; \theta\right) - \frac{\partial \Phi}{\partial \theta}\left(x, s_*; \theta\right)\right) \to -\frac{\partial \mathcal{L}^*}{\partial \theta}. \tag{2.3}$$

2.2 Difference between the primitive function Φ and the energy function E

We want to highlight here the relationship between the discrete-time setting (resp. the primitive function Φ) of this paper and the real-time setting (resp. the energy function E) of [148, 160].

As we shown in part III, previous work on EP has studied real-time dynamics of the form:

$$\frac{ds}{dt} = -\frac{\partial E}{\partial s}\left(x, s_t; \theta\right). \tag{2.4}$$

In contrast, in this chapter we study discrete-time dynamics of the form

$$s_{t+1} = \frac{\partial \Phi}{\partial s}\left(x, s_t; \theta\right). \tag{2.5}$$

Here we explain why we changed the sign convention in the dynamics and why we called Φ a 'primitive function' rather than an 'energy function'.

While it is useful to think of the primitive function Φ in the discrete-time setting as an equivalent of the energy function E in the real-time setting, there is an important difference between E and Φ. We argue next that, rather than an energy function, Φ is much better thought of as a primitive of the transition function F. First we show how the two settings are related.

Casting real-time dynamics to discrete-time dynamics. The real-time dynamics of Eq. (2.4) can be cast to the discrete-time setting of Eq. (2.5) as follows. The Euler scheme of Eq. (2.4) with discretization step ε reads:

$$s_{t+1} = s_t - \varepsilon \frac{\partial E}{\partial s}\left(x, s_t; \theta\right). \tag{2.6}$$

This equation rewrites

$$s_{t+1} = \frac{\partial \Phi_\varepsilon}{\partial s}\left(x, s_t; \theta\right), \qquad \text{where} \qquad \Phi_\varepsilon(x, s, \theta) = \frac{1}{2}\|s\|^2 - \varepsilon\, E(x, s; \theta). \tag{2.7}$$

However, although the real-time dynamics can be mapped to the discrete-time setting, the discrete-time setting is more general.

*The EP learning rule is a form of contrastive Hebbian learning similar to that of Boltzmann machines [69] and similar to the one presented in [72].

The scalar function Φ is better thought of as a primitive function of F than of an energy. The primitive function Φ cannot be interpreted in terms of an energy in general. In the real-time setting, s_t *follows* the gradient of E, so that $E(s_t)$ decreases as time progresses until s_t settles to a (local) minimum of E. This property motivates the name of 'energy function' for E by analogy with physical systems whose dynamics settle down to low-energy configurations. In contrast, in the discrete-time setting, s_t *is mapped* onto the gradient of Φ (at the point s_t). In general, there is no guarantee that the discrete-time dynamics of Eq. (2.5) optimizes Φ and there is no guarantee that the dynamics of s_t converges to an optimum of Φ. For this reason, there is no reason to call Φ an 'energy function', since the intuition of optimizing an energy does not hold.

The name of 'primitive function' for Φ is motivated by the fact that Φ is a primitive of the transition function F, whose property better captures the assumptions under which the theory of EP holds. To see this, let us consider again the general form of the dynamics as defined by Eq. (1.1) with:

$$F(x, s, \theta) = \frac{\partial \Phi}{\partial s}(x, s; \theta)\,. \tag{2.8}$$

For the theory of EP to hold (in particular Theorem 4 as we shall see later), the following two conditions must be satisfied (see Lemma 2 and Lemma 3 in chapter 3):

1. The steady state s_* (at the end of the first phase and at the beginning of the second phase) must satisfy the condition

$$s_* = F(x, s_*; \theta)\,, \tag{2.9}$$

2. the Jacobian of the transition function F must be symmetric, i.e.

$$\frac{\partial F}{\partial s}(x, s; \theta)^\top = \frac{\partial F}{\partial s}(x, s; \theta). \tag{2.10}$$

The condition of Eq. (2.10) is equivalent to the existence of a scalar function $\Phi(x, s, \theta)$ such that Eq. (2.8) holds. Going from Eq. (2.8) to Eq. (2.10) is straightforward: in this case the Jacobian of F is the Hessian of Φ, which is symmetric. Indeed $\frac{\partial F}{\partial s}(x, s; \theta) = \frac{\partial^2 \Phi}{\partial s^2}(x, s; \theta) = \frac{\partial F}{\partial s}(x, s; \theta)^\top$. Going from Eq. (2.10) to Eq. (2.8) is also true – though less obvious – and is a consequence of Green's theorem. * We say that F derives from the scalar function Φ, or that Φ is a primitive of F. Hence the name of 'primitive function' for Φ.

Assumption of Convergence in the Discrete-Time Setting. In the real-time setting, the gradient dynamics of Eq. (2.4) guarantees convergence to a (local) minimum of E. In contrast, in the discrete-time setting, no intrinsic property of F or Φ a priori guarantees that

*Another equivalent formulation is that the curl of F is null, i.e. $\vec{F} = \vec{0}$.

the dynamics of Eq (2.5) settles to steady state. This discussion is out of the scope of this work and we refer to [161] where sufficient (but not necessary) conditions are discussed to ensure convergence based on the contraction map theorem.

Chapter 3

Forward-Time Dynamics of EP Compute Backward-Time Gradients of BPTT

Note that for fixed $\beta > 0$, defining the neural and weight updates:

$$\begin{cases} \forall t \geq 0: & \Delta_s^{\mathrm{EP}}(\beta, t) = \dfrac{1}{\beta}\left(s_{t+1}^{\beta} - s_t^{\beta}\right), \\ \forall t \geq 1: & \Delta_\theta^{\mathrm{EP}}(\beta, t) = \dfrac{1}{\beta}\left(\dfrac{\partial \Phi}{\partial \theta}\left(x, s_t^{\beta}, \theta\right) - \dfrac{\partial \Phi}{\partial \theta}\left(x, s_{t-1}^{\beta}, \theta\right)\right), \end{cases} \tag{3.1}$$

Eq. (2.3) rewrites as the following telescoping sum:

$$\sum_{t=0}^{\infty} \Delta_\theta^{\mathrm{EP}}(\beta, t) \to -\frac{\partial \mathcal{L}^*}{\partial \theta} \qquad \text{as} \qquad \beta \to 0. \tag{3.2}$$

Therefore, BPTT and EP compute the gradient of the loss in very different ways: while the former algorithm iteratively adds up gradients going backward in time, as in Eq. (1.6), the latter algorithm adds up weight updates going forward in time, as in Eq. (3.2). In fact, under a condition stated below, the sums are equal term by term: there is a step-by-step correspondence between the two algorithms. To prove our main result, we first prove two intermediate Lemmas. Theorem 4 is a consequence of Lemma 2 and Lemma 3 below, which are stated for general dynamics with a transition function F. Theorem 4 proves formally, in the context of discrete-time dynamics, the intuition presented in part III that temporal derivatives of the system during the second phase of Equilibrium Propagation encode error gradients.

3.1 Backpropagation Through Time error processes

Lemma 2. *In our specific setting with static input x, suppose that the network has reached the steady state s_* after $T-K$ steps, i.e.*

$$s_{T-K} = s_{T-K+1} = \cdots = s_{T-1} = s_T = s_*. \tag{3.3}$$

Then the first K gradients of BPTT satisfy the recurrence relationship *

$$\nabla_s^{\mathrm{BPTT}}(0) = \frac{\partial \ell}{\partial s}(s_*, y), \tag{3.4}$$

$$\forall t = 1, 2, \ldots, K, \qquad \nabla_s^{\mathrm{BPTT}}(t) = \frac{\partial F}{\partial s}(x, s_*, \theta)^\top \cdot \nabla_s^{\mathrm{BPTT}}(t-1), \tag{3.5}$$

$$\forall t = 1, 2, \ldots, K, \qquad \nabla_\theta^{\mathrm{BPTT}}(t) = \frac{\partial F}{\partial \theta}(x, s_*, \theta)^\top \cdot \nabla_s^{\mathrm{BPTT}}(t-1). \tag{3.6}$$

Proof of Lemma 2. This is a direct application of Proposition 1 along with $s_{T-K} = s_{T-K+1} = \cdots = s_{T-1} = s_T = s_*$ so that we evaluate the Jacobians $\partial F/\partial s$ and $\partial F/\partial \theta$ at s_*. □

3.2 Equilbrium Propagation error processes

Before stating Lemma 3, we formulate EP for arbitrary transition function F, inspired by the ideas of [147]. Recall that at the beginning of the second phase of EP the state of the network is the steady state $s_0^\beta = s_*$ characterized by

$$s_* = F(x, s_*, \theta), \tag{3.7}$$

and that, given some value $\beta > 0$ of the hyperparameter β, the successive neural states $s_1^\beta, s_2^\beta, \ldots$ are defined and computed as follows:

$$\forall t \geq 0, \qquad s_{t+1}^\beta = F\left(x, s_t^\beta, \theta\right) - \beta \frac{\partial \ell}{\partial s}\left(s_t^\beta, y\right). \tag{3.8}$$

In this more general setting, we redefine the 'weight updates' as:

$$\forall t \geq 1, \qquad \Delta_\theta^{\mathrm{EP}}(\beta, t) = \frac{1}{\beta}\frac{\partial F}{\partial \theta}\left(x, s_{t-1}^\beta, \theta\right)^\top \cdot \left(s_t^\beta - s_{t-1}^\beta\right). \tag{3.9}$$

Note that when $F = \frac{\partial \Phi}{\partial s}$, in the liimt $\beta \to 0$, Eq. (3.9) coincide with the previous definition of $\Delta_\theta^{\mathrm{EP}}$ (Eq. (3.1)):

*Note that the stability of the steady state implies that the eigenvalues of the Jacobian $\frac{\partial F}{\partial s}(x, s_*, \theta)$ are smaller than 1 in magnitude. As a consequence of Lemma 2, the gradients $\nabla_\theta^{\mathrm{BPTT}}(t)$ decay (i.e. vanish) exponentially fast, which ensures that the full gradient $\sum_{t=0}^{K-1} \nabla_\theta^{\mathrm{BPTT}}(t)$ converges, even if $K \to \infty$. In the context of convergent RNNs with a static input, vanishing gradients of BPTT are consequently not a problem, as it is the case when learning from temporal data with RNNs.

$$\begin{aligned}\Delta_\theta^{\mathrm{EP}}(t+1) &= \frac{1}{\beta}\left(\frac{\partial \Phi}{\partial \theta}\left(x, s_{t+1}^\beta, \theta\right) - \frac{\partial \Phi}{\partial \theta}\left(x, s_t^\beta, \theta\right)\right) \\ &= \frac{1}{\beta}\underbrace{\frac{\partial^2 \Phi}{\partial s \partial \theta}\left(x, s_t^\beta, \theta\right)}_{=\frac{\partial F}{\partial s}\left(x, s_t^\beta, \theta\right)^\top} \cdot \left(s_{t+1}^\beta - s_t^\beta\right) + o(\beta)\end{aligned}$$

Lemma 3. *Let $\Delta_s^{\mathrm{EP}}(t) = \lim_{\beta\to 0} \Delta_s^{\mathrm{EP}}(\beta, t)$ and $\Delta_\theta^{\mathrm{EP}}(t) = \lim_{\beta\to 0} \Delta_\theta^{\mathrm{EP}}(\beta, t)$ be the neural and weight updates of EP in the limit $\beta \to 0$. They satisfy the recurrence relationship*

$$\Delta_s^{\mathrm{EP}}(0) = -\frac{\partial \ell}{\partial s}(s_*, y), \tag{3.10}$$

$$\forall t \geq 0, \qquad \Delta_s^{\mathrm{EP}}(t+1) = \frac{\partial F}{\partial s}(x, s_*, \theta) \cdot \Delta_s^{\mathrm{EP}}(t), \tag{3.11}$$

$$\forall t \geq 0, \qquad \Delta_\theta^{\mathrm{EP}}(t+1) = \frac{\partial F}{\partial \theta}(x, s_*, \theta)^\top \cdot \Delta_s^{\mathrm{EP}}(t). \tag{3.12}$$

Proof of Lemma 3. First, in the limit $\beta \to 0$, the weight update $\Delta_\theta^{\mathrm{EP}}(\beta, t)$ of Eq. (3.9) simply rewrites as Eq. (3.12) by evaluating $\partial F/\partial s$ at s_* and using the definition of Δ_s^{EP} in Eq. (3.1). Now we prove Eq. (3.10) and Eq. (3.11). Note that the neural update $\Delta_s^{\mathrm{EP}}(\beta, t)$ of Eq. (3.1) rewrites:

$$\Delta_s^{\mathrm{EP}}(t) = \left.\frac{\partial s_{t+1}^\beta}{\partial \beta}\right|_{\beta=0} - \left.\frac{\partial s_t^\beta}{\partial \beta}\right|_{\beta=0}. \tag{3.13}$$

This is because for every $t \geq 0$ we have $s_t^\beta \to s_*$ as $\beta \to 0$: starting from $s_0^0 = s_*$, setting $\beta = 0$ in Eq. (3.8) yields $s_1^0 = s_2^0 = \ldots = s_*$.

Differentiating Eq. (3.8) with respect to β, we get:

$$\forall t \geq 0, \qquad \frac{\partial s_{t+1}^\beta}{\partial \beta} = \frac{\partial F}{\partial s}\left(x, s_t^\beta, \theta\right) \cdot \frac{\partial s_t^\beta}{\partial \beta} - \frac{\partial \ell}{\partial s}\left(s_t^\beta, y\right) - \beta \frac{\partial^2 \ell}{\partial s^2}\left(s_t^\beta, y\right) \cdot \frac{\partial s_t^\beta}{\partial \beta}. \tag{3.14}$$

Letting $\beta \to 0$, we have $s_t^\beta \to s_*$, so that:

$$\forall t \geq 0, \qquad \left.\frac{\partial s_{t+1}^\beta}{\partial \beta}\right|_{\beta=0} = \frac{\partial F}{\partial s}(x, s_*, \theta) \cdot \left.\frac{\partial s_t^\beta}{\partial \beta}\right|_{\beta=0} - \frac{\partial \ell}{\partial s}(s_*, y). \tag{3.15}$$

Since at time $t = 0$ the initial state of the network $s_0^\beta = s_*$ is independent of β, we have:

$$\frac{\partial s_0^\beta}{\partial \beta} = 0. \tag{3.16}$$

Using Eq. (3.15) for $t = 0$ and Eq. (3.16), we get the initial condition on $\Delta_s^{\mathrm{EP}}(0)$ (Eq. (3.10))

$$\Delta_s^{\mathrm{EP}}(0) = \left.\frac{\partial s_1^\beta}{\partial \beta}\right|_{\beta=0} - \left.\frac{\partial s_0^\beta}{\partial \beta}\right|_{\beta=0} = -\frac{\partial \ell}{\partial s}(s_*, y)\,. \tag{3.17}$$

Moreover, if we take Eq. (3.15) and subtract itself from it at time step $t-1$, we get:

$$\Delta_s^{\mathrm{EP}}(t+1) = \frac{\partial F}{\partial s}(x, s_*, \theta) \cdot \Delta_s^{\mathrm{EP}}(t). \tag{3.18}$$

Hence Eq. (3.11) and the final result. □

3.3 Main result

We can now state our main theoretical result.

Theorem 4 (Gradient-Descending Updates, GDU)**.** *Consider the setting with a transition function of the form $F(x, s, \theta) = \frac{\partial \Phi}{\partial s}(x, s, \theta)$. Let $s_0, s_1, \ldots, s_T$ be the convergent sequence of states and denote $s_* = s_T$ the steady state. If we further assume that there exists some step K where $0 < K \leq T$ such that $s_* = s_T = s_{T-1} = \ldots s_{T-K}$, then, in the limit $\beta \to 0$, the first K updates in the second phase of EP are equal to the negatives of the first K gradients of BPTT, i.e.*

$$\forall t = 0, 1, \ldots, K : \begin{cases} \Delta_s^{\mathrm{EP}}(\beta, t) \to -\nabla_s^{\mathrm{BPTT}}(t), \\ \Delta_\theta^{\mathrm{EP}}(\beta, t) \to -\nabla_\theta^{\mathrm{BPTT}}(t). \end{cases} \tag{3.19}$$

Proof of Theorem 4. Note that in the context of convergent RNNs, the equations governing BPTT (Eq.(3.4)-(3.6)) and those governing EP (Eq. (3.10)-(3.12)) only differ by one matrix multiplication: the error signals carried by the neurons in BPTT are transmitted from one timestep to another through $\frac{\partial F}{\partial s}(x, s_*; \theta)^\top$ (Eq. (3.5)) while neural updates in the second phase of EP are transmitted through $\frac{\partial F}{\partial s}(x, s_*; \theta)$ (Eq. (3.11)). Other things being equal (including initial condition), $-\nabla_s^{\mathrm{BPTT}}$ (resp. $-\nabla_\theta^{\mathrm{BPTT}}$) and Δ_s^{EP} (resp. $\Delta_\theta^{\mathrm{EP}}$) satisfy different recurrence relationship, thereby are different in general. However, since for since we assume here that F is of the form $F(x, s, \theta) = \frac{\partial \Phi}{\partial s}(x, s, \theta)$, the Jacobian matrix of the transition function F is the Hessian of Φ, thus is symmetric:

$$\frac{\partial F}{\partial s}(x, s, \theta)^\top = \frac{\partial^2 \Phi}{\partial s^2}(x, s, \theta) = \frac{\partial F}{\partial s}(x, s, \theta)\,. \tag{3.20}$$

Consequently, $-\nabla_s^{\mathrm{BPTT}}$ (resp. $-\nabla_\theta^{\mathrm{BPTT}}$) and Δ_s^{EP} (resp. $\Delta_\theta^{\mathrm{EP}}$) satisfy the same recurrence relationship with the same initial condition, so that they are equal at all time.

□

The convergence requirement enables to derive the equations satisfied by the neural and weight updates (Lemma 3). Then, the existence of a primitive function ensures that these equations are equal to those satisfied by the gradients of BPTT (Lemma 2), with same initial conditions, yielding the desired equality (Theorem 4).

Note that other algorithms such as RTRL [92] and UORO [162] also compute the gradients by forward-time dynamics.

Chapter 4

Energy-based and Prototypical settings

In this chapter, we introduce the two classes of models we have considered in this study.

4.1 Definition

Energy-based setting. The system is defined in terms of a primitive function of the form:

$$\Phi_\varepsilon(s; W, W_x) = (1-\varepsilon)\frac{1}{2}\|s\|^2 + \varepsilon\,\left(\sigma(s)^\top \cdot W \cdot \sigma(s) + \sigma(s)^\top \cdot W_x \cdot \sigma(x)\right), \tag{4.1}$$

where ε is a discretization parameter, σ is an activation function, W is a symmetric weight matrix and W_x the weight matrix connecting the input to the system. In this setting, we consider $\Delta^{\mathrm{EP}}(\beta\varepsilon, t)$ instead of $\Delta^{\mathrm{EP}}(\beta, t)$ and write $\Delta^{\mathrm{EP}}(t)$ for simplicity, so that:

$$\begin{aligned}
\Delta_s^{\mathrm{EP}}(\beta, t) &= \frac{s_{t+1}^{\beta\varepsilon} - s_t^{\beta\varepsilon}}{\beta\varepsilon} \\
\Delta_W^{\mathrm{EP}}(\beta, t) &= \frac{1}{\beta}\left(\sigma\left(s_{t+1}^{\beta\varepsilon}\right)^\top \cdot \sigma\left(s_{t+1}^{\beta\varepsilon}\right) - \sigma\left(s_t^{\beta\varepsilon}\right)^\top \cdot \sigma\left(s_t^{\beta\varepsilon}\right)\right). \\
\Delta_{W_x}^{\mathrm{EP}}(\beta, t) &= \frac{1}{\beta}\left(\sigma\left(s_{t+1}^{\beta\varepsilon}\right)^\top \cdot \sigma\left(x\right) - \sigma\left(s_t^{\beta\varepsilon}\right)^\top \cdot \sigma\left(x\right)\right).
\end{aligned} \tag{4.2}$$

With Φ_ε as a primitive function and with the hyperparameter β rescaled by a factor ε, we recover the discretized version of the real-time setting of [160], i.e. the Euler scheme of $\frac{ds}{dt} = -\frac{\partial E}{\partial s} - \beta\frac{\partial \ell}{\partial s}$ with $E = \frac{1}{2}\|s\|^2 - \sigma(s)^\top \cdot W \cdot \sigma(s) - \sigma(s)^\top \cdot W_x \cdot \sigma(x)$. We will show that up to defining properly W and W_x, these equations apply for any number of layers.

Prototypical setting. In this case, the dynamics of the system does not derive from a primitive function Φ. Instead, the dynamics is directly defined as:

$$s_{t+1} = \sigma\left(W \cdot s_t + W_x \cdot x\right), \tag{4.3}$$

where again we same the same notations as above for W and W_x. The dynamics of Eq. (4.3) is a standard and simple neural network dynamics: we will show that up to defining properly W and W_x, these dynamics apply for any number of layers. Although the model is not defined in terms of a primitive function, note that $s_{t+1} \approx \frac{\partial \Phi}{\partial s}\left(s_t; W, W_x\right)$ with $\Phi(s; W, W_x) = \frac{1}{2} s^\top \cdot W \cdot s \frac{1}{2} s^\top \cdot W_x \cdot x$ if we ignore the activation function σ. Following Eq. (3.1), we define:

$$\begin{aligned}
\Delta_s^{\mathrm{EP}}(\beta, t) &= \frac{1}{\beta}\left(s_{t+1}^{\beta} - s_t^{\beta}\right), \\
\Delta_W^{\mathrm{EP}}(\beta, t) &= \frac{1}{\beta}\left(s_{t+1}^{\beta^\top} \cdot s_{t+1}^{\beta} - s_t^{\beta^\top} \cdot s_t^{\beta}\right), \\
\Delta_{W_x}^{\mathrm{EP}}(\beta, t) &= \frac{1}{\beta}\left(s_{t+1}^{\beta^\top} \cdot x - s_t^{\beta^\top} \cdot x\right)
\end{aligned} \tag{4.4}$$

4.2 Demonstrating the property of Gradient Descending Updates (GDU)

The approach we propose is to use Theorem 4 as a tool to design neural networks that are trainable with EP: if a model satisfies the GDU property of Eq. 3.19, then we expect EP to perform as well as BPTT on this model. We have defined the *energy-based setting* and *prototypical setting* where the conditions of Theorem 4 are exactly and approximately met respectively (Section 4.1). After introducing our protocol, we show the GDU property on a toy model (Fig. 4.1) and on fully connected layered architectures in the two settings (subsubection 4.3.2 and subsubection 4.4.1). We define a convolutional architecture in the prototypical setting (Section 4.4.2) which also satisfies the GDU property. Finally, we validate our approach by training these models with EP and BPTT (Table 5.1).

Property of Gradient-Descending Updates. We say that a convergent RNN model fed with a fixed input has the *GDU property* if during the second phase, the updates it computes by EP (Δ^{EP}) on the one hand and the gradients it computes by BPTT ($-\nabla^{\mathrm{BPTT}}$) on the other hand are 'equal' - or 'approximately equal', as measured per the RelMSE (Relative Mean Squared Error) metric.

Relative Mean Squared Error (RelMSE). In order to quantitatively measure how well the GDU property is satisfied, we introduce a metric which we call Relative Mean Squared Error (RelMSE) such that RelMSE(Δ^{EP}, -∇^{BPTT}) measures the distance between Δ^{EP} and

$-\nabla^{\text{BPTT}}$ processes, averaged over time, over neurons or synapses (layer-wise) and over a mini-batch of samples - see Appendix 2.3.2 for the details.

Protocol. In order to measure numerically if a given model satisfies the GDU property, we proceed as follows. Considering an input x and associated target y, we perform the first phase for T steps. Then we perform on the one hand BPTT for K steps (to compute the gradients ∇^{BPTT}), on the other hand EP for K steps (to compute the neural updates Δ^{EP}) and compare the gradients and neural updates provided by the two algorithms, either qualitatively by looking at the plots of the curves (as in Figs. 4.1 and 4.12), or quantitatively by computing their RelMSE (as in Fig. 5.1).

4.3 Real-time RNNs in the energy-based setting

4.3.1 Toy model

Equations. The toy model is an architecture where input, hidden and output neurons are connected altogether, without lateral connections. Denoting input neurons as x, hidden neurons as s^1 and output neurons as $s^2 = \hat{y}$, the primitive function for this model reads:

$$\begin{aligned}\Phi\left(x, s^1, \hat{y}\right) = (1-\varepsilon)\frac{1}{2}\left(||s^1||^2 + ||\hat{y}||^2\right) \\ +\varepsilon\left(\sigma(\hat{y})^\top \cdot w_1 \cdot \sigma(s^1) + \sigma(s^1)^\top \cdot w_{1,0} \cdot \sigma(x) + \sigma(\hat{y})^\top \cdot w_{\hat{y},0} \cdot \sigma(x)\right),\end{aligned}$$

where ε is a discretization parameter. Furthermore the cost function ℓ is

$$\ell\left(\hat{y}, y\right) = \frac{1}{2}\|\hat{y} - y\|^2. \tag{4.5}$$

As a reminder, we define the following convention for the dynamics of the second phase: $\forall t \in [0, K] : s_t^{n,\beta} = s_{t+T}^n$ where T is the length of the first phase. The equations of motion read in the first phase read:

$$\forall t \in [0, T] : \left\{ \begin{array}{ll} \hat{y}_{t+1} & = (1-\varepsilon)\hat{y}_t + \varepsilon\sigma'(\hat{y}_t) \odot (w_1 \cdot \sigma(s_t^1) + w_{\hat{y},0} \cdot \sigma(x)) \\ s_{t+1}^1 & = (1-\varepsilon)s_t^1 + \varepsilon\sigma'(s_t^1) \odot (w_1^\top \cdot \sigma(\hat{y}_t) + w_{1,0} \cdot \sigma(x)), \end{array} \right.$$

and in the second phase:

$$\forall t \in [0, K] : \left\{ \begin{array}{ll} \hat{y}_{t+1}^\beta & = (1-\varepsilon)\hat{y}_t^\beta + \varepsilon\sigma'(\hat{y}_t^\beta) \odot (w_1 \cdot \sigma(s_t^{1,\beta}) + w_{\hat{y},0} \cdot \sigma(x)) \\ & +\varepsilon\beta(y - \hat{y}_t^\beta) \\ s_{t+1}^{1,\beta} & = (1-\varepsilon)s_t^{1,\beta} + \varepsilon\sigma'(s_t^{1,\beta}) \odot (w_1^\top \cdot \sigma(\hat{y}_t^\beta) + w_{1,0} \cdot \sigma(x)), \end{array} \right. \tag{4.6}$$

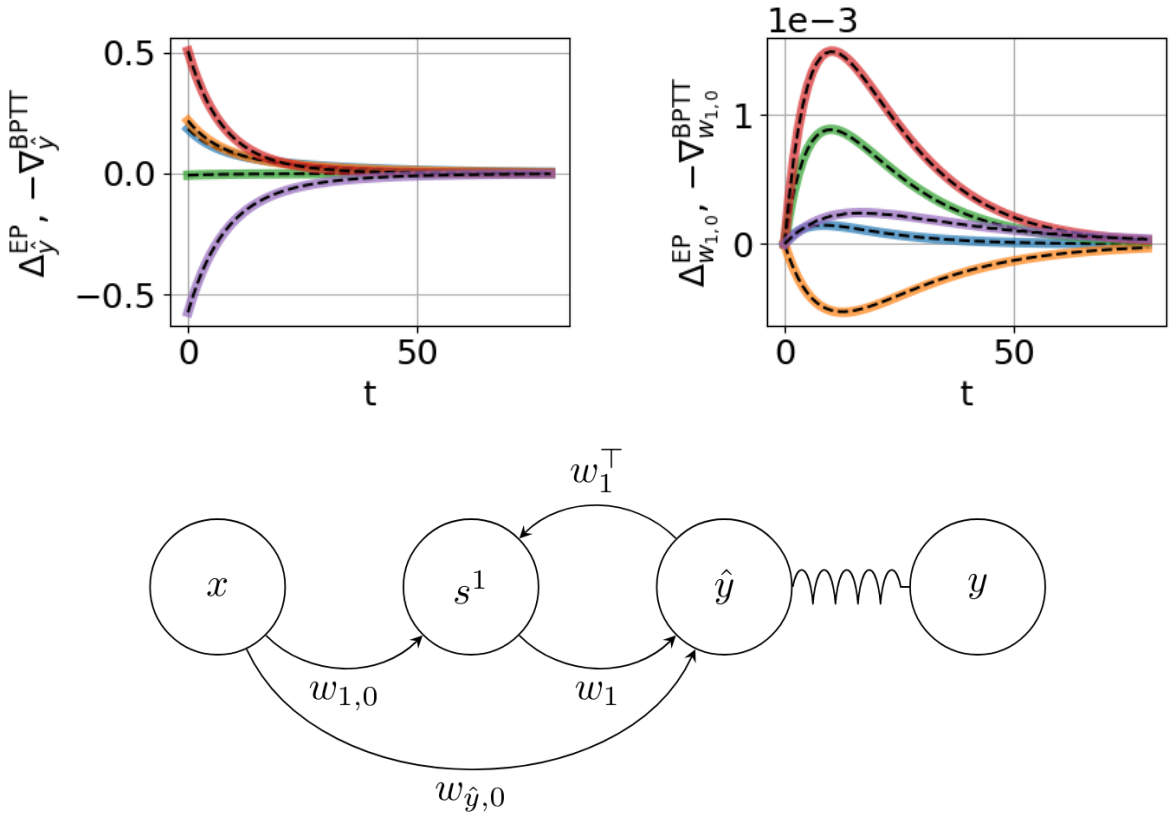

Figure 4.1: Demonstrating the property of gradient-descending updates in the energy-based setting on a toy model with dummy data x and a target y elastically nudging the output neurons s^0 (right). Dashed and solid lines represent Δ^{EP} and $-\nabla^{\text{BPTT}}$ processes respectively and perfectly coincide for 5 randomly selected neurons (left) and synapses (middle). Each randomly selected neuron or synapse corresponds to one color.

where y denotes the target. In this case and according to the definition Eq. (3.1), the EP error processes for the parameters $\theta = \{w_{1,0}, w_{\hat{y},0}, w_1\}$ read:

$$\forall t \in [0, K] : \begin{cases} \Delta^{\text{EP}}_{w_1}(\beta, t) &= \frac{1}{\beta}\left(\sigma(\hat{y}^\beta_{t+1}) \cdot \sigma(s^{1,\beta}_{t+1})^\top - \sigma(\hat{y}^\beta_t) \cdot \sigma(s^{1,\beta}_t)^\top\right) \\ \Delta^{\text{EP}}_{w_{\hat{y},0}}(\beta, t) &= \frac{1}{\beta}\left(\hat{y}^\beta_{t+1}) \cdot \sigma(x)^\top - \sigma(\hat{y}^\beta_t) \cdot \sigma(x)^\top\right) \\ \Delta^{\text{EP}}_{w_{1,0}}(\beta, t) &= \frac{1}{\beta}\left(\sigma(s^{1,\beta}_{t+1}) \cdot \sigma(x)^\top - \sigma(s^{1,\beta}_t) \cdot \sigma(x)^\top\right), \end{cases}$$

Experiment. We took 5 output neurons, 50 hidden neurons and 10 visible neurons, using $\sigma(x) = \tanh(x)$. The data x is a dummy uniformly distributed random input $x \sim U[0, 1]$ (of size 1×10) and y is a dummy random one-hot encoded target (of size 1×5). We run the protocol described above with $\varepsilon = 0.08$, $T = 5000$ steps for the first phase, $K = 80$ steps and $\beta = 0.01$ for the second phase.

4.3.2 Fully connected architectures

In this subsection, we shall denote N the number of hidden layers, so that in general, $s^1, s^2, \cdots, s^N$ stand for the hidden layers and $s^{N+1} = \hat{y}$ is the output layer.

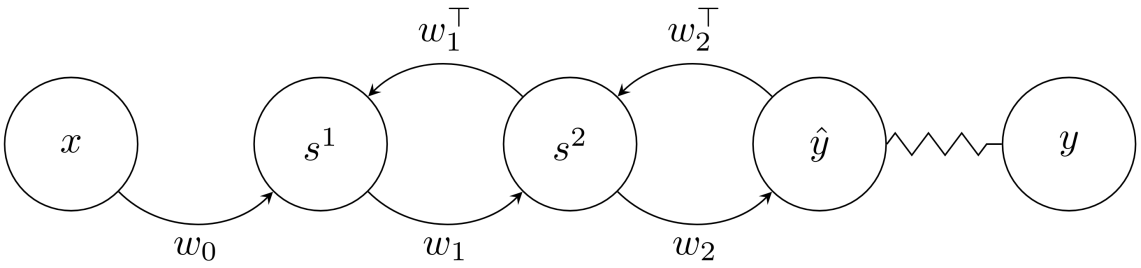

Figure 4.2: Fully connected layered architecture with $N = 2$ hidden layers.

Equations with $N = 2$. For this model, the primitive function is defined as:

$$\begin{aligned}\Phi\left(x, s^1, s^2, \hat{y}\right) = \frac{1}{2}(1-\varepsilon)\left(\|s^1\|^2 + \|s^1\|^2 + \|\hat{y}\|^2\right) \\ + \varepsilon\left(\sigma(s^1)^\top \cdot w_0 \cdot \sigma(x) + \sigma(s^2)^\top \cdot w_1 \cdot \sigma(s^1) + \sigma(\hat{y})^\top \cdot w_2 \cdot \sigma(s^2)\right)\end{aligned} \tag{4.7}$$

so that the equations of motion read:

$$\forall t \in [0,T] : \left\{\begin{array}{ll} \hat{y}_{t+1} & = (1-\varepsilon)\hat{y}_t + \varepsilon\sigma'(\hat{y}_t) \odot w_2 \cdot \sigma(s_t^2) \\ s_{t+1}^2 & = (1-\varepsilon)s_t^2 + \varepsilon\sigma'(s_t^2) \odot (w_1 \cdot \sigma\left(s_t^1\right) + w_2^\top \cdot \sigma(\hat{y}_t)) \\ s_{t+1}^1 & = (1-\varepsilon)s_t^1 + \varepsilon\sigma'(s_t^1) \odot (w_0 \cdot \sigma\left(x\right) + w_1^\top \cdot \sigma(s_t^2)) \end{array}\right.$$

In the second phase:

$$\forall t \in [0,K] : \left\{\begin{array}{ll} \hat{y}_{t+1}^\beta & = (1-\varepsilon)\hat{y}_t^\beta + \varepsilon\sigma'(\hat{y}_t^\beta) \odot w_2 \cdot \sigma(s_t^{2,\beta}) + \beta\varepsilon(y - \hat{y}_t^\beta) \\ s_{t+1}^{2,\beta} & = (1-\varepsilon)s_t^{2,\beta} + \varepsilon\sigma'(s_t^{2,\beta}) \odot (w_1 \cdot \sigma\left(s_t^{1,\beta}\right) + w_2^\top \cdot \sigma(\hat{y}_t^\beta)) \\ s_{t+1}^{1,\beta} & = (1-\varepsilon)s_t^{1,\beta} + \varepsilon\sigma'(s_t^{1,\beta}) \odot (w_0 \cdot \sigma\left(x\right) + w_1^\top \cdot \sigma(s_t^{2,\beta})) \end{array}\right.$$

In this case and according to the definition Eq. (3.1), the EP error processes for the parameters $\theta = \{w_0, w_1, w_2\}$ read :

$$\forall t \in [0,K] \left\{\begin{array}{l} \Delta_{w_0}^{\mathrm{EP}}(\beta,t) = \frac{1}{\beta}\left(\sigma\left(s_{t+1}^{1,\beta}\right) \cdot \sigma\left(x\right)^\top - \sigma\left(s_t^{1,\beta}\right) \cdot \sigma\left(x\right)^\top\right), \\ \Delta_{w_1}^{\mathrm{EP}}(\beta,t) = \frac{1}{\beta}\left(\sigma\left(s_{t+1}^{2,\beta}\right) \cdot \sigma\left(s_{t+1}^{1,\beta^\top}\right) - \sigma\left(s_t^{2,\beta}\right) \cdot \sigma\left(s_t^{1,\beta^\top}\right)\right), \\ \Delta_{w_2}^{\mathrm{EP}}(\beta,t) = \frac{1}{\beta}\left(\sigma\left(\hat{y}_{t+1}^{\beta}\right) \cdot \sigma\left(s_{t+1}^{2,\beta^\top}\right) - \sigma\left(\hat{y}_t^{\beta}\right) \cdot \sigma\left(s_t^{2,\beta^\top}\right)\right). \end{array}\right. \tag{4.8}$$

Simplifying the equations with $N = 2$. To go from our multi-layered architecture to the more general model presented in the previous section, we define the state s of the network as the concatenation of all the layers' states, i.e. $s = (s^2, s^1, s^0)^\top$ and we define the weight matrices W and W_x as:

$$W = \begin{pmatrix} 0 & w_1^\top & 0 \\ w_1 & 0 & w_2^\top \\ 0 & w_2 & 0 \end{pmatrix}, \qquad W_x = \begin{pmatrix} w_0 \\ 0 \\ 0 \end{pmatrix}, \tag{4.9}$$

so that the primitive function Φ defined in Eq. (4.7) rewrites as Eq. (4.1).

Generalizing the equations for any N. For this model, the primitive function is defined as:

$$\Phi\left(x, s^1, s^2, \dots, s^{N+1} = \hat{y}\right) = \frac{1}{2}(1-\varepsilon)\left(\sum_{n=1}^{N+1} ||s^n||^2\right) + \varepsilon \sum_{n=1}^{N} \sigma(s^{n+1})^\top \cdot w_n \cdot \sigma(s^n) + \sigma(s^1) \cdot w_0 \cdot \sigma(x) \tag{4.10}$$

so that the equations of motion read:

$$\forall t \in [0,T] : \begin{cases} \hat{y}_{t+1} &= (1-\varepsilon)\hat{y}_t + \varepsilon\sigma'(\hat{y}_t) \odot w_N \cdot \sigma(s_t^N) \\ s_{t+1}^n &= (1-\varepsilon)s_t^n + \sigma'(s_t^n) \odot \varepsilon(w_{n-1} \cdot \sigma\left(s_t^{n-1}\right) + w_n^\top \cdot \sigma(s_t^{n+1})) \qquad \forall n \in [2, N] \\ s_{t+1}^1 &= (1-\varepsilon)s_t^1 + \varepsilon\sigma'(s_t^1) \odot (w_0 \cdot \sigma\left(x\right) + w_1^\top \cdot \sigma(s_t^2)) \end{cases},$$

and in the second phase:

$$\forall t \in [0,T] : \begin{cases} \hat{y}_{t+1}^\beta &= (1-\varepsilon)\hat{y}_t^\beta + \varepsilon\sigma'(\hat{y}_t^\beta) \odot w_N \cdot \sigma(s_t^{N,\beta}) + \beta\varepsilon(y - \hat{y}_t^\beta) \\ s_{t+1}^{n,\beta} &= (1-\varepsilon)s_t^{n,\beta} + \sigma'(s_t^{n,\beta}) \odot \varepsilon(w_{n-1} \cdot \sigma\left(s_t^{n-1,\beta}\right) + w_n^\top \cdot \sigma(s_t^{n+1,\beta})) \\ & \forall n \in [2, N], \\ s_{t+1}^{1,\beta} &= (1-\varepsilon)s_t^{1,\beta} + \varepsilon\sigma'(s_t^{1,\beta}) \odot (w_0 \cdot \sigma\left(x\right) + w_1^\top \cdot \sigma(s_t^{2,\beta})) \end{cases}$$

According to Eq. (3.1) again, we have:

$$\begin{cases} \Delta_{w_0}^{\mathrm{EP}}(\beta, t) &= \frac{1}{\beta}\left(\sigma\left(s_{t+1}^{1,\beta}\right) \cdot \sigma\left(x\right)^\top - \sigma\left(s_t^{1,\beta}\right) \cdot \sigma\left(x\right)^\top\right) \\ \Delta_{w_n}^{\mathrm{EP}}(\beta, t) &= \frac{1}{\beta}\left(\sigma\left(s_{t+1}^{n+1,\beta}\right) \cdot \sigma\left(s_{t+1}^{n,\beta}\right)^\top - \sigma\left(s_t^{n+1,\beta}\right) \cdot \sigma\left(s_t^{n,\beta}\right)^\top\right) \qquad \forall n \in [1, N] \end{cases}$$

Defining $s = (s^1, s^2, \dots, \hat{y})^\top$ and:

$$W = \begin{pmatrix} 0 & w_1^\top & 0 & 0 & 0 \\ w_1 & 0 & w_2^\top & 0 & 0 \\ 0 & w_2 & 0 & \ddots & 0 \\ 0 & 0 & \ddots & 0 & w_n^\top \\ 0 & 0 & 0 & w_n & 0 \end{pmatrix}, \qquad W_x = \begin{pmatrix} w_0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \tag{4.11}$$

the primitive function Φ defined in Eq. (4.10) again writes as Eq. (4.1).

Experiment. We consider architectures of the kind 784-512-...-512-10 where we have 784 input neurons, 10 output neurons, and each hidden layer has 512 neurons, using $\sigma(x) = \tanh(x)$. The data x is a random MNIST sample (of size 1×784) and y is the associated target (of size 1×10). We have ran experiments for $\varepsilon = 0.08$. The values of T, K and β depend on the depth of the architecture considered and can be found in Table 2.1.

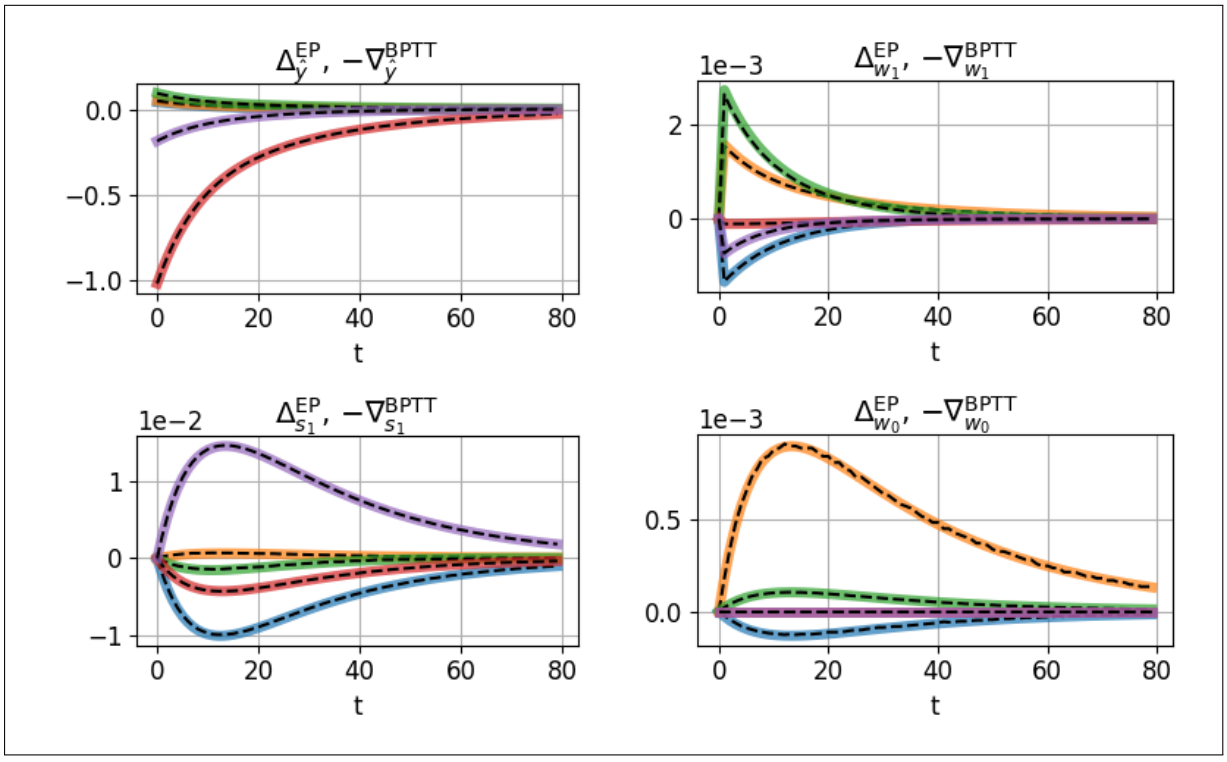

Figure 4.3: Real-time RNN with symmetric weights with one hidden layer. **Left**: $\Delta_s^{\text{EP}}(t)$ neural updates and $-\nabla_s^{\text{BPTT}}(t)$ gradients. **Right**: $\Delta_\theta^{\text{EP}}(t)$ weight updates and $-\nabla_\theta^{\text{BPTT}}(t)$ gradients.

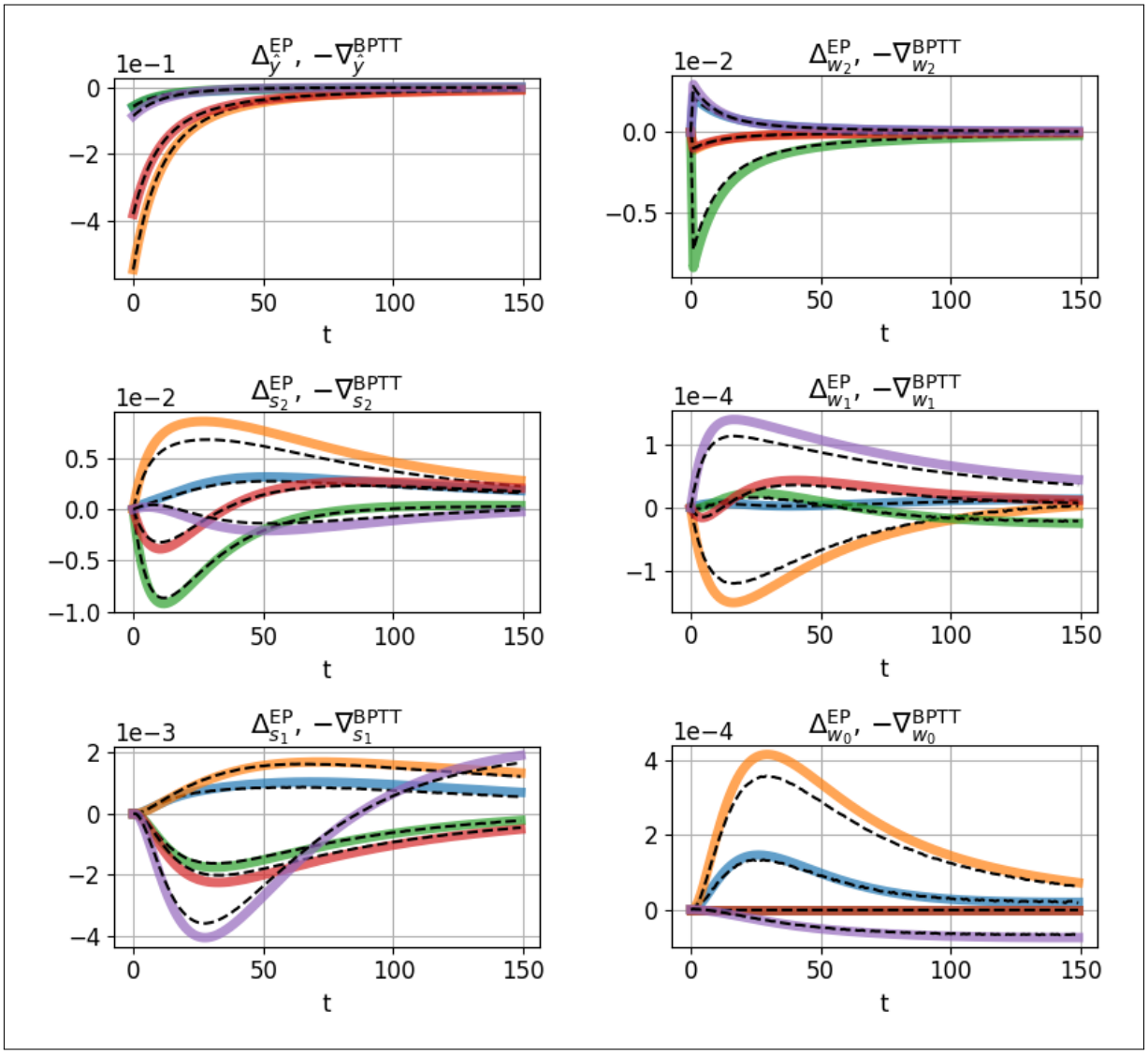

Figure 4.4: Real-time RNN with symmetric weights with two hidden layers. **Left**: $\Delta_s^{\mathrm{EP}}(t)$ neural updates and $-\nabla_s^{\mathrm{BPTT}}(t)$ gradients. **Right**: $\Delta_\theta^{\mathrm{EP}}(t)$ weight updates and $-\nabla_\theta^{\mathrm{BPTT}}(t)$ gradients.

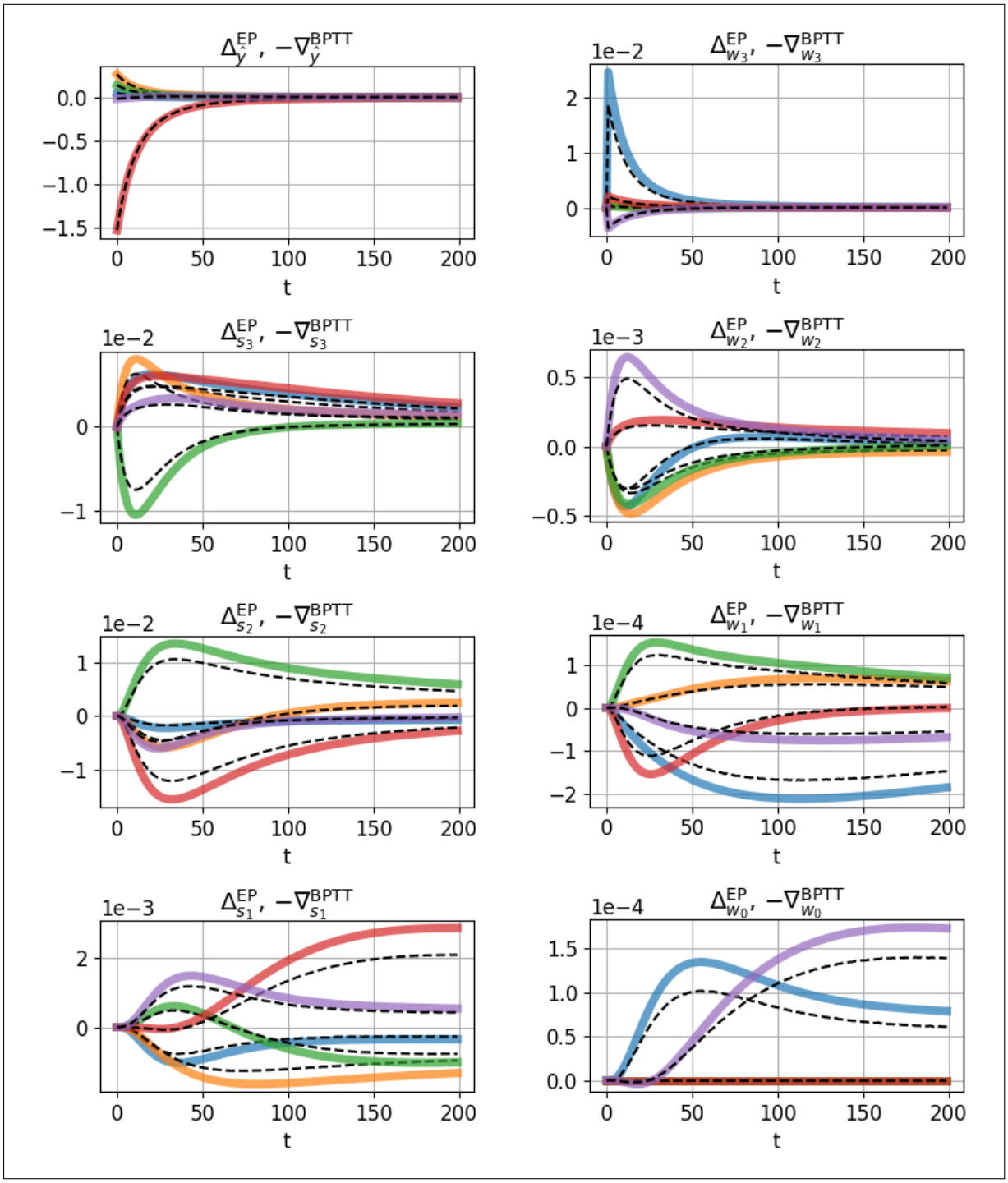

Figure 4.5: Real-time RNN with symmetric weights with three hidden layers. **Left**: $\Delta_s^{\mathrm{EP}}(t)$ neural updates and $-\nabla_s^{\mathrm{BPTT}}(t)$ gradients. **Right**: $\Delta_\theta^{\mathrm{EP}}(t)$ weight updates and $-\nabla_\theta^{\mathrm{BPTT}}(t)$ gradients.

4.4 Discrete-time RNNs in the prototypical setting

4.4.1 Fully connected architecture

Equations with $N = 2$. We consider again the layered architecture of Fig. 4.2. In the discrete-time setting of EP, the dynamics of the first phase are defined as:

$$\forall t \in [0, T] : \begin{cases} \hat{y}_{t+1} = \sigma\left(w_2 \cdot s_t^2\right), \\ s_{t+1}^2 = \sigma\left(w_1 \cdot s_t^1 + w_2^\top \cdot \hat{y}_t\right), \\ s_{t+1}^1 = \sigma\left(w_0 \cdot x + w_1^\top \cdot s_t^2\right). \end{cases} \tag{4.12}$$

In the second phase, the dynamics read:

$$\forall t \in [0, K] : \begin{cases} \hat{y}_{t+1}^\beta = \sigma\left(w_2 \cdot s_t^{2,\beta}\right) + \beta(y - \hat{y}_t^\beta), \\ s_{t+1}^{2,\beta} = \sigma\left(w_1 \cdot s_t^{1,\beta} + w_2^\top \cdot \hat{y}_t^\beta\right), \\ s_{t+1}^{1,\beta} = \sigma\left(w_0 \cdot x + w_1^\top \cdot s_t^{2,\beta}\right). \end{cases} \tag{4.13}$$

As usual, y denotes the target. Consider the function:

$$\Phi\left(x, s^1 s^2, \hat{y}\right) = \hat{y}^\top \cdot w_2 \cdot s^2 + {s^2}^\top \cdot w_1 \cdot s^1 + {s^1}^\top \cdot w_0 \cdot x. \tag{4.14}$$

We can compute, for example:

$$\frac{\partial \Phi}{\partial s^2} = w_1 \cdot s^1 + w_2^\top \cdot \hat{y}. \tag{4.15}$$

Comparing Eq. (4.12) and Eq. (4.15), and ignoring the activation function σ, we can see that

$$s_t^2 \approx \frac{\partial \Phi}{\partial s^2}\left(x, s_{t-1}^1, s_{t-1}^2, \hat{y}_{t-1}\right). \tag{4.16}$$

And similarly for the layers $\hat{y}$ and s^1.

According to the definition of $\Delta_\theta^{\text{EP}}$ in Eq. (3.1), for every layer and every $t \in [0, K]$:

$$\begin{cases} \Delta_{w_0}^{\text{EP}}(\beta, t) = \frac{1}{\beta}\left(s_{t+1}^{1,\beta} \cdot x^\top - s_t^{1,\beta} \cdot x^\top\right), \\ \Delta_{w_1}^{\text{EP}}(\beta, t) = \frac{1}{\beta}\left(s_{t+1}^{2,\beta} \cdot s_{t+1}^{1,\beta^\top} - s_t^{2,\beta} \cdot s_t^{1,\beta^\top}\right), \\ \Delta_{w_2}^{\text{EP}}(\beta, t) = \frac{1}{\beta}\left(\hat{y}_{t+1}^\beta \cdot s_{t+1}^{2,\beta^\top} - \hat{y}_t^\beta \cdot s_t^{2,\beta^\top}\right). \end{cases} \tag{4.17}$$

Simplifying the equations with $N = 2$. Again, we define the state s of the network as the concatenation of all the layers' states, i.e. $s = (s^2, s^1, s^0)^\top$ and we define the weight matrices W and W_x as in Eq. (4.9) so that Eq. (4.12) and Eq. (4.14) can be vectorized into:

$$s_{t+1} = \sigma(W \cdot s_t + W_x \cdot x), \tag{4.18}$$

$$\Phi = \frac{1}{2} s^T \cdot W \cdot s + \frac{1}{2} s^T \cdot W_x \cdot x. \tag{4.19}$$

Generalizing the equations for any N. For a general architecture with a given N, the dynamics of the first phase are defined as:

$$\forall t \in [0, T] : \begin{cases} \hat{y}_{t+1} = \sigma\left(w_N \cdot s_t^N\right) \\ s_{t+1}^n = \sigma\left(w_{n-1} \cdot s_t^{n-1} + w_n^\top \cdot s_t^{n+1}\right) \qquad \forall n \in [2, N] \\ s_{t+1}^1 = \sigma\left(w_0 \cdot x + w_1^\top \cdot s_t^2\right), \end{cases} \tag{4.20}$$

and those of the second phase as:

$$\forall t \in [0, T] : \begin{cases} \hat{y}_{t+1}^\beta = \sigma\left(w_N \cdot s_t^{N,\beta}\right) + \beta(y - \hat{y}_t^\beta) \\ s_{t+1}^{n,\beta} = \sigma\left(w_{n-1} \cdot s_t^{n-1,\beta} + w_n^\top \cdot s_t^{n+1,\beta}\right) \qquad \forall n \in [2, N] \\ s_{t+1}^{1,\beta} = \sigma\left(w_0 \cdot x + w_1^\top \cdot s_t^{2,\beta}\right), \end{cases} \tag{4.21}$$

where y denotes the target. Defining:

$$\Phi(x, s^1, \ldots, \hat{y}) = \sum_{n=1}^{N} {s^{n+1}}^\top \cdot w_n \cdot s^n + s^1 \cdot w_0 \cdot x, \tag{4.22}$$

ignoring the activation function σ, Eq. (4.20) rewrites:

$$s_{t+1}^n \approx \frac{\partial \Phi}{\partial s^n}(x, s^1, \ldots, \hat{y}) \qquad \forall n \in [1, N+1] \tag{4.23}$$

According to Eq. (3.1), for every layer w_n and every $t \in [0, K]$:

$$\begin{cases} \Delta_{w_0}^{\mathrm{EP}}(\beta, t) = \frac{1}{\beta}\left(s_{t+1}^{1,\beta} \cdot x^\top - s_t^{1,\beta} \cdot x^\top\right), \\ \Delta_{w_n}^{\mathrm{EP}}(\beta, t) = \frac{1}{\beta}\left(s_{t+1}^{n+1,\beta} \cdot {s_{t+1}^{n,\beta}}^\top - s_t^{n+1,\beta} \cdot {s_t^{n,\beta}}^\top\right) \qquad \forall n \in [1, N] \end{cases} \tag{4.24}$$

Defining $s = (s^1, s^2, \ldots, \hat{y})^\top$ and taking again W and W_x as defined in Eq. (4.11), Eq. (4.20) and Eq. (4.22) can also be vectorized into:

$$s_{t+1} = \sigma(W \cdot s_t + W_x \cdot x) \tag{4.25}$$

$$\Phi(x, s, W, W_x) = \frac{1}{2} s^T \cdot W \cdot s + \frac{1}{2} s^T \cdot W_x \cdot x \tag{4.26}$$

Experiment. We have considered the same architectures as the ones used for real-time RNNs and on the same data (MNIST). The values of T, K and β depend on the depth of the architecture considered and can be found in Table 2.1.

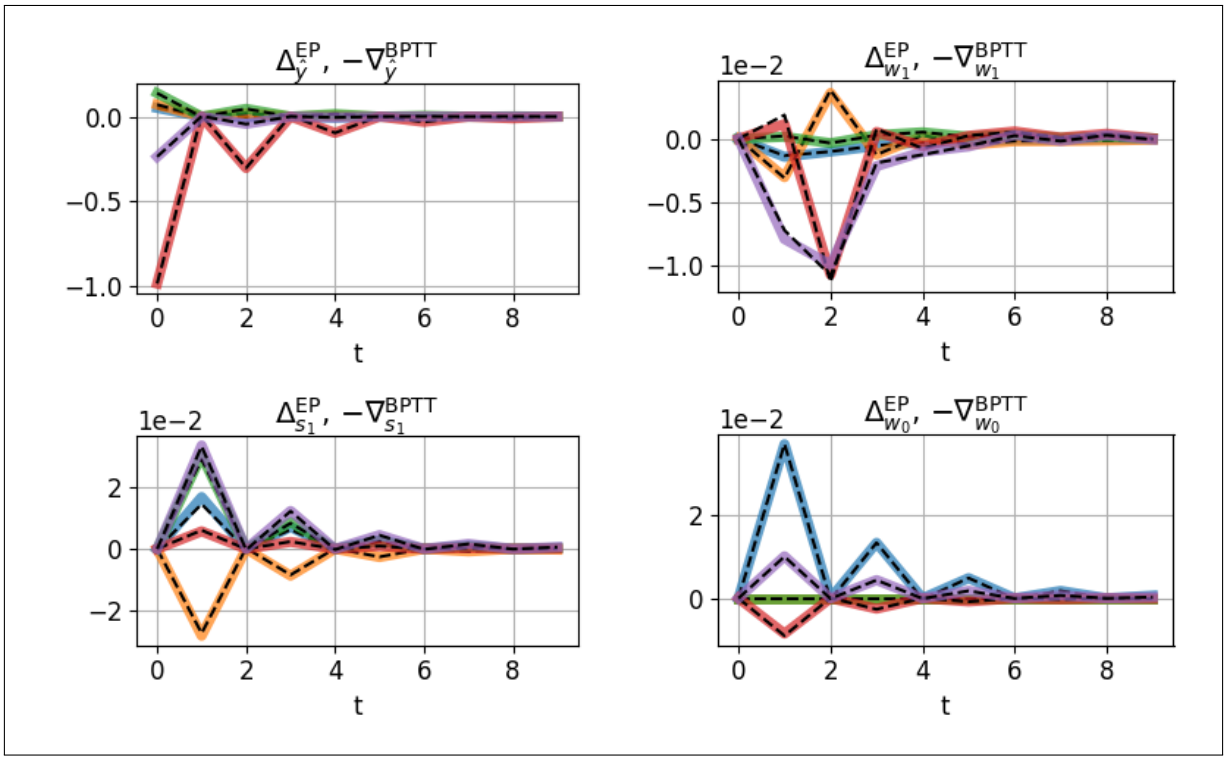

Figure 4.6: Discrete-time RNN with symmetric weights with one hidden layer. **Left**: $\Delta_s^{\mathrm{EP}}(t)$ neural updates and $-\nabla_s^{\mathrm{BPTT}}(t)$ gradients. **Right**: $\Delta_\theta^{\mathrm{EP}}(t)$ weight updates and $-\nabla_\theta^{\mathrm{BPTT}}(t)$ gradients.

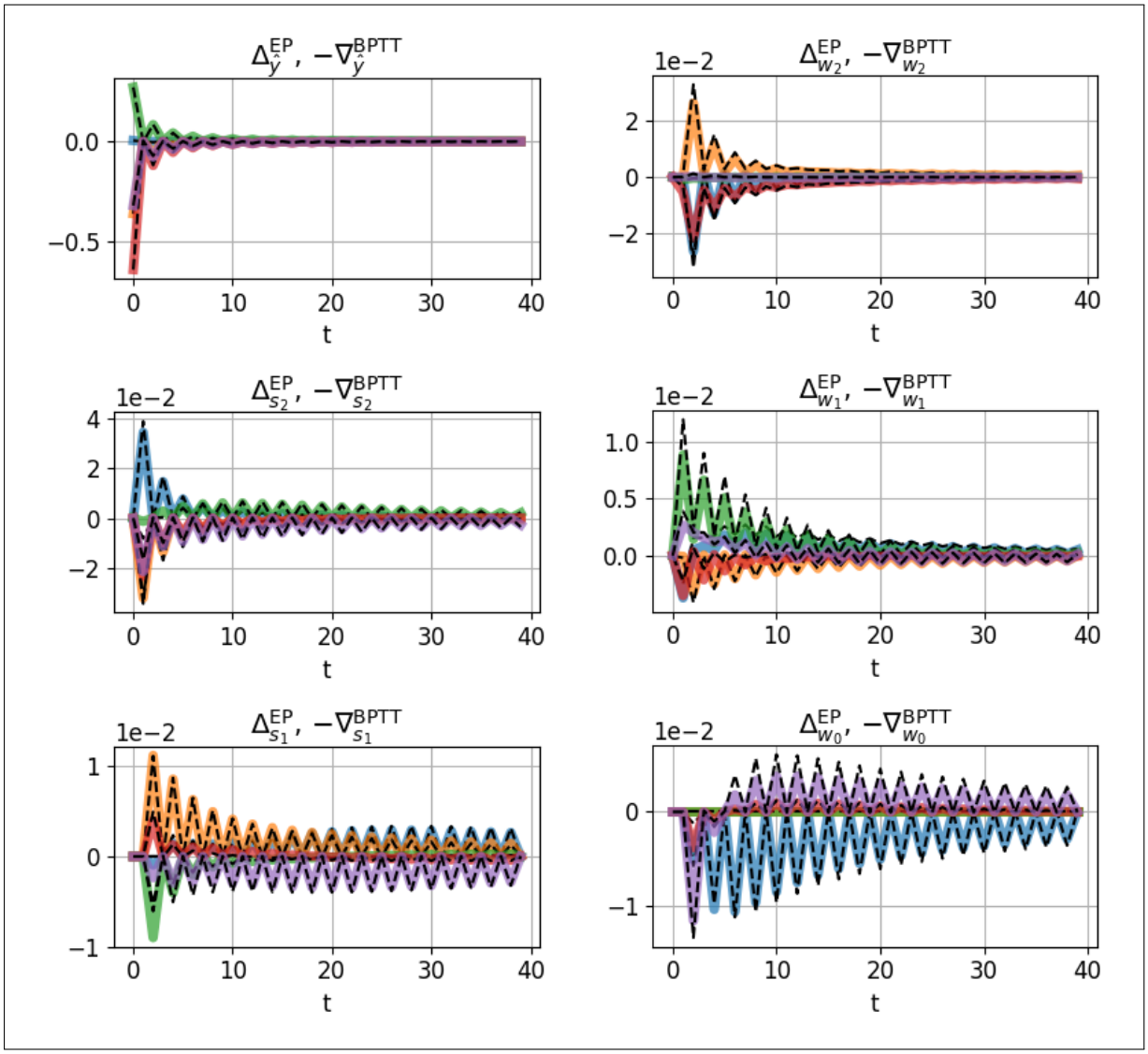

Figure 4.7: Discrete-time RNN with symmetric weights with two hidden layers. **Left**: $\Delta_s^{\mathrm{EP}}(t)$ neural updates and $-\nabla_s^{\mathrm{BPTT}}(t)$ gradients. **Right**: $\Delta_\theta^{\mathrm{EP}}(t)$ weight updates and $-\nabla_\theta^{\mathrm{BPTT}}(t)$ gradients.

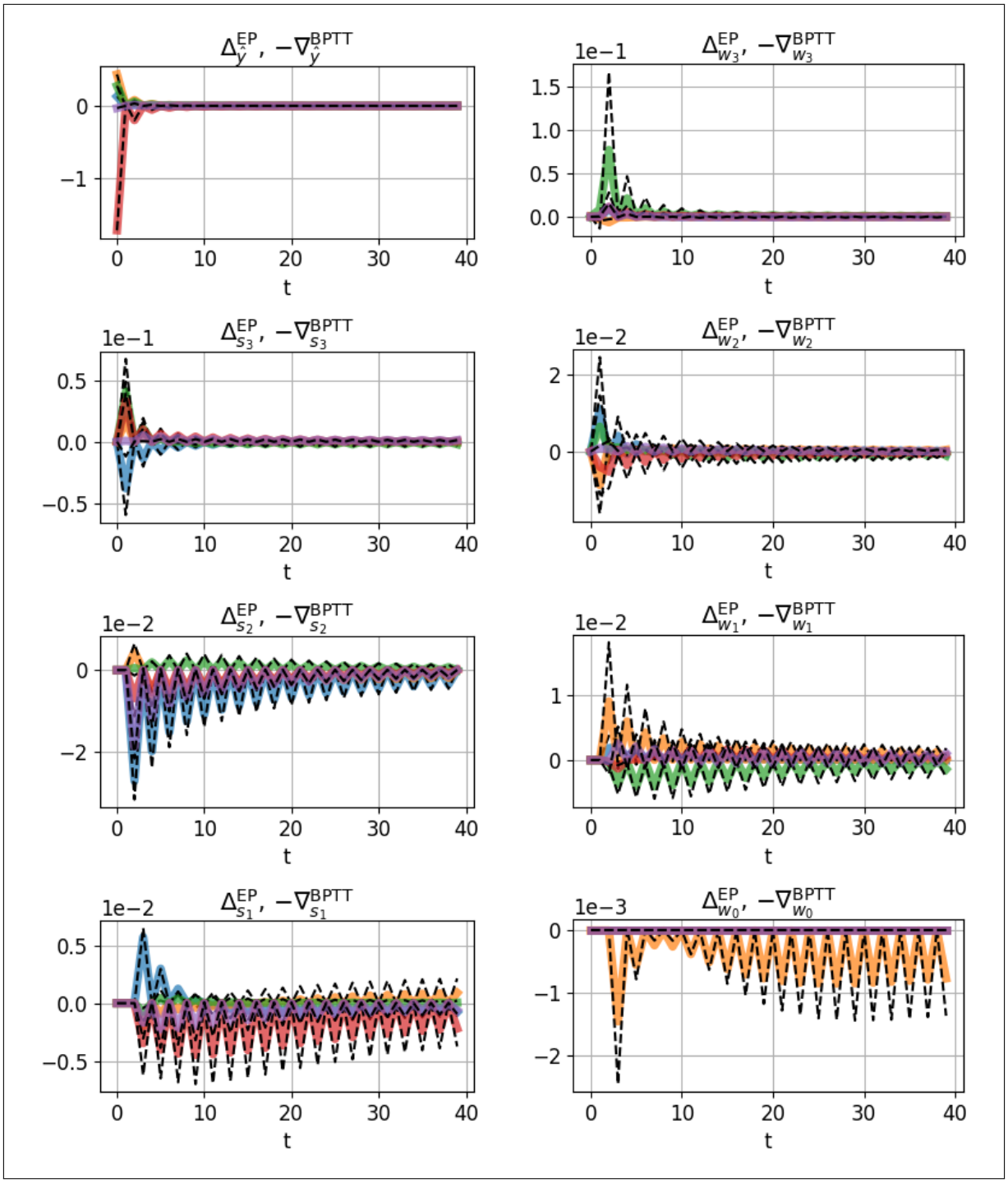

Figure 4.8: Discrete-time RNN with symmetric weights with three hidden layers. **Left**: $\Delta_s^{\mathrm{EP}}(t)$ neural updates and $-\nabla_s^{\mathrm{BPTT}}(t)$ gradients. **Right**: $\Delta_\theta^{\mathrm{EP}}(t)$ weight updates and $-\nabla_\theta^{\mathrm{BPTT}}(t)$ gradients.

4.4.2 Convolutional architecture

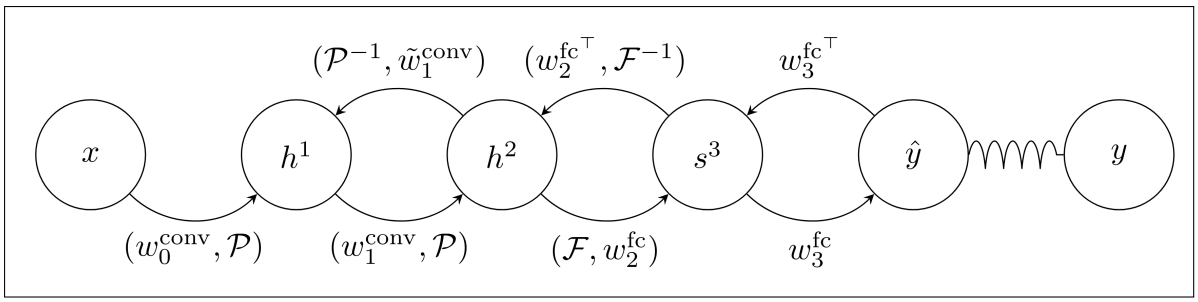

Figure 4.9: **Convolutional architecture**. Summary of the operations, notations and conventions adopted in this subsection.

Architecture. The model is a layered architecture composed of a fully connected part and a convolutional part. We therefore distinguish between the flat layers (i.e. those of the fully connected part) and the convolutional layers (i.e. those of the convolutional part). We denote n_{fc} and n_{conv} the number of flat (hidden) layers and of convolutional layers respectively.

As previously, layers are labelled in a forward fashion, but we differentiate convolutional layers (h) from fully connected layers (s). h^1 labels the first convolutional layer, h^2 the second one, $h^{n_{\text{conv}}}$ the last one. Convolutional layers, also called *feature maps*, are three-dimensional *, i.e. $h_{c,i,j}$ where c labels a channel, i and j label one pixel of this feature map. A convolutional layer h_n is deduced from an *upstream* convolutional layer h_{n-1} by the composition of a convolution and a pooling operation, which we shall respectively denote by $\star$ and $\mathcal{P}$. Conversely, a convolutional layer h_n is deduced from a *downstream* convolutional layer h_{n+1} by the composition of a unpooling operation and of a transpose convolution.

After flattening $h^{n_{\text{conv}}}$, the next layer is the first fully connected layer and is denoted $s^{n_{\text{conv}}+1}$. The second fully connected layer is denoted $s^{n_{\text{conv}}+2}$ and the last (output) layer is $\hat{y} = s^{n_{\text{conv}}+n_{\text{fc}}+1}$. Fully connected layers are bi-dimensional†, i.e. $s_{i,j}$ where i and j label one pixel.

We note w^{fc} and w^{conv} the fully connected weights and the convolutional filters respectively, so that w^{fc} is a two-order tensor and w^{conv} is a four order tensor, i.e. $w^{\text{conv}}_{c_{out},c_{in},i,j}$ is the element (i,j) of the filter connecting the channel c_{in} of the input feature map to the channel c_{out} of the output feature map. We denote the filter size by F. We keep the same notation x for the input data. Fig. 4.9 summarizes the whole architecture.

Definition of the operations. In this paragraph, we define all the operations involved in the definition and the properties of our convolutional model:

*Four-dimensional in pratice, considering the mini-batch dimension.

†Three-dimensional in practice, considering the mini-batch dimension.

- the *convolution* of a filter w of size F with C_{out} output channels and C_{in} input channels by a vector X as:

$$(w \star X)_{c_{\text{out}},i,j} := \sum_{c_{\text{int}}=1}^{C_{\text{in}}} \sum_{r,s=1}^{F} w_{c_{\text{out}},c_{\text{in}},r,s} X_{c_{\text{in}},i+r-1,j+s-1}\cdot, \tag{4.27}$$

- the associated *transpose convolution* is defined as the convolution of kernel $\tilde{W}$ (also called "flipped kernel"):

$$\tilde{w}_{c_{in},c_{out},r,s} = w_{c_{out},c_{in},F-r+1,F-s+1}, \tag{4.28}$$

  with an input padded with $\tilde{P} = F - 1 - P$ where P denotes the padding applied in the forward convolution: in this way transpose convolution recovers the original input size before convolution. Whenever $\tilde{w}$ is applied on a vector, we shall implicitly assume this padding. The transpose convolution can be seen as the gradient of the associated forward convolution with respect to its input - see Eq. (4.39) of Lemma 5. Fig. 4.10 provides a simple sketch of convolution and transpose convolution.

- We define the *general dot product* between two vectors X^1 and X^2 as:

$$X^1 \bullet X^2 = \sum_{c_{\text{in}}=1}^{C_{\text{in}}} \sum_{i,j=1}^{d} X^1_{c_{\text{in}},i,j} X^2_{c_{\text{in}},i,j}. \tag{4.29}$$

- We define the *pooling* operation with filter size F and stride F as:

$$\mathcal{P}(X)_{c,i,j} = \max_{r,s\in[1,F]} \left\{ X_{c,F(i-1)+r,F(j-1)+s} \right\}. \tag{4.30}$$

  We also introduce the relative indices within a pooling zone for which the maximum is reached as:

$$\text{ind}(X)_{c,i,j} =_{r,s\in[1,F]} \left\{ X_{c,F(i-1)+r,F(j-1)+s} \right\} = (r^*(X,c,i), s^*(X,c,j)). \tag{4.31}$$

- We define the *inverse pooling* operation as:

$$\mathcal{P}^{-1}(Y, \text{ind}(X))_{c,p,q} = \begin{cases} Y_{c,\lceil p/F\rceil,\lceil q/F\rceil} \text{if} & p = F(\lceil p/F\rceil - 1) + r^*(X,c,\lceil p/F\rceil), \\ & q = F(\lceil q/F\rceil - 1) + s^*(X,c,\lceil q/F\rceil) \\ 0 & \text{otherwise} \end{cases} \tag{4.32}$$

  Putting Eq. (4.32) into words, the inverse pooling operation applied to a vector Y given the indices of another vector X up-samples Y to a vector of the same size of X with the elements of Y placed at the location of the maximal elements of X within each pooling zone, and zero elsewhere. Note that Eq. (4.32) can be written more conveniently as:

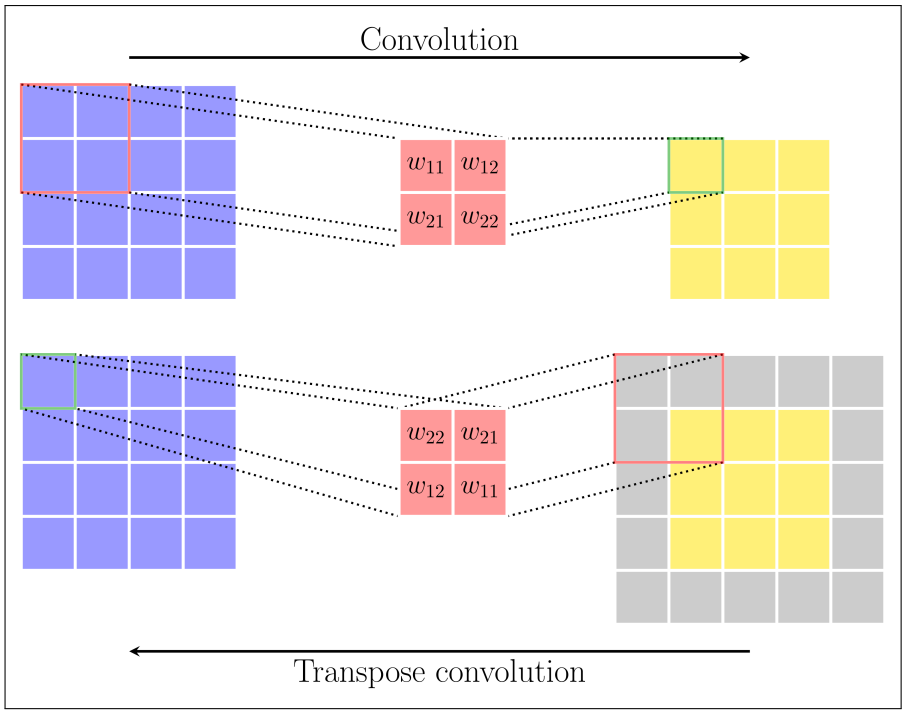

Figure 4.10: **Convolution and transpose convolution**. The convolution operation depicted uses $F = 2$ (kernel size), $P = 0$ (padding), $S = 1$ (stride). The input of the convolution is in blue, the kernel in red and the output of the convolution in yellow. With an input length of $L^{\text{in}} = 4$, the output length is $L^{\text{out}} = \frac{L^{\text{in}} - F + 2P}{S} + 1 = 3$. The transpose convolution uses the flipped kernel (deduced from the convolution kernel by flipping rows and columns). In order to ensure that the output of the transpose convolution is of the same size as the original input ($L^{\text{in}} = 4$), the input of the transpose convolution should be padded with zeros, with $\tilde{P} = F - 1 - P = 1$ (gray squares). The length of the output of the transpose convolution thereby is $\frac{L^{\text{out}} - F + 2\tilde{P}}{S} + 1 = 4$.

$$\mathcal{P}^{-1}(Y, \text{ind}(X))_{c,p,q} = \sum_{i,j} Y_{c,i,j} \cdot \delta_{p,F(i-1)+r^*(X,i)} \cdot \delta_{q,F(j-1)+s^*(X,j)}. \tag{4.33}$$

Similarly to the transpose convolution, the inverse pooling can be seen as the gradient of pooling with respect to its input - see Eq. (4.38) of Lemma 5. Fig. 4.11 provides a simple sketch of pooling and inverse pooling.

- The *flattening* operation which maps a vector X into its flattened shape, i.e. $\mathcal{F} : C^{\text{in}} \times d \times d \to 1 \times C^{\text{in}} D^2$. We denote its inverse operation, i.e. the *inverse flattening operation* as $\mathcal{F}^{-1}$.

Equations for $n_{\text{conv}} = 2$ and $n_{\text{fc}} = 0$. We first consider a simple example where the input image is convolved and pooled twice, then directly flattened and fed into the output layer.

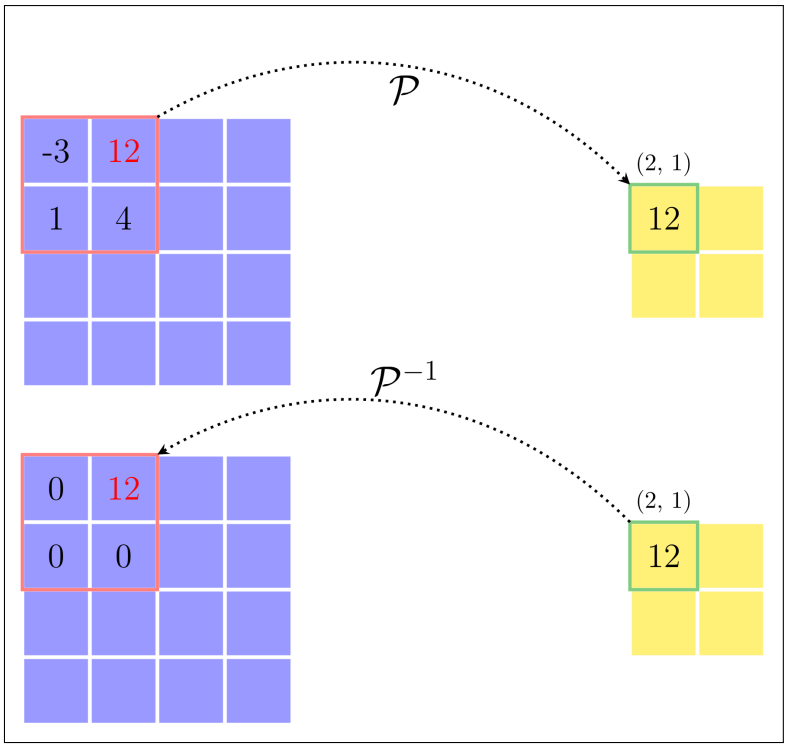

Figure 4.11: **Pooling and inverse pooling**. The pooling operation depicted uses $F = 2$ (kernel size), $P = 0$ (padding), $S = 2$ (stride). The input of the pooling is in blue and its output in yellow. With an input length of $L^{\text{in}} = 4$, the output length is $L^{\text{out}} = \frac{L^{\text{in}} - F + 2P}{S} + 1 = 2$. Note that upon pooling within a zone (red frame), the relative indices of the maximum element (with respect to the upper left corner) are retained for the inverse operation. Inverse pooling therefore simply amounts to reconstruct an input, with the maximum elements located at their initial position, and putting zeros elsewhere.

This is the architecture which was used to process the MNIST data set in the experiments. For concreteness, let us assume here the same hyperparameters.

The first convolutional layer uses a 5×5 ($F_{\text{conv}} = 5$) kernel with 32 features maps, stride 1 ($S = 1$) and pooling 0 ($P = 0$), the second convolutional layer uses a 5×5 kernel with 64 feature maps, stride 1 and pooling 0 as well. Pooling is achieved with 2×2 filters ($F_{\text{pool}} = 2$) and stride 2. With the notations we have introduced before, h^1 and h^2 are respectively the first two convolutional layers, and $s^3 = \hat{y}$ is the output layer. We denote d^1, d^2 and d^3 the length of h^1, h^2 and $\hat{y}$ respectively so that their dimension is $c^1 \times d^1 \times d^1$, $c^2 \times d^2 \times d^2$ and $1 \times d^3$ respectively *. In our case, we have $c^1 = 32$, $c^2 = 64$. Taking $\dim(x) = 1 \times 28 \times 28$ (MNIST samples), the length of the output of the first convolution is $L^1 = \frac{d^0 - F_{\text{conv}} + 2P}{S} + 1 = 24$. After pooling, we get h^1 of length $d^1 = \frac{L^1 - F_{\text{pool}}}{2} + 1 = 12$ so that $\dim(h^1) = 32 \times 12 \times 12$. Similarly, $\dim(h^2) = 64 \times 4 \times 4$. After flattening, we get $\mathcal{F}(h^1)$ of dimension $1 \times (64 \cdot 4 \cdot 4) = 1 \times 1024$.

*We omit here the batch dimension for simplicity.

Finally, as usual for MNIST classification, $\dim(\hat{y}) = 1 \times 10$.

Also as pointed out before, note that for a convolution of filter size F and padding P, padding $\tilde{P} = F-1-P$ should be applied to the input of the associated transpose convolution to retrieve the dimension of the initial input. Concretely, the output of the first convolution has dimension $32 \times 24 \times 24$ has shown above ($L^1 = 24$ with the previous notations). Using the same formula as before with padding $\tilde{P} = F - 1 - P = 4$, we obtain the length of the output of the deconvolution by the tilted filter $\tilde{w}_0$ as $L^{\text{back}} = \frac{L^1 - F_{\text{conv}} + 2\tilde{P}}{S} + 1 = 28$, so that we get the initial input dimension back. This technical point is addressed more formally in the demonstration of Eq. (4.39) of Lemma 5.

With these notations, the system is driven by the following dynamics during the first phase:

$$\forall t \in [0,T] : \left\{ \begin{array}{ll} \hat{y}_{t+1} & = \sigma\left(w_2^{\text{fc}} \cdot \mathcal{F}\left(h_t^2\right)\right) \\ h_{t+1}^2 & = \sigma\left(\mathcal{P}\left(w_1^{\text{conv}} \star h_t^1\right) + \mathcal{F}^{-1}\left(w_2^{\text{fc}^\top} \cdot \hat{y}_t\right)\right) \\ h_{t+1}^1 & = \sigma\left(\mathcal{P}\left(w_0^{\text{conv}} \star x\right) + \tilde{w}_2^{\text{conv}} \star \mathcal{P}^{-1}\left(h_t^2, \text{ind}(w_1^{\text{conv}} \star h_{t-1}^1)\right)\right) \end{array} \right. .$$

Note that to unpool h^2 at time step $t+1$, we need to store the indices of the maximum elements within each pooling window of $w_1^{\text{conv}} * h^1$ upon pooling at time step t. During the second phase, the system dynamics read:

$$\forall t \in [0,K] : \left\{ \begin{array}{ll} \hat{y}_{t+1}^\beta & = \sigma\left(w_2^{\text{fc}} \cdot \mathcal{F}\left(h_t^{2,\beta}\right)\right) + \beta\left(y - \hat{y}_t^\beta\right) \\ h_{t+1}^{2,\beta} & = \sigma\left(\mathcal{P}\left(w_1^{\text{conv}} \star h_t^{1,\beta}\right) + \mathcal{F}^{-1}\left(w_2^{\text{fc}^\top} \cdot \hat{y}_t^\beta\right)\right) \\ h_{t+1}^{1,\beta} & = \sigma\left(\mathcal{P}\left(w_0^{\text{conv},} \star x\right) + \tilde{w}_2^{\text{conv}} \star \mathcal{P}^{-1}\left(h_t^{2,\beta}, \text{ind}(w_1^{\text{conv}} \star h_{t-1}^{1,\beta})\right)\right) \end{array} \right. .$$

Considering the function:

$$\Phi(x, h^1, h^2, \hat{y}) = \hat{y} \cdot w_2^{\text{fc}} \cdot \mathcal{F}(h_t^2) + h^2 \bullet \mathcal{P}\left(w_1^{\text{conv}} \star h^1\right) + h^1 \bullet \mathcal{P}\left(w_1^{\text{conv}} \star x\right),$$

and ignoring the activation function, we have:

$$\left\{ \begin{array}{l} h_t^1 \approx \dfrac{\partial \Phi}{\partial h^2} \\ h_t^2 \approx \dfrac{\partial \Phi}{\partial h^2} \\ \hat{y} \approx \dfrac{\partial \Phi}{\partial \hat{y}} \end{array} \right. , \tag{4.34}$$

so that in this case, we define the EP error processes for the parameters $\theta = \{w_0^{\text{conv}}, w_1^{\text{conv}}, w_2^{\text{fc}}\}$ as, $\forall t \in [0, K]$:

$$\begin{cases} \Delta^{\text{EP}}_{w_2^{\text{fc}}}(t) = \frac{1}{\beta}\left(\hat{y}^{\beta}_{t+1} \cdot \mathcal{F}\left(h^{2,\beta}_{t+1}\right)^{\top} - \hat{y}^{\beta}_{t} \cdot \mathcal{F}\left(h^{2,\beta}_{t}\right)^{\top}\right) \\ \Delta^{\text{EP}}_{w_1^{\text{conv}}}(t) = \frac{1}{\beta}\left(\mathcal{P}^{-1}(h^{2,\beta}_{t+1}) \star h^{1,\beta}_{t+1} - \mathcal{P}^{-1}(h^{2,\beta}_{t}) \star h^{1,\beta}_{t}\right) \\ \Delta^{\text{EP}}_{w_0^{\text{conv}}}(t) = \frac{1}{\beta}\left(\mathcal{P}^{-1}(h^{1,\beta}_{t+1}) \star x - \mathcal{P}^{-1}(h^{1,\beta}_{t}) \star x\right) \end{cases} , \tag{4.35}$$

Equations for any number of layers. The equations in the fully connected layers read in the first phase:

$$\forall t \in [0,T] : \begin{cases} \hat{y}_{t+1} & = \sigma\left(w^{\text{fc}}_{n_{\text{conv}}+n_{\text{fc}}} \cdot s_t^{n_{\text{conv}}+n_{\text{fc}}}\right) \\ s^{n}_{t+1} & = \sigma\left(w^{\text{fc}}_{n-1} \cdot s_t^{n-1} + w^{\text{fc}^{\top}}_{n+1} \cdot s_t^{n+1}\right) \qquad \forall n \in [n_{\text{conv}}+2, n_{\text{conv}}+n_{\text{fc}}] \\ s^{n_{\text{conv}}+1}_{t+1} & = \sigma\left(w^{\text{fc}}_{n_{\text{conv}}} \cdot \mathcal{F}(h_t^{n_{\text{conv}}}) + w^{\text{fc}^{\top}}_{n_{\text{conv}}+2} \cdot s_t^{n_{\text{conv}}+2}\right) \end{cases} ,$$

where $\mathcal{F}$ denotes the flatten operation. In the second phase:

$$\forall t \in [0,K] : \begin{cases} \hat{y}^{\beta}_{t+1} & = \sigma\left(w^{\text{fc}}_{n_{\text{conv}}+n_{\text{fc}}} \cdot s_t^{n_{\text{conv}}+n_{\text{fc}},\beta}\right) + \beta(y - \hat{y}^{\beta}_t) \\ s^{n,\beta}_{t+1} & = \sigma\left(w^{\text{fc}}_{n-1} \cdot s_t^{n-1,\beta} + w^{\text{fc}^{\top}}_{n+1} \cdot s_t^{n+1,\beta}\right) \qquad \forall n \in [n_{\text{conv}}+2, n_{\text{conv}}+n_{\text{fc}}] \\ s^{n_{\text{conv}}+1,\beta}_{t+1} & = \sigma\left(w^{\text{fc}}_{n_{\text{conv}}} \cdot \mathcal{F}(h_t^{n_{\text{conv}},\beta}) + w^{\text{fc}^{\top}}_{n_{\text{conv}}+2} \cdot s_t^{n_{\text{conv}}+2,\beta}\right) \end{cases} ,$$

where y denotes the target. Conversely, convolutional layers read the following set of equations at all time:

$$\forall t : \begin{cases} h^{n_{\text{conv}}}_{t+1} & = \sigma\left(\mathcal{P}\left(w^{\text{conv}}_{n_{\text{conv}}-1} \star h_t^{n_{\text{conv}}-1}\right) + \mathcal{F}^{-1}\left(w^{\text{fc}^{\top}}_{n_{\text{conv}}+1} \cdot s_t^{n_{\text{conv}}+1}\right)\right) \\ h^{n}_{t+1} & = \sigma\left(\mathcal{P}\left(w^{\text{conv}}_{n-1} \star h_t^{n-1}\right) + \tilde{w}^{\text{conv}}_{n+1} \star \mathcal{P}^{-1}\left(h_t^{n+1}, \text{ind}(w_n^{\text{conv}} \star h_{t-1}^{n})\right)\right) \\ & \forall n \in [2, n_{\text{conv}}-1] \\ h^{1}_{t+1} & = \sigma\left(\mathcal{P}\left(w_0^{\text{conv}} \star x\right) + \tilde{w}_2^{\text{conv}} \star \mathcal{P}^{-1}\left(h_t^2, \text{ind}(w_1^{\text{conv}} \star h_{t-1}^1)\right)\right) \end{cases} .$$

From here on, we shall omit the second argument of inverse pooling $\mathcal{P}^{-1}$ - i.e. the locations of the maximal neuron values before applying pooling - to improve readability of the equations and proofs. Considering the function:

$$\Phi(x, h^1, \cdots, h^{n_{\text{conv}}}, s^{n_{\text{conv}}+1}, \cdots, \hat{y}) = \sum_{n=n_{\text{conv}}+2}^{n_{\text{conv}}+n_{\text{fc}}} s^{n+1^\top} \cdot w_n^{\text{fc}} \cdot s^n + s^{n_{\text{conv}}+1} \cdot w_{n_{\text{conv}}}^{\text{fc}} \cdot \mathcal{F}(h_t^{n_{\text{conv}}}) + \sum_{n=2}^{n_{\text{conv}}-1} h^{n+1} \bullet \mathcal{P}\left(w_n^{\text{conv}} \star h^n\right) + h^1 \bullet \mathcal{P}\left(w_1^{\text{conv}} \star x\right),$$

and ignoring the activation function, we have:

$$\begin{cases} \forall n \in [1, n_{\text{conv}}]: & h_t^n \approx \dfrac{\partial \Phi}{\partial h^n} \\ \forall n \in [n_{\text{conv}}, n_{\text{conv}} + n_{\text{fc}} + 1]: & s_t^n \approx \dfrac{\partial \Phi}{\partial s^n} \end{cases}, \tag{4.36}$$

so that in this case, we define the EP error processes for the parameters $\theta = \{w_n^{\text{fc}}, w_n^{\text{conv}}\}$ as, $\forall t \in [0, K]$:

$$\begin{cases} \forall n \in [n_{\text{conv}} + 1, n_{\text{fc}} + n_{\text{conv}}]: & \Delta_{w_n^{\text{fc}}}^{\text{EP}}(t) = \frac{1}{\beta}\left(s_{t+1}^{n+1,\beta} \cdot s_{t+1}^{n,\beta^\top} - s_t^{n+1,\beta} \cdot s_t^{n,\beta^\top}\right) \\ \Delta_{w_{n_{\text{conv}}}^{\text{fc}}}^{\text{EP}}(t) = \frac{1}{\beta}\left(s_{t+1}^{n_{\text{conv}}+1,\beta} \cdot \mathcal{F}\left(h_{t+1}^{n_{\text{conv}},\beta}\right)^\top - s_t^{n_{\text{conv}}+1,\beta} \cdot \mathcal{F}\left(h_t^{n_{\text{conv}},\beta}\right)^\top\right) \\ \forall n \in [1, N_{\text{conv}} - 2]: & \Delta_{w_n^{\text{conv}}}^{\text{EP}}(t) = \frac{1}{\beta}\left(\mathcal{P}^{-1}(h_{t+1}^{n+1,\beta}) \star h_{t+1}^{n,\beta} - \mathcal{P}^{-1}(h_t^{n+1,\beta}) \star h_t^{n,\beta}\right) \\ \Delta_{w_0^{\text{conv}}}^{\text{EP}}(t) = \frac{1}{\beta}\left(\mathcal{P}^{-1}(h_{t+1}^{1,\beta}) \star x - \mathcal{P}^{-1}(h_t^{1,\beta}) \star x\right) \end{cases}, \tag{4.37}$$

To further justify Eq. (4.36) and Eq. (4.37), we state and prove the following lemma.

Lemma 5. *Taking:*

$$\Phi = Y \bullet \mathcal{P}\left(w \star X\right),$$

and denoting $Z = w \star X$, we have:

$$\frac{\partial \Phi}{\partial Z} = \mathcal{P}^{-1}\left(Y\right) \tag{4.38}$$

$$\frac{\partial \Phi}{\partial X} = \tilde{w} \star \mathcal{P}^{-1}\left(Y\right) \tag{4.39}$$

$$\frac{\partial \Phi}{\partial w} = \mathcal{P}^{-1}\left(Y\right) \star X \tag{4.40}$$

$$\frac{\partial \Phi}{\partial Y} = \mathcal{P}\left(w \star X\right) \tag{4.41}$$

Proof of Lemma 5. Let us prove Eq. (4.38). We have:

$$\begin{aligned}\frac{\partial \Phi}{\partial Z_{c,x,y}} &= \sum_{c',i,j} Y_{c',i,j} \frac{\partial \mathcal{P}(Z)_{c',i,j}}{\partial Z_{c,x,y}} \\ &= \sum_{c',i,j} Y_{c',i,j} \frac{\partial Z_{c',F(i-1)+1+r^*(i),F(j-1)+1+s^*(j)}}{\partial Z_{c,x,y}} \\ &= \sum_{i,j} Y_{c,i,j} \delta_{x,F(i-1)+1+r^*(i)} \delta_{y,F(j-1)+1+s^*(j)} \\ &= \mathcal{P}^{-1}(Y)_{c,x,y},\end{aligned}$$

where we used Eq. (4.33) at the last step.

We can now proceed to proving Eq. (4.39). We have:

$$\begin{aligned}\frac{\partial \Phi}{\partial X_{c,p,q}} &= \sum_{c',x,y} \frac{\partial \Phi}{\partial Z_{c',x,y}} \cdot \frac{\partial Z_{c',x,y}}{\partial X_{c,p,q}} \\ &= \sum_{c',x,y} \mathcal{P}^{-1}(Y)_{c',x,y} \cdot \frac{\partial}{\partial X_{c,p,q}} \left(\sum_{c'',r,s} w_{c',c'',r,s} X_{c'',x+r-1,y+s-1} \right) \\ &= \sum_{c',x,y} \sum_{r,s} \mathcal{P}^{-1}(Y)_{c',x,y} w_{c',c,r,s} \delta_{p,x+r-1} \delta_{q,y+s-1} \\ &= \sum_{c',r,s} w_{c',c,r,s} \mathcal{P}^{-1}(Y)_{c',p-(r-1),q-(s-1)}.\end{aligned}$$

Using the flipped kernel $\tilde{w}$ and performing the change of variable $r \leftarrow F - r + 1$ and $s \leftarrow F - s + 1$, we obtain:

$$\frac{\partial \Phi}{\partial X_{c,p,q}} = \sum_{c',r,s} \tilde{w}_{c,c',r,s} \cdot \mathcal{P}^{-1}(Y)_{c',p+r-F,q+s-F}. \tag{4.42}$$

Note in Eq. (4.42) that $\mathcal{P}^{-1}(Y)$ indices can exceed their boundaries. Also, as stated previously, $\mathcal{P}^{-1}(Y)$ should be padded with $\tilde{P} = F - 1 - P$ so that we recover the size of X after transpose convolution. Without loss of generality, we assume $P = 0$. We subsequently defined the padded input $\overline{\mathcal{P}^{-1}(Y)}$ as:

$$\overline{\mathcal{P}^{-1}(Y)}_{c,p,q} = \begin{cases} \mathcal{P}^{-1}(Y)_{c,p-F+1,q-F+1} \text{ if } p,q \in [F, N+F-1] \\ 0 \text{ if } p,q \in [1, F-1] \cup [N+F, N+2(F-1)] \end{cases}, \tag{4.43}$$

where N denotes the dimension of $\mathcal{P}^{-1}(Y)$. Finally Eq. (4.42) can conveniently be rewritten as:

$$\frac{\partial \Phi}{\partial X_{c,p,q}} = \left(\tilde{w} \star \overline{\mathcal{P}^{-1}(Y)}\right)_{p,q}. \tag{4.44}$$

For the sake of readability, the padding is implicitly assumed whenever transpose convolution is performed so that we drop the bar notation.

We can now proceed to proving Eq. (4.40). We have:

$$\begin{aligned}
\frac{\partial \Phi}{\partial w_{c',c,r,s}} &= \sum_{c'',x,y} \frac{\partial \Phi}{\partial Z_{c'',x,y}} \cdot \frac{\partial Z_{c'',x,y}}{\partial w_{c',c,r,s}} \\
&= \sum_{c'',x,y} \mathcal{P}^{-1}(Y)_{c'',x,y} \cdot \frac{\partial}{\partial w_{c',c,r,s}} \left(\sum_{k,r',s'} w_{c'',k,r',s'} X_{k,x+r'-1,y+s'-1} \right) \\
&= \sum_{x,y} \mathcal{P}^{-1}(Y)_{c',x,y} \cdot X_{c,r+x-1,s+y-1} \\
&= \left(\mathcal{P}^{-1}(Y) \star X\right)_{c',c,r,s}
\end{aligned}$$

Finally, proving Eq. (4.41) is straightforward. □

Experiment. We have implemented an architecture with 2 convolution-pooling layers and 1 fully connected layer. The first and second convolution layers are made up of 5×5 kernels with 32 and 64 feature maps respectively. Convolutions are performed without padding and with stride 1. Pooling is performed with 2×2 filters and with stride 2.

The experimental protocol is the exact same as the one used on the fully connected layered architecture. The only difference is the activation function that we have used here is $\sigma(x) = \max(\min(x, 1), 0)$ which we shall refer to here for convenience as 'hard sigmoid function'. Precise values of the hyperparameters T, K, beta are given in Tab. 2.1.

We show on Fig. 4.12 that Δ^{EP} and $-\nabla^{\text{BPTT}}$ processes qualitatively very well coincide when presenting one MNIST sample to the network. Looking more carefully, we note that some Δ_s^{EP} processes collapse to zero. This signals the presence of neurons which saturate to their maximal or minimal values, as an effect of the non linearity used. Consequently, as these neurons cannot move, they cannot carry the error signals. We hypothesize that this accounts for the discrepancy in the results obtained with EP on the convolutional architecture with respect to BPTT.

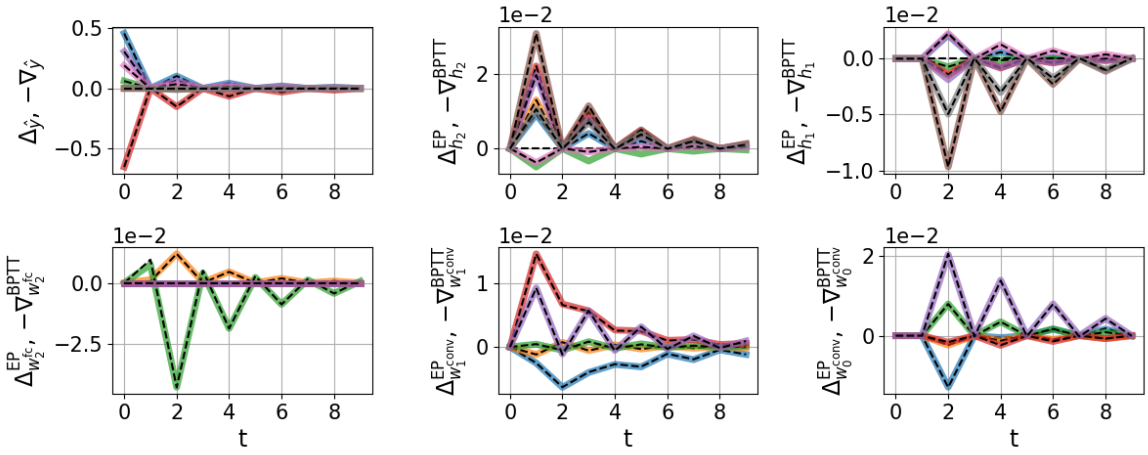

Figure 4.12: Demonstrating the GDU property with the convolutional architecture on MNIST. Dashed and continuous lines represent Δ^{EP} and $-\nabla^{\text{BPTT}}$ processes respectively, for 5 randomly selected neurons (top) and synapses (bottom) in each layer. Each randomly selected neuron or synapse corresponds to one color. Dashed and continuous lines mostly coincide. Some Δ^{EP} processes collapse to zero as an effect of the non-linearity. Interestingly, the Δ_s^{EP} and $-\nabla_s^{\text{BPTT}}$ processes are saw-teeth-shaped ; Appendix 2.3.3 accounts for this phenomenon.

Chapter 5

Experiments

5.1 Effect of depth and approximation

We consider a fully connected layered architecture where layers s^n are labelled in a backward fashion: s^0 denotes the output layer, s^1 the last hidden layer, and so forth. Two consecutive layers are reciprocally connected with tied weights with the convention that $W_{n,n+1}$ connects s^{n+1} to s^n. We study this architecture in the energy-based and prototypical setting as described per Equations (4.1) and (4.3) respectively, with corresponding weight updates (4.2) and (4.4). We study the GDU property layer-wise, e.g. RelMSE($\Delta^{\text{EP}}_{s^n}$, -$\nabla^{\text{BPTT}}_{s^n}$) measures the distance between the $\Delta^{\text{EP}}_{s^n}$ and $-\nabla^{\text{BPTT}}_{s^n}$ processes, averaged over all elements of layer s^n.

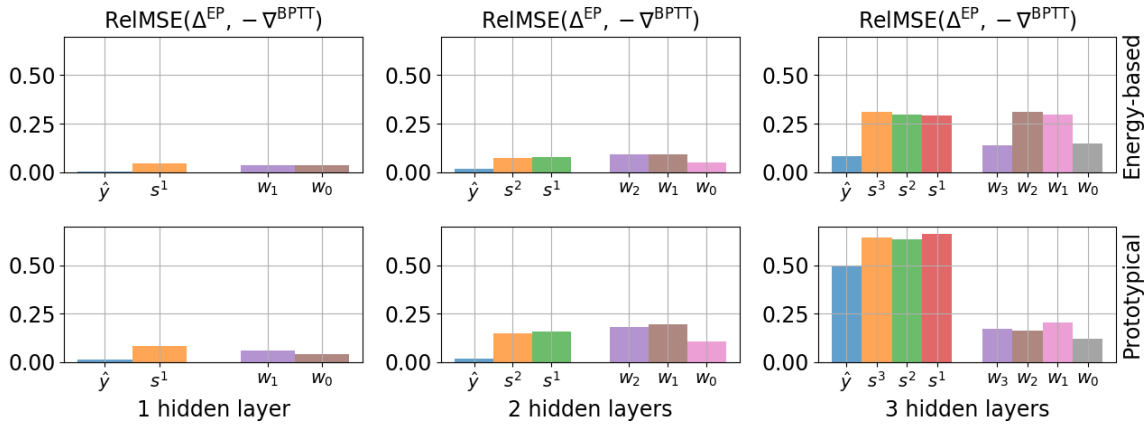

Figure 5.1: RelMSE analysis in the energy-based (top) and prototypical (bottom) setting. For one given architecture, each bar is labelled by a layer or synapses connecting two layers, e.g. the orange bar above s^1 represents $\text{RelMSE}(\Delta^{\text{EP}}_{s^1}, -\nabla^{\text{BPTT}}_{s^1})$. For each architecture, the recurrent hyperparameters T, K and β have been tuned to make the Δ^{EP} and $-\nabla^{\text{BPTT}}$ processes match best.

Table 5.1: Training results on MNIST with EP benchmarked against BPTT, in the energy-based and prototypical settings. "EB" and "P" respectively denote "energy-based" and "prototypical", "-#h" stands for the number of hidden layers and WCT for "Wall-clock time" in $hours : minutes$. We indicate over five trials the mean and standard deviation for the test error, the mean error in parenthesis for the train error. T (resp. K) is the number of iterations in the first (resp. second) phase.

	EP (error %)		BPTT (error %)		T	K	Epochs	WCT
	Test	Train	Test	Train				
EB-1h	2.06 ± 0.17	(0.13)	2.11 ± 0.09	(0.46)	100	12	30	1 : 33
EB-2h	2.01 ± 0.21	(0.11)	2.02 ± 0.12	(0.29)	500	40	50	16 : 04
P-1h	2.00 ± 0.13	(0.20)	2.00 ± 0.12	(0.55)	30	10	30	**0 : 17**
P-2h	1.95 ± 0.10	(0.14)	2.09 ± 0.12	(0.37)	100	20	50	**1 : 56**
P-3h	2.01 ± 0.18	(0.10)	2.30 ± 0.17	(0.32)	180	20	100	**8 : 27**
P-conv	$\mathbf{1.02 \pm 0.04}$	(0.54)	0.88 ± 0.06	(0.12)	200	10	40	**8 : 58**

Table 5.2: Best training results on MNIST with EP reported in the literature.

	EP (error %)	
	Test	Train
[160]	~ 2.2	(~ 0)
[163]	2.37	(0.15)
[164]	2.19	

We display in Fig. 5.1 the RelMSE, layer-wise for one, two and three hidden layered architecture (from left to right), in the energy-based (upper panels) and prototypical (lower panels) settings, so that each architecture in a given setting is displayed in one panel - see Table 2.1 of Appendix 2.3.1 for a detailed description of the hyperparameters and curve samples. In terms of RelMSE, we can see that the GDU property is best satisfied in the energy-based setting with one hidden layer where RelMSE is around $\sim 10^{-2}$ (top left). When adding more hidden layers in the energy-based setting (top middle and top right), the RelMSE increases to $\sim 10^{-1}$, with a greater RelMSE when going away from the output layer. The same is observed in the prototypical setting when we add more hidden layers (lower panels). Compared to the energy-based setting, although the RelMSEs associated with neurons are significantly higher in the prototypical setting, the RelMSEs associated with synapses are similar or lower. On average, the weight updates provided by EP match well the gradients of BPTT, in the energy-based setting as well as in the prototypical setting.

Chapter 6

Discussion

Table 5.1 shows the accuracy results on MNIST of several variations of our approach and Table 5.2 those of the literature - see Table 2.2 of Appendix 2.3.1 for a complete description of the hyperparameters used. First, EP overall performs as well or practically as well as BPTT in terms of test accuracy in all situations. Second, no degradation of accuracy is seen between using the prototypical (P) rather than the energy-based (EB) setting, although the prototypical setting requires three to five times less time steps in the first phase (T) and cuts the simulation time by a factor five to eight. Finally, the best EP result, $\sim 1\%$ test error, is obtained with our convolutional architecture. This is also the best performance reported in the literature on MNIST training with EP. BPTT achieves 0.90% test error using the same architecture. This slight degradation is due to saturated neurons which do no route error signals (as reported in the previous section). The prototypical situation allows using highly reduced number of time steps in the first phase than [160] and [163]. On the other hand, [164] manages to cut this number even more. This comes at the cost of using an extra network to learn proper initial states for the EP network, which is not needed in our approach.

Overall, our work broadens the scope of EP from its original formulation for biologically motivated real-time dynamics and sheds new light on its practical understanding. We first extended EP to a discrete-time setting, which reduces its computational cost and allows addressing situations closer to conventional machine learning. Theorem 4 demonstrated that the gradients provided by EP are strictly equal to the gradients computed with BPTT in specific conditions. Our numerical experiments confirmed the theorem and showed that its range of applicability extends well beyond the original formulation of EP to prototypical neural networks widely used today. These results highlight that, in principle, EP can reach the same performance as BPTT on benchmark tasks, for RNN models with *fixed input*. One limitation of our theory however is that it has yet to be adapted to sequential data: such an extension would require to capture and learn correlations between successive equilibrium states corresponding to different inputs.
Layer-wise analysis of the gradients computed by EP and BPTT show that the deeper the layer, the more difficult it becomes to ensure the GDU property. On top of non-linearity

effects, this is mainly due to the fact that the deeper the network, the longer it takes to reach equilibrium.

While this may be a conundrum for current processors, it should not be an issue for alternative computing schemes. Physics research is now looking at neuromorphic computing approaches that leverage the transient dynamics of physical devices for computation [67, 165, 166]. In such systems, based on magnetism or optics, dynamical equations are solved directly by the physical circuits and components, in parallel and at speed much higher than processors. On the other hand, in such systems, the nonlocality of backprop is a major concern [34]. In this context, EP appears as a powerful approach as computing gradients only requires measuring the system at the end of each phase, and going backward in time is not needed. In a longer term, interfacing the algorithmics of EP with device physics could help cutting drastically the cost of inference and learning of conventional computers, and thereby address one of the biggest technological limitations of deep learning.

Part V

Equilibrium Propagation with Continual Weight Updates

Summary

As presented in the last two parts, EP prescribes a learning rule which is local in space, with the same local computations being performed in both prediction and credit assignment phases. However, in existing implementations of EP, the learning rule is not local in time. EP proceeds in two successive phases, the weight update is performed after the dynamics of the second phase (credit assignment) have converged, and it requires information of the first phase that is no longer available physically. In this part, we propose a version of EP named Continual Equilibrium Propagation (C-EP), where neural and synaptic dynamics occur simultaneously throughout the second phase, so that the weight update becomes local in time. We prove theoretically that, provided the learning rates are sufficiently small, at each time step of the second phase, the dynamics of neurons and synapses follow the gradients of the loss given by BPTT. We demonstrate training with C-EP on MNIST and generalize C-EP to neural networks where neurons are connected by asymmetric connections. We show through experiments that the more closely the network updates follow the gradients of BPTT, the best it performs in terms of training. These results bring EP a step closer to biology and open up the possibility of extremely energy efficient hardware implementations while maintaining an intimate link with backpropagation. This work was done in collaboration with Benjamin Scellier, who carried out the derivation of theorem 10.

Introduction

The implementation of backpropagation on conventional computers or dedicated hardware consumes more energy than the brain by several orders of magnitude [167]. One path towards reducing the gap between brains and machines in terms of power consumption and thereby achieving fast and energy efficient AI is by investigating alternative learning paradigms relying, as synapses do in the brain, on locally available information, as we showed in section 3.3 of part I. In these regards, we showed in part III that Equilibrium Propagation (EP) is an alternative style of computation for estimating error gradients that presents significant advantages. A key property of EP is that, unlike Contrastive Hebbian Learning (CHL) and related algorithms, it is intimately linked to backpropagation. It has been shown that synaptic updates in EP compute gradients of recurrent backpropagation (RBP) (section 2.3 of part III) and backpropagation through time (part IV). This makes EP especially attractive for bridging the gap between neural networks developed by neuroscientists, neuromorphic researchers, and deep learning researchers.

Nevertheless, the bioplausibility of EP still undergoes major limitations. First, although EP is local in space, it is non-local in time. In all existing implementations of EP, the weight update is performed after the dynamics of the second phase have converged, when the first steady state is no longer physically available: the first steady state has to be stored. Second, the network dynamics have to derive from a primitive function which, in the Hopfield model, translates into to the requirement of symmetric weights. These two requirements are biologically unrealistic and also hinder the development of efficient EP computing hardware.

In this part, we propose an alternative implementation of EP, called *Continual Equilibrium Propagation* (C-EP) which features temporal locality, by enabling synaptic dynamics to occur throughout the second phase, simultaneously with neural dynamics. We then address the second issue by adapting C-EP to systems having asymmetric synaptic connections, taking inspiration from [147]; we call this modified version the Continual Vector Field approach (or C-VF).

More specifically, the contributions of the current part are the following:

- We introduce C-EP (Section 1.1-1.2), a new version of EP with continual weight updates. Like standard EP, the C-EP algorithm applies to networks whose synaptic

connections between neurons are assumed to be symmetric and tied.

- We show mathematically that, provided that the changes in synaptic strengths are sufficiently slow (i.e. the learning rates are sufficiently small), at each time step of the second phase the dynamics of neurons and synapses follow the gradients of the loss obtained with BPTT (Theorem 10 and Fig. 1.2, chapter 2). We call this property the *Gradient Descending Dynamics* (GDD) property, following the terminology used in part IV.

- We demonstrate training with C-EP on MNIST, with accuracy approaching the one obtained with standard EP (Section 4.1).

- We adapt C-EP to the more bio-realistic situation of a neural network with asymmetric connections between neurons. We call this C-VF as it is inspired by the *Vector Field* method proposed in [147]. We demonstrate this approach on MNIST, and show numerically that the training performance is correlated with the satisfaction of the Gradient Descending Dynamics property (Section 4.3).

- We also show how the Recurrent Backpropagation (RBP) algorithm of [154,155] relates to C-EP, EP and BPTT. We illustrate the equivalence of these four algorithms on a simple analytical model (Fig. 2.1) and we develop their general relationship.

Chapter 1

Equilibrium Propagation with Continual Weight Updates (C-EP)

Algorithm 4 EP

Input: x, y, θ, β, η.
Output: θ.

$s_0 \leftarrow 0$ ▷ First Phase
repeat
  $s_{t+1} \leftarrow \frac{\partial \Phi}{\partial s}(x, s_t, \theta)$
until $s_t = s_*$
Store s_*
$s_0^\beta \leftarrow s_*$ ▷ Second Phase
repeat
  $s_{t+1}^\beta \leftarrow \frac{\partial \Phi}{\partial s}\left(x, s_t^\beta, \theta\right) - \beta \frac{\partial \ell}{\partial s}\left(s_t^\beta, y\right)$
until $s_t^\beta = s_*^\beta$
▷ Global Parameter Update
$\theta \leftarrow \theta + \frac{\eta}{\beta}\left(\frac{\partial \Phi}{\partial \theta}\left(s_*^\beta, \theta\right) - \frac{\partial \Phi}{\partial \theta}\left(s_*, \theta\right)\right)$

Algorithm 5 C-EP (with simplified notations)

Input: x, y, θ, β, η.
Output: θ.

$s_0 \leftarrow 0$ ▷ First Phase
repeat
  $s_{t+1} \leftarrow \frac{\partial \Phi}{\partial s}(x, s_t, \theta)$
until $s_t = s_*$
$s_0^\beta \leftarrow s_*$ ▷ Second Phase
repeat
  $s_{t+1}^\beta \leftarrow \frac{\partial \Phi}{\partial s}\left(x, s_t^\beta, \theta\right) - \beta \frac{\partial \ell}{\partial s}\left(s_t^\beta, y\right)$
  ▷ Parameter Update at Time t
  $\theta \leftarrow \theta + \frac{\eta}{\beta}\left(\frac{\partial \Phi}{\partial \theta}\left(s_{t+1}^\beta\right) - \frac{\partial \Phi}{\partial \theta}\left(s_t^\beta\right)\right)$
until s_t^β and θ are converged.

Figure 1.1: **Left.** Pseudo-code of EP. This is the version of EP for discrete-time dynamics introduced in part IV. **Right.** Pseudo-code of C-EP with simplified notations (see section 1.2 for a formal definition of C-EP). **Difference between EP and C-EP.** In EP, one global parameter update is performed at the end of the second phase ; in C-EP, parameter updates are performed throughout the second phase. Eq. (1.2) shows that the continual updates of C-EP add up to the global update of EP.

This chapter presents the main theoretical contributions of this part. Again, the loss of interest is the cost function ℓ evaluated at equilibrium:

$$\mathcal{L}^* = \ell(s_*, y). \tag{1.1}$$

We introduce a new algorithm to optimize $\mathcal{L}^*$, a new version of EP with continual parameter updates that we call C-EP. Unlike typical machine learning algorithms (such as BPTT, RBP and EP) in which the weight updates occur after all the other computations in the system are performed, our algorithm offers a mechanism in which the weights are updated continuously as the state of the neurons change.

1.1 From EP to C-EP: An intuition behind continual weight updates

The key idea to understand how to go from EP to C-EP is that the gradient of EP appearing in Eq. (2.3) reads as the following telescopic sum:

$$\begin{aligned} &\underbrace{\frac{1}{\beta}\left(\frac{\partial \Phi}{\partial \theta}\left(x, s_*^\beta, \theta\right) - \frac{\partial \Phi}{\partial \theta}\left(x, s_*, \theta\right)\right)}_{\text{global parameter gradient in EP}} \\ &= \sum_{t=1}^{\infty} \underbrace{\frac{1}{\beta}\left(\frac{\partial \Phi}{\partial \theta}\left(x, s_t^\beta, \theta\right) - \frac{\partial \Phi}{\partial \theta}\left(x, s_{t-1}^\beta, \theta\right)\right)}_{\text{parameter gradient at time t in C-EP}}. \end{aligned} \tag{1.2}$$

In Eq. (1.2) we have used that $s_0^\beta = s_*$ and $s_t^\beta \to s_*^\beta$ as $t \to \infty$. Here lies the very intuition of continual updates motivating this work; instead of keeping the weights fixed throughout the second phase and updating them at the end of the second phase based on the steady states s_* and s_*^β, as in EP (Alg. 4), the idea of the C-EP algorithm is to update the weights at each time t of the second phase between two consecutive states s_{t-1}^β and s_t^β (Alg. 5). One key difference in C-EP compared to EP though, is that, in the second phase, the weight update at time step t influences the neural states at time step $t+1$ in a nontrivial way, as illustrated in the computational graph of Fig. 1.2. In the next section, we define C-EP using notations that explicitly show this dependency.

1.2 Description of the C-EP algorithm

The first phase of C-EP is the same as that of EP (see Alg. 5 compared to Alg. 4). In the second phase of C-EP, the parameter variable is regarded as another dynamic variable θ_t that evolves with time t along with s_t. The dynamics of s_t and θ_t in the second phase of C-EP depend on the values of the two hyperparameters β (the hyperparameter of influence) and η (the learning rate), therefore we write $s_t^{\beta,\eta}$ and $\theta_t^{\beta,\eta}$ to show explicitly this dependence. With

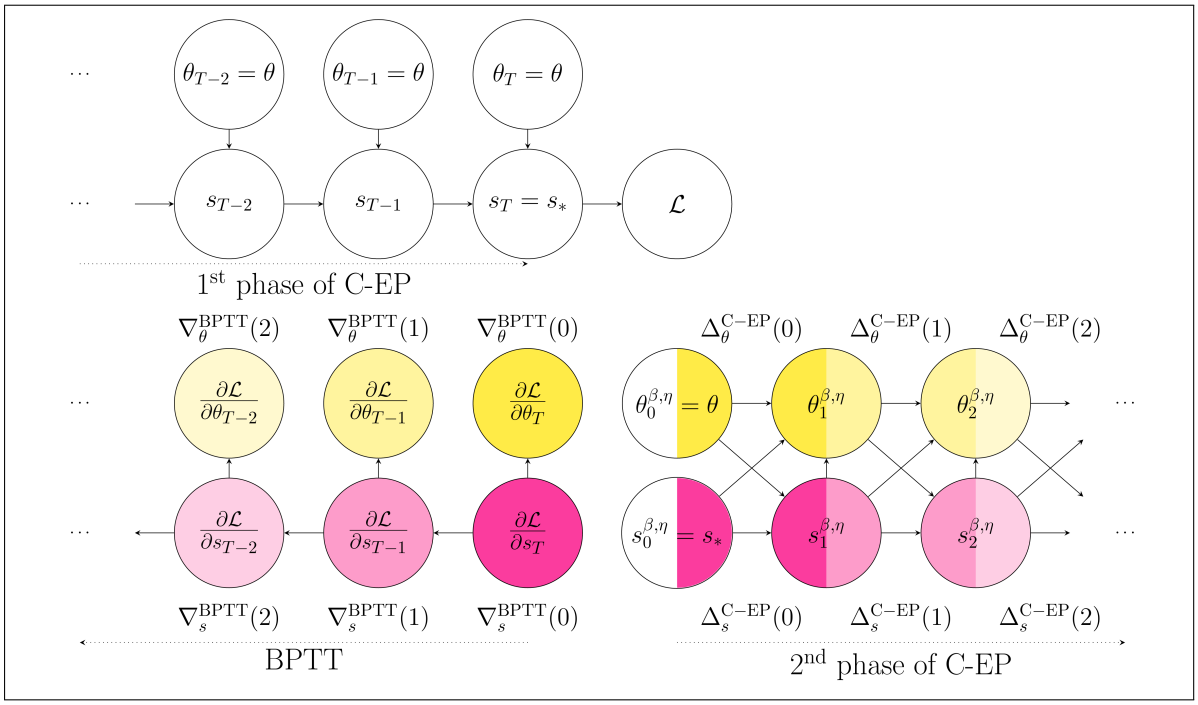

Figure 1.2: **Gradient-Descending Dynamics (GDD, Theorem 10).** In the second phase of Continual Equilibrium Prop (C-EP), the dynamics of neurons and synapses descend the gradients of BPTT, i.e. $\Delta^{\mathrm{C-EP}}(t) = -\nabla^{\mathrm{BPTT}}(t)$. The colors illustrate when corresponding computations are realized in C-EP and BPTT. **Top left.** 1st phase of C-EP with static input x and target y. The final State s_T is the steady state s_*. **Bottom left.** Backprop through time (BPTT). **Bottom right.** 2nd phase of C-EP. The starting State $s_0^{\beta,\eta}$ is the final State of the forward-time pass, i.e. the steady state s_*.

now both the neurons and the synapses evolving in the second phase, the dynamic variables $s_t^{\beta,\eta}$ and $\theta_t^{\beta,\eta}$ start from $s_0^{\beta,\eta} = s_*$ and $\theta_0^{\beta,\eta} = \theta$ and follow, $\forall t \geq 0$:

$$\begin{aligned} s_{t+1}^{\beta,\eta} &= \frac{\partial \Phi}{\partial s}\left(x, s_t^{\beta,\eta}, \theta_t^{\beta,\eta}\right) - \beta \, \frac{\partial \ell}{\partial s}\left(s_t^{\beta,\eta}, y\right), \\ \theta_{t+1}^{\beta,\eta} &= \theta_t^{\beta,\eta} + \frac{\eta}{\beta}\left(\frac{\partial \Phi}{\partial \theta}\left(x, s_{t+1}^{\beta,\eta}, \theta_t^{\beta,\eta}\right) - \frac{\partial \Phi}{\partial \theta}\left(x, s_t^{\beta,\eta}, \theta_t^{\beta,\eta}\right)\right). \end{aligned} \tag{1.3}$$

The difference in C-EP compared to EP is that the value of the parameter used to update $s_{t+1}^{\beta,\eta}$ in Eq. (1.3) is the current $\theta_t^{\beta,\eta}$, not θ. Provided the learning rate η is small enough, i.e. the synapses are slow to change compared to the neurons, this effect is weak. Intuitively, in the limit $\eta \to 0$, the parameter changes are negligible so that $\theta_t^{\beta,\eta}$ can be approximated by its initial value $\theta_0^{\beta,\eta} = \theta$. Under this approximation, the dynamics of $s_t^{\beta,\eta}$ in C-EP and the dynamics of s_t^{β} in EP are the same. See Fig. 2.1 for a simple example.

Chapter 2

Gradient Descending Dynamics (GDD) property

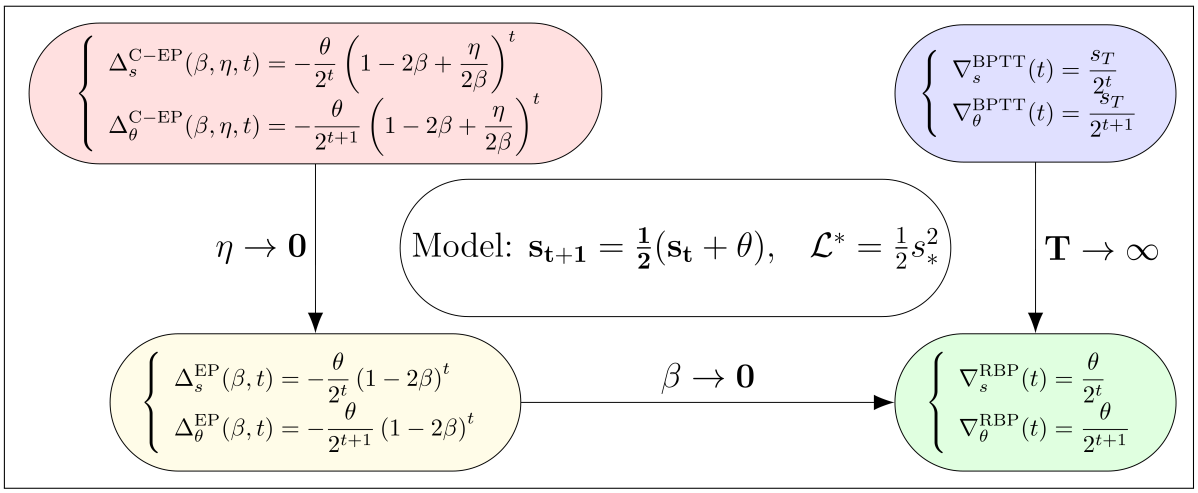

Figure 2.1: Illustration of Theorem 10 on a simple model. The variables s and θ are scalars, the first phase equation is $s_{t+1} = \frac{1}{2}(s_t + \theta)$, the steady state is denoted s_* and the loss is $\mathcal{L}^* = \frac{1}{2}s_*^2$. See Appendix 3.3 for derivation details. For completeness, we also include the corresponding gradients of Recurrent Backpropagation (RBP) and the normalized updates of EP, denoted ∇^{RBP} and Δ^{EP} respectively. The equivalence between C-EP, EP, RBP and BPTT holds in the general setting.

The main theoretical contribution of this part is to prove that, provided the hyperparameter β and the learning rate η are small enough, the dynamics of the neurons and the weights given by Eq. (1.3) follow the gradients of BPTT (Theorem 10 and Fig. 1.2). For a formal statement of this property and in a similar way than the previous part, we define the *normalized (continual) updates* of C-EP:

$$\begin{cases} \Delta_s^{\text{C-EP}}(\beta, \eta, t) = \dfrac{1}{\beta}\left(s_{t+1}^{\beta,\eta} - s_t^{\beta,\eta}\right), \\ \Delta_\theta^{\text{C-EP}}(\beta, \eta, t) = \dfrac{1}{\eta}\left(\theta_{t+1}^{\beta,\eta} - \theta_t^{\beta,\eta}\right), \end{cases} \tag{2.1}$$

as well as the gradients of the loss $\mathcal{L} = \ell\left(s_T, y\right)$ after T time steps, computed with BPTT:

$$\begin{cases} \nabla_s^{\text{BPTT}}(t) = \dfrac{\partial \mathcal{L}}{\partial s_{T-t}}, \\ \nabla_\theta^{\text{BPTT}}(t) = \dfrac{\partial \mathcal{L}}{\partial \theta_{T-t}}. \end{cases} \tag{2.2}$$

Note that injecting Eq. (1.3) in Eq. (2.1), the normalized updates of C-EP read:

$$\Delta_\theta^{\text{C-EP}}(\beta, \eta, t) = \frac{1}{\beta}\left(\frac{\partial \Phi}{\partial \theta}\left(x, s_{t+1}^{\beta,\eta}, \theta_t^{\beta,\eta}\right) - \frac{\partial \Phi}{\partial \theta}\left(x, s_t^{\beta,\eta}, \theta_t^{\beta,\eta}\right)\right), \tag{2.3}$$

which corresponds to the parameter gradient at time t, defined informally in Eq. (1.2).

Again and as in the two previous parts, the loss to optimize is still the cost function at equilibrium:

$$\mathcal{L}^* = \ell(s_*, y). \tag{2.4}$$

Backpropagation Through Time (BPTT) described in the Introduction and in the previous part, Equilibrium Propagation and Recurrent Backpropagation (RBP) [154, 155] can all optimize $\mathcal{L}_*$, and we show in this part that Continual Equilibrium Propagation also can. To demonstrate the equivalence between C-EP and BPTT, we will show that these four algorithms are computationally equivalent. Namely, the proof outline is the following:

- BPTT and RBP are equivalent (section 2.1, Lemma 7). This link is known since the late 1980s and can be found in [168].
- EP and RBP are also equivalent (section 2.2, Lemma 8). This result was proved in [148] in the setting of real-time dynamics.
- Finally, C-EP and EP are equivalent (section 2.3, Lemma 9), which is the new ingredient of this part.
- We conclude that, by transitivity, BPTT and C-EP are equivalent (section 2.4, Theorem 10).

This outline is illustrated on a very simple model in Fig. 2.1: the equivalence between BPTT and RBP is illustrated with the link between the blue and the green blobs, the equivalence between EP and RBP with the link between the yellow and green blobs and finally

the equivalence between C-EP and EP with the link between the red and yellow blobs. See Appendix 3.3 for the derivation details of this toy model.

2.1 Equivalence between BPTT and RBP

Definition of Recurrent Backpropagation (RBP). The Almeida-Pineda algorithm (a.k.a. Recurrent Backpropagation, or RBP for short), which was invented independently in [154] and [155], relies on this property to compute the gradients of the loss $\mathcal{L}^*$ using only the steady state s_*. Similarly to BPTT, it computes quantities $\nabla_s^{\mathrm{RBP}}(t)$ and $\nabla_\theta^{\mathrm{RBP}}(t)$, which we call 'gradients of RBP', iteratively for $t = 0, 1, 2, \ldots$.

Definition 6 (Gradients of RBP)**.** *The gradients $\nabla_s^{\mathrm{RBP}}(t)$ and $\nabla_\theta^{\mathrm{RBP}}(t)$ are defined and computed iteratively as follows:*

$$\nabla_s^{\mathrm{RBP}}(0) = \frac{\partial \ell}{\partial s}(s_*, y), \tag{2.5}$$

$$\forall t \geq 0, \qquad \nabla_s^{\mathrm{RBP}}(t+1) = \frac{\partial F}{\partial s}(x, s_*, \theta)^\top \cdot \nabla_s^{\mathrm{RBP}}(t), \tag{2.6}$$

$$\forall t \geq 0, \qquad \nabla_\theta^{\mathrm{RBP}}(t+1) = \frac{\partial F}{\partial \theta}(x, s_*, \theta)^\top \cdot \nabla_s^{\mathrm{RBP}}(t). \tag{2.7}$$

In Appendix 3.1, we justify the name of 'gradients' for the quantities $\nabla_s^{\mathrm{RBP}}(t)$ and $\nabla_\theta^{\mathrm{RBP}}(t)$ by proving that they are the gradients of $\mathcal{L}^*$. More explicitely:

$$\sum_{t=1}^{\infty} \nabla_\theta^{\mathrm{RBP}}(t) = \frac{\partial \mathcal{L}^*}{\partial \theta}. \tag{2.8}$$

In general, to apply BPTT, it is necessary to store in memory the history of past hidden states $s_1, s_2, \ldots, s_T$ in order to compute the gradients $\nabla_s^{\mathrm{BPTT}}(t)$ and $\nabla_\theta^{\mathrm{BPTT}}(t)$ as in Eq. (1.8)-(1.9). However, in our specific setting with static input x, if the network has reached the steady state s_* after $T-K$ steps, i.e. if $s_{T-K} = s_{T-K+1} = \cdots = s_{T-1} = s_T = s_*$, then we see that, in order to compute the first K gradients of BPTT, all one needs to know is $\frac{\partial F}{\partial s}(x, s_*, \theta)$ and $\frac{\partial F}{\partial \theta}(x, s_*, \theta)$. To this end, all one needs to keep in memory is the steady state s_*. In this particular setting, it is not necessary to store the past hidden states $s_T, s_{T-1}, \ldots, s_{T-K}$ since they are all equal to s_* Unlike BPTT where keeping the history of past hidden states is necessary to compute (or 'backpropagate') the gradients, in RBP Eq. (2.6)-(2.7) show that it is sufficient to keep in memory the steady state s_* only in order to iterate the computation of the gradients. RBP is more memory efficient than BPTT.

Lemma 7 (Equivalence of BPTT and RBP)**.** *In the setting with static input x, suppose that the network has reached the steady state s_* after $T-K$ steps, i.e. $s_{T-K} = s_{T-K+1} = \cdots = s_{T-1} = s_T = s_*$. Then the first K gradients of BPTT are equal to the first K gradient of*

Algorithm 6 BPTT

Input: x, y, θ.
Output: θ.

$s_0 \leftarrow 0$
for $t = 0$ to $T-1$ **do**
  $s_{t+1} \leftarrow F(x, s_t, \theta)$
end for
$\nabla_s^{\mathrm{BPTT}}(0) \leftarrow \frac{\partial \ell}{\partial s}(s_T, y)$
for $t = 1$ to T **do**
  $\nabla_s(t) \leftarrow \frac{\partial F}{\partial s}(x, s_{T-t}, \theta)^\top \cdot \nabla_s(t-1)$
  $\nabla_\theta(t) \leftarrow \frac{\partial F}{\partial \theta}(x, s_{T-t}, \theta)^\top \cdot \nabla_s(t-1)$
end for
$\nabla_\theta^{\mathrm{BPTT}}(\mathrm{tot}) \leftarrow \sum_{t=0}^{T-1} \nabla_\theta^{\mathrm{BPTT}}(t)$

Algorithm 7 RBP

Input: x, y, θ.
Output: θ.

$s_0 \leftarrow 0$
repeat
  $s_{t+1} \leftarrow F(x, s_t, \theta)$
until $s_t = s_*$
$\nabla_s^{\mathrm{RBP}}(0) \leftarrow \frac{\partial \ell}{\partial s}(s_*, y)$
repeat
  $\nabla_s^{\mathrm{RBP}}(t) \leftarrow \frac{\partial F}{\partial s}(x, s_*, \theta)^\top \cdot \nabla_s^{\mathrm{RBP}}(t-1)$
  $\nabla_\theta^{\mathrm{RBP}}(t) \leftarrow \frac{\partial F}{\partial \theta}(x, s_*, \theta)^\top \cdot \nabla_s^{\mathrm{RBP}}(t-1)$
until $\nabla_\theta^{\mathrm{RBP}}(t) = 0$.
$\nabla_\theta^{\mathrm{RBP}}(\mathrm{tot}) \leftarrow \sum_{t=0}^{\infty} \nabla_\theta^{\mathrm{RBP}}(t)$

Figure 2.2: **Left.** Pseudo-code of BPTT. The gradients $\nabla(t)$ denote the gradients $\nabla^{\mathrm{BPTT}}(t)$ of BPTT. **Right.** Pseudo-code of RBP. **Difference between BPTT and RBP.** In BPTT, the state s_{T-t} is required to compute $\frac{\partial F}{\partial s}(x, s_{T-t}, \theta)$ and $\frac{\partial F}{\partial \theta}(x, s_{T-t}, \theta)$; thus it is necessary to store in memory the sequence of states $s_1, s_2, \ldots, s_T$. In contrast, in RBP, only the steady state s_* is required to compute $\frac{\partial F}{\partial s}(x, s_*, \theta)$ and $\frac{\partial F}{\partial \theta}(x, s_*, \theta)$; it is not necessary to store the past states of the network.

RBP, i.e.

$$\forall t = 0, 1, \ldots, K : \begin{cases} \nabla_s^{\mathrm{BPTT}}(t) = \nabla_s^{\mathrm{RBP}}(t), \\ \nabla_\theta^{\mathrm{BPTT}}(t) = \nabla_\theta^{\mathrm{RBP}}(t). \end{cases} \tag{2.9}$$

Proof of Lemma 7. Using $s_{T-K} = s_{T-K+1} = \cdots = s_{T-1} = s_T = s_*$ along with Eq. (1.7), Eq. (1.8) and Eq. (1.9) (the set of equations satisfied by BPTT described in the previous part), ∇_s^{BPTT} (resp. $\nabla_\theta^{\mathrm{BPTT}}$) and ∇_s^{RBP} (resp. $\nabla_\theta^{\mathrm{RBP}}$) satisfy the same recursive equations with the same initial conditions, so that BPTT and RBP error processes are equal at all times. Hence $\nabla_s^{\mathrm{RBP}}(t) = \nabla_s^{\mathrm{BPTT}}(t), \quad \nabla_\theta^{\mathrm{RBP}}(t) = \nabla_\theta^{\mathrm{BPTT}}(t) \quad \forall t = 0, 1, \ldots, K.$ □

2.2 Equivalence between EP and RBP

For completeness of this part, we state the equivalence between EP and RBP. Note that the equations defining RBP Eq. (2.5)-(2.7) are the same than those satisfied by BPTT at equilibrium in the previous part (Lemma 2), so that Lemma 8 is simply a reformulation of the theorem of the previous part (Theorem 4).

Lemma 8 (Equivalence of EP and RBP)**.** *Assume that the transition function derives from a primitive function, i.e. that F is of the form $F(x, s, \theta) = \frac{\partial \Phi}{\partial s}(x, s, \theta)$. Then, in the limit of*

small hyperparameter β, the normalized updates of EP are equal to the gradients of RBP:

$$\forall t \geq 0 : \begin{cases} \lim\limits_{\beta \to 0 \ (\beta > 0)} \Delta_s^{\mathrm{EP}}(\beta, t) = -\nabla_s^{\mathrm{RBP}}(t), \\ \lim\limits_{\beta \to 0 \ (\beta > 0)} \Delta_\theta^{\mathrm{EP}}(\beta, t) = -\nabla_\theta^{\mathrm{RBP}}(t). \end{cases} \tag{2.10}$$

Proof of Lemma 8. See proof of Theorem 4 in the previous part. □

2.3 Equivalence between EP and C-EP

First, recall the dynamics of C-EP in the second phase: starting from $s_0^{\beta,\eta} = s_*$ and $\theta_0^{\beta,\eta} = \theta$ we have $\forall t \geq 0$:

$$\begin{cases} s_{t+1}^{\beta,\eta} = \dfrac{\partial \Phi}{\partial s}\left(x, s_t^{\beta,\eta}, \theta_t^{\beta,\eta}\right) - \beta \dfrac{\partial \ell}{\partial s}\left(s_t^{\beta,\eta}, y\right), \\ \theta_{t+1}^{\beta,\eta} = \theta_t^{\beta,\eta} + \dfrac{\eta}{\beta}\left(\dfrac{\partial \Phi}{\partial \theta}\left(x, s_{t+1}^{\beta,\eta}, \theta_t^{\beta,\eta}\right) - \dfrac{\partial \Phi}{\partial \theta}\left(x, s_t^{\beta,\eta}, \theta_t^{\beta,\eta}\right)\right). \end{cases} \tag{2.11}$$

We have also defined the normalized updates of C-EP:

$$\forall t \geq 0 : \begin{cases} \Delta_s^{\mathrm{C-EP}}(\beta, \eta, t) = \dfrac{1}{\beta}\left(s_{t+1}^{\beta,\eta} - s_t^{\beta,\eta}\right), \\ \Delta_\theta^{\mathrm{C-EP}}(\beta, \eta, t) = \dfrac{1}{\eta}\left(\theta_{t+1}^{\beta,\eta} - \theta_t^{\beta,\eta}\right). \end{cases} \tag{2.12}$$

We also recall the dynamics of EP in the second phase:

$$s_0^\beta = s_* \quad \text{and} \quad s_{t+1}^\beta = \frac{\partial \Phi}{\partial s}\left(x, s_t^\beta, \theta\right) - \beta \frac{\partial \ell}{\partial s}\left(s_t^\beta, y\right), \tag{2.13}$$

as well as the normalized updates of EP, as defined in part IV:

$$\forall t \geq 0 : \begin{cases} \Delta_s^{\mathrm{EP}}(\beta, t) = \dfrac{1}{\beta}\left(s_{t+1}^\beta - s_t^\beta\right), \\ \Delta_\theta^{\mathrm{EP}}(\beta, t) = \dfrac{1}{\beta}\left(\dfrac{\partial \Phi}{\partial \theta}\left(x, s_{t+1}^\beta, \theta\right) - \dfrac{\partial \Phi}{\partial \theta}\left(x, s_t^\beta, \theta\right)\right). \end{cases} \tag{2.14}$$

Lemma 9 (Equivalence of C-EP and EP)**.** *In the limit of small learning rate, i.e. $\eta \to 0$, the (normalized) updates of C-EP are equal to those of EP:*

$$\forall t \geq 0 : \begin{cases} \lim\limits_{\eta \to 0 \ (\eta > 0)} \Delta_s^{\mathrm{C-EP}}(\beta, \eta, t) = \Delta_s^{\mathrm{EP}}(\beta, t), \\ \lim\limits_{\eta \to 0 \ (\eta > 0)} \Delta_\theta^{\mathrm{C-EP}}(\beta, \eta, t) = \Delta_\theta^{\mathrm{EP}}(\beta, t). \end{cases} \tag{2.15}$$

Proof of Lemma 9. We want to compute the limits of $\Delta_s^{\mathrm{C-EP}}(\beta, \eta, t)$ and $\Delta_\theta^{\mathrm{C-EP}}(\beta, \eta, t)$ as $\eta \to 0$ with $\eta > 0$. Note that it is crucial that this property is derived for $\eta > 0$ so that learning is actually performed. In a subtle way though, the proof can be derived with $\eta = 0$

for the following mathematical reasons. Note that assuming the regularity on the functions Φ and ℓ (e.g. continuous differentiability), for fixed t and β, the quantities $s_t^{\beta,\eta}$ and $\theta_t^{\beta,\eta}$ are continuous as functions of η; this is straightforward from the form of Eq. (2.11). As a consequence, $\Delta_s^{\mathrm{C-EP}}(\beta,\eta,t)$ is a continuous function of η, which implies in particular that:

$$\lim_{\eta\to 0\ (\eta>0)} \Delta_s^{\mathrm{C-EP}}(\beta,\eta,t) = \Delta_s^{\mathrm{C-EP}}(\beta,0,t). \tag{2.16}$$

Now, taking $\eta = 0$ in the bottom equation of Eq. (2.11) yields the recurrence relation $\theta_{t+1}^{\beta,0} = \theta_t^{\beta,0}$, so that $\theta_t^{\beta,0} = \theta_0^{\beta,0} = \theta$ for every t. Injecting $\theta_t^{\beta,0} = \theta$ in the top equation of Eq. (2.11) yields for $s_t^{\beta,0}$ the same recurrence relation as that of s_t^{β} (Eq. 2.13), so that $s_t^{\beta,0} = s_t^{\beta}$ for every t. Therefore, for $\eta = 0$, we have:

$$\begin{aligned}\Delta_s^{\mathrm{C-EP}}(\beta,0,t) &= \frac{1}{\beta}\left(s_{t+1}^{\beta,0} - s_t^{\beta,0}\right)\\ &= \frac{1}{\beta}\left(s_{t+1}^{\beta} - s_t^{\beta}\right)\\ &= \Delta_s^{\mathrm{EP}}(\beta,t).\end{aligned} \tag{2.17}$$

It follows from Eq. (2.16) and Eq. (2.17) that

$$\lim_{\eta\to 0\ (\eta>0)} \Delta_s^{\mathrm{C-EP}}(\beta,\eta,t) = \Delta_s^{\mathrm{EP}}(\beta,t). \tag{2.18}$$

Now let us compute $\lim_{\eta\to 0\ (\eta>0)} \Delta_\theta^{\mathrm{C-EP}}(\beta,\eta,t)$. Using Eq. (2.11), we have

$$\Delta_\theta^{\mathrm{C-EP}}(\beta,\eta,t) = \frac{1}{\eta}\left(\theta_{t+1}^{\beta,\eta} - \theta_t^{\beta,\eta}\right) \tag{2.19}$$

$$= \frac{1}{\beta}\left(\frac{\partial\Phi}{\partial\theta}\left(x, s_{t+1}^{\beta,\eta}, \theta_t^{\beta,\eta}\right) - \frac{\partial\Phi}{\partial\theta}\left(x, s_t^{\beta,\eta}, \theta_t^{\beta,\eta}\right)\right). \tag{2.20}$$

Similarly as before, for fixed t, $\frac{\partial\Phi}{\partial\theta}\left(x, s_t^{\beta,\eta}, \theta_t^{\beta,\eta}\right)$ is a continuous function of η. Therefore

$$\begin{aligned}&\lim_{\eta\to 0\ (\eta>0)} \Delta_\theta^{\mathrm{C-EP}}(\beta,\eta,t)\\ &= \frac{1}{\beta}\left(\frac{\partial\Phi}{\partial\theta}\left(x, s_{t+1}^{\beta,0}, \theta_t^{\beta,0}\right) - \frac{\partial\Phi}{\partial\theta}\left(x, s_t^{\beta,0}, \theta_t^{\beta,0}\right)\right)\\ &= \frac{1}{\beta}\left(\frac{\partial\Phi}{\partial\theta}\left(x, s_{t+1}^{\beta}, \theta\right) - \frac{\partial\Phi}{\partial\theta}\left(x, s_t^{\beta}, \theta\right)\right)\\ &= \Delta_\theta^{\mathrm{EP}}(\beta,t).\end{aligned} \tag{2.21}$$

□

An explicit link between EP and C-EP weight updates. A consequence of Lemma 9 is that the total update of C-EP matches the total update of EP in the limit of small η, so that we retrieve the standard EP learning rule of Eq. (2.3). More explicitly, after K steps in the second phase and starting from $\theta_0^{\beta,\eta} = \theta_0$:

$$\theta_K^{\beta,\eta} - \theta_0 = \sum_{t=0}^{K-1} \theta_{t+1}^{\beta,\eta} - \theta_t^{\beta,\eta} \tag{2.22}$$

$$= \sum_{t=0}^{K-1} \eta \Delta_\theta^{\mathrm{C-EP}}(\beta, \eta, t) \tag{2.23}$$

$$\underset{\eta \gtrsim 0}{\approx} \sum_{t=0}^{K-1} \eta \Delta_\theta^{\mathrm{EP}}(\beta, t) \tag{2.24}$$

$$= \sum_{t=0}^{K-1} \eta \frac{1}{\beta} \left(\frac{\partial \Phi}{\partial \theta}(x, s_{t+1}, \theta_0) - \frac{\partial \Phi}{\partial \theta}(x, s_{t+1}, \theta_0) \right) \tag{2.25}$$

$$= \frac{\eta}{\beta} \left(\frac{\partial \Phi}{\partial \theta}(x, s_K^\beta, \theta_0) - \frac{\partial \Phi}{\partial \theta}(x, s_*, \theta_0) \right) \tag{2.26}$$

Here we have used successively the definition of $\Delta_\theta^{\mathrm{C-EP}}$ (Eq. (2.1)), Lemma 9 and the definition of $\Delta_\theta^{\mathrm{EP}}$ (Eq. (2.14)).

2.4 Main result

We now ready to state the main result of this part: the equivalence between C-EP and BPTT.

Theorem 10 (GDD Property)**.** *Let $s_0, s_1, \ldots, s_T$ be the convergent sequence of states and denote $s_* = s_T$ the steady state. Further assume that there exists some step K where $0 < K \leq T$ such that $s_* = s_T = s_{T-1} = \ldots s_{T-K}$. Then, in the limit $\eta \to 0$ and $\beta \to 0$, the first K normalized updates in the second phase of C-EP are equal to the negatives of the first K gradients of BPTT, i.e. $\forall t = 0, 1, \ldots, K$:*

$$\begin{cases} \lim\limits_{\beta \to 0} \lim\limits_{\eta \to 0} \Delta_s^{\mathrm{C-EP}}(\beta, \eta, t) = -\nabla_s^{\mathrm{BPTT}}(t), \\ \lim\limits_{\beta \to 0} \lim\limits_{\eta \to 0} \Delta_\theta^{\mathrm{C-EP}}(\beta, \eta, t) = -\nabla_\theta^{\mathrm{BPTT}}(t). \end{cases} \tag{2.27}$$

Proof of Lemma 10. As stated in the proof outline, Theorem 10 is a consequence of Lemma (8), Lemma (9) and Lemma (7). □

Remarks. Note that:

- Fig. 2.1 illustrates Theorem 10 with a simple dynamical system for which the normalized updates $\Delta^{\mathrm{C-EP}}$ and the gradients ∇^{BPTT} are analytically tractable.
- Theorem 10 rewrites $s_{t+1}^{\beta,\eta} \approx s_t^{\beta,\eta} - \beta \frac{\partial \mathcal{L}}{\partial s_{T-t}}$ and $\theta_{t+1}^{\beta,\eta} \approx \theta_t^{\beta,\eta} - \eta \frac{\partial \mathcal{L}}{\partial \theta_{T-t}}$, showing that in the second phase of C-EP, neurons and synapses descend the gradients of the loss $\mathcal{L}$

obtained with BPTT, with the hyperparameters β and η playing the role of learning rates for $s_t^{\beta,\eta}$ and $\theta_t^{\beta,\eta}$, respectively. Theorem 10 holds in the limit $\beta \to 0$, $\eta \to 0$, which means that β and η have to be small enough for the neurons and synapses to compute approximately the gradient of $\mathcal{L}^*$. On the other hand β and η also have to be large enough so that an error signal can be transmitted in the second phase and that optimization of the loss happens within a reasonable number of epochs. This trade-off is well reflected by the table of hyperparameters of Table 3.2 in Appendix 3.4.1. *In particular, the values $\beta = 0$ (there is no second phase) and (or) $\eta = 0$ (there is no learning) are excluded.*

2.5 Extending the GDD property: Continual Vector Field Equilibrium Propagation (C-VF)

The *Gradient Descending Dynamics* property (GDD, Theorem 10) states that, when the system dynamics derive from a primitive function, i.e. when the transition function F is of the form $F = \frac{\partial \Phi}{\partial s}$, then the normalized updates of C-EP match the gradients provided by BPTT. Remarkably, even in the case of dynamics that do not derive from a primitive function Φ, experiments in the next chapter show that the biologically plausible update rule of C-VF follows well the gradients of BPTT *. In this section, we give a theoretical justification for this fact by proving a more general result than Theorem 10. We call this version of Continual Equilibrium Propagation where the dynamics follow any transition function (or "vector field") F, *Continual Vector Field Equilibrium Propagation.*

Now let us consider general dynamics with transition function F the first phase rewrites:

$$s_{t+1} = F\left(x, s_t; \theta\right), \tag{2.28}$$

and the second phase:

$$\forall t \geq 0 : \qquad \begin{cases} s_{t+1}^{\beta,\eta} = F\left(x, s_t^{\beta,\eta}; \theta_t^{\beta,\eta}\right) - \beta \, \dfrac{\partial \ell}{\partial s}\left(s_t^{\beta,\eta}\right), \\ \theta_{t+1}^{\beta,\eta} = \theta_t^{\beta,\eta} + \dfrac{\eta}{\beta} \dfrac{\partial F}{\partial \theta}\left(x, s_t^{\beta,\eta}; \theta_t^{\beta,\eta}\right)^\top \cdot \left(s_{t+1}^{\beta,\eta} - s_t^{\beta,\eta}\right). \end{cases} \tag{2.29}$$

The definition of the normalized updates of C-VF is also:

$$\begin{cases} \Delta_s^{\mathrm{C-VF}}(\beta, \eta, t) = \dfrac{1}{\beta}\left(s_{t+1}^{\beta,\eta} - s_t^{\beta,\eta}\right), \\ \Delta_\theta^{\mathrm{C-VF}}(\beta, \eta, t) = \dfrac{1}{\eta}\left(\theta_{t+1}^{\beta,\eta} - \theta_t^{\beta,\eta}\right). \end{cases} \tag{2.30}$$

*More illustrations of this property are shown on Fig. 3.7 and Fig. 3.8.

We can now state the generalization of Theorem 10.

Theorem 11 (Generalisation of the GDD Property)**.** *Let $s_0, s_1, \ldots, s_T$ be the convergent sequence of states and denote $s_* = s_T$ the steady state. Further assume that there exists some step K where $0 < K \leq T$ such that $s_* = s_T = s_{T-1} = \ldots s_{T-K}$. Finally, assume that the Jacobian of the transition function at the steady state is symmetric, i.e. $\frac{\partial F}{\partial s}(x, s_*; \theta) = \frac{\partial F}{\partial s}(x, s_*; \theta)^\top$. Then, in the limit $\eta \to 0$ and $\beta \to 0$, the first K normalized updates of C-VF follow the the first K gradients of BPTT, i.e.*

$$\forall t = 0, 1, \ldots, K: \quad \begin{cases} \lim\limits_{\beta \to 0} \lim\limits_{\eta \to 0} \Delta_s^{\text{C-VF}}(\beta, \eta, t) = -\nabla_s^{\text{BPTT}}(t), \\ \lim\limits_{\beta \to 0} \lim\limits_{\eta \to 0} \Delta_\theta^{\text{C-VF}}(\beta, \eta, t) = -\nabla_\theta^{\text{BPTT}}(t). \end{cases} \tag{2.31}$$

Proof of Lemma 11. Defining:

$$\forall t = 0, 1, \ldots, K: \quad \begin{cases} \Delta_s^{\text{VF}}(\beta, t) = \lim\limits_{\eta \to 0} \Delta_s^{\text{C-VF}}(\beta, \eta, t), \\ \Delta_\theta^{\text{VF}}(\beta, t) = \lim\limits_{\eta \to 0} \Delta_\theta^{\text{C-VF}}(\beta, \eta, t). \end{cases} \tag{2.32}$$

We can readily show using Eq. (2.29) along with Eq. (2.30) that $\Delta_s^{\text{VF}}(\beta, t)$ and $\Delta_\theta^{\text{VF}}(\beta, t)$ satisfy the recurrence relationship:

$$\Delta_s^{\text{VF}}(\beta, 0) = -\frac{\partial \ell}{\partial s}(s_*, y), \tag{2.33}$$

$$\forall t \geq 0, \qquad \Delta_s^{\text{VF}}(\beta, t+1) = \frac{\partial F}{\partial s}(x, s_*, \theta) \cdot \Delta_s^{\text{C-VF}}(\beta, t), \tag{2.34}$$

$$\forall t \geq 0, \qquad \Delta_\theta^{\text{VF}}(\beta, t+1) = \frac{\partial F}{\partial \theta}(x, s_*, \theta)^\top \cdot \Delta_s^{\text{C-VF}}(\beta, t). \tag{2.35}$$

Again, using the exact same kind of reasoning than the one used for the demonstration of Theorem 4 in the previous part, since $\frac{\partial F}{\partial s}(x, s_*; \theta) = \frac{\partial F}{\partial s}(x, s_*; \theta)^\top$, Eq. (2.33)-(2.35) are the same as Eq. (3.4)-(3.6), so that $\Delta_s^{\text{VF}}(\beta, t)$ and $\Delta_\theta^{\text{VF}}(\beta, t)$ are equal to $-\nabla_s^{\text{BPTT}}(t)$ and $-\nabla_\theta^{\text{BPTT}}(t)$ at all time. □

Chapter 3

Models with symmetric and asymmetric weights

In this chapter, we validate our continual version of Equilibrium Propagation against training on the MNIST data set with two models. Following the terminology of section 4.1 in part IV, the first model is a prototypical RNN with tied and symmetric weights: the dynamics of this model approximately derive from a primitive function, which allows training with C-EP. The second model is a prototypical RNN with untied and asymmetric weights, which is therefore closer to biology. These two models belong to the class of prototypical models introduced in part IV, and we use them for this study to accelerate training simulations. We train this second model with C-VF which was previously introduced. In part IV, we showed with simulations the intuitive result that, if a model is such that the normalized updates of EP 'match' the gradients of BPTT (i.e. if they are approximately equal), then the model trained with EP performs as well as the model trained with BPTT. Along the same lines, we show in this work that the more the EP normalized updates follow the gradients of BPTT *before training* (which depends on the alignment between feedforward and feedback weights), the best is the resulting training performance.

3.1 Definition

Prototypical model with symmetric weights trained by C-EP. The first phase dynamics is defined as:

$$s_{t+1} = \sigma\left(W \cdot s_t + W_x \cdot x\right), \tag{3.1}$$

where σ is an activation function, W is a symmetric weight matrix, and W_x is a matrix connecting x to s. The same dynamics were introduced in Eq. (4.3), section 4.1 of part IV when defining the prototypical setting. Although the dynamics are not directly defined in terms of a primitive function, note that $s_{t+1} \approx \frac{\partial \Phi}{\partial s}\left(s_t, W\right)$ with $\Phi(s, W) = \frac{1}{2} s^\top \cdot W \cdot s$ if we

ignore the activation function σ. Following Eq. (1.3) and Eq. (2.1), we define the normalized updates of this model as:

$$\Delta_s^{\mathrm{C-EP}}(\beta,\eta,t) = \frac{1}{\beta}\left(s_{t+1}^{\beta,\eta} - s_t^{\beta,\eta}\right), \quad \Delta_W^{\mathrm{C-EP}}(\beta,\eta,t) = \frac{1}{\beta}\left(s_{t+1}^{\beta,\eta^\top} \cdot s_{t+1}^{\beta,\eta} - s_t^{\beta,\eta^\top} \cdot s_t^{\beta,\eta}\right). \quad (3.2)$$

Note that this model applies to any topology as long as existing connections have symmetric values: this includes deep networks with any number of layers. More explicitly, for a network whose layers of neurons are s^0, s^1, ..., s^N, with $W_{n,n+1}$ connecting the layers s^{n+1} and s^n in both directions, the corresponding primitive function is $\Phi = \sum_n (s^n)^\top \cdot W_{n,n+1} \cdot s_{n+1} + s^{N^\top} \cdot W_x \cdot x$.

Prototypical model with asymmetric weights trained by C-VF. In this model, the dynamics in the first phase is the same as Eq. (3.1) but now the weight matrix W is no longer assumed to be symmetric. In this setting the weight dynamics in the second phase is replaced by a version for asymmetric weights: $W_{t+1}^{\beta,\eta} = W_t^{\beta,\eta} + \frac{\eta}{\beta} s_t^{\beta,\eta^\top} \cdot \left(s_{t+1}^{\beta,\eta} - s_t^{\beta,\eta}\right)$, so that the normalized updates are equal to:

$$\Delta_s^{\mathrm{C-VF}}(\beta,\eta,t) = \frac{1}{\beta}\left(s_{t+1}^{\beta,\eta} - s_t^{\beta,\eta}\right), \qquad \Delta_W^{\mathrm{C-VF}}(\beta,\eta,t) = \frac{1}{\beta} s_t^{\beta,\eta^\top} \cdot \left(s_{t+1}^{\beta,\eta} - s_t^{\beta,\eta}\right). \quad (3.3)$$

Like the previous model, the prototypical RNN with asymmetric weights also applies to deep networks with any number of layers. Although in C-VF the dynamics of the weights is not one of the form of Eq. (1.3) that derives from a primitive function, the (bioplausible) normalized weight updates of Eq. (3.3) can approximately follow the gradients of BPTT, provided that the values of reciprocal connections are not too dissimilar: this is a consequence of Theorem 11. Indeed, defining the transition function $F(s,W) = \sigma(W \cdot s + W_x \cdot x)$, so that the dynamics of the first phase (Eq. (3.1)) rewrite:

$$s_{t+1} = F\left(x, s_t; W, W_x\right). \quad (3.4)$$

As for the second phase, notice that $\frac{\partial F}{\partial W}(x,s;W,W_x) = \sigma'(W \cdot s + W_x \cdot x) \cdot s$, so that if we ignore the factor $\sigma'(W \cdot s + W_x \cdot x)$, the second phase rewrites:

$$\forall t \geq 0 : \quad \begin{cases} s_{t+1}^{\beta,\eta} = F\left(x, s_t^{\beta,\eta}; W_t^{\beta,\eta}, W_{x,t}^{\beta,\eta}\right) - \beta \, \dfrac{\partial \ell}{\partial s}\left(s_t^{\beta,\eta}\right), \\ W_{t+1}^{\beta,\eta} = W_t^{\beta,\eta} + \dfrac{\eta}{\beta} \dfrac{\partial F}{\partial W}\left(x, s_t^{\beta,\eta}; W_t^{\beta,\eta}, W_{x,t}^{\beta,\eta}\right)^\top \cdot \left(s_{t+1}^{\beta,\eta} - s_t^{\beta,\eta}\right). \end{cases} \quad (3.5)$$

Also observe that:

$$\frac{\partial F}{\partial s}(x,s;W,W_x) = \sigma'(W \cdot s + W_x \cdot x) \cdot W^\top. \quad (3.6)$$

Ignoring the factor $\sigma'(W \cdot s + W_x \cdot x)$ again, we see that if W is symmetric then the Jacobian of F is also symmetric, therefore the conditions of Theorem 11 are met. This observation is illustrated in Fig. 4.1 (as well as in Fig. 3.7 and Fig. 3.8).

3.1.1 Models under consideration

In the rest of this part, we study the following models:

- The real-time model with symmetric weights trained by C-EP (subsection 3.2.1). This model corresponds to the energy-based model introduced in subsubsection 4.3.2 of part IV, where the second phase has been adapted for C-EP training.
- The prototypical model with symmetric weights trained by C-EP (subsection 3.2.2). This model corresponds to the prototypical model introduced in subsubsection 4.4.1 of part IV, where the second phase has also been adapted for C-EP training.
- The real-time model with asymmetric weights trained by C-VF (subsection 3.3.1). This model is inspired from [147] - see section 2.2 in chapter III.
- The prototypical model with asymmetric weights trained by C-VF (subsection 3.3.2).

3.1.2 Figures for the GDD experiments

For each of the model described below, we show the effect of using continual updates with a finite learning rate in terms of the $\Delta^{\text{C-EP}}$ and $-\nabla^{\text{BPTT}}$ processes. These figures have been realized for each of the models with one hidden layer on MNIST. Dashed an continuous lines respectively represent the normalized updates Δ and the gradients ∇^{BPTT}. Each randomly selected synapse or neuron correspond to one color. We add an s or θ index to specify whether we analyse neuron or synapse updates and gradients. Each C-VF simulation has been realized with an angle between forward and backward weights of 0 degrees (i.e. $\Psi(\theta_{\text{f}}, \theta_{\text{b}}) = 0°$) - see Appendix 3.4.1 for a precise definition of $\Psi(\theta_{\text{f}}, \theta_{\text{b}})$.

3.2 Models with symmetric weights trained by C-EP

3.2.1 Real-time (energy-based) model

Equations. In this subsection (as in subsection 4.3.2 of the previous part), we denote N the number of hidden layers, so that in general, $s^1, s^2, \cdots, s^N$ stand for the hidden layers and $s^{N+1} = \hat{y}$ is the output layer.

As in subsection 4.3.2, we consider the following primitive function:

$$\Phi\left(x, s^1, s^2, \ldots, s^{N+1} = \hat{y}\right) = \frac{1}{2}(1-\varepsilon)\left(\sum_{n=1}^{N+1} ||s^n||^2\right) + \varepsilon \sum_{n=1}^{N} \sigma(s^{n+1})^\top \cdot w_n \cdot \sigma(s^n) + \sigma(s^1) \cdot w_0 \cdot \sigma(x) \tag{3.7}$$

so that the equations of motion for the first phase read:

$$\forall t \in [0,T] : \begin{cases} \hat{y}_{t+1} &= (1-\varepsilon)\hat{y}_t + \varepsilon\sigma'(\hat{y}_t) \odot w_N \cdot \sigma(s_t^N) \\ s_{t+1}^n &= (1-\varepsilon)s_t^n + \sigma'(s_t^n) \odot \varepsilon(w_{n-1} \cdot \sigma\left(s_t^{n-1}\right) + w_n^\top \cdot \sigma(s_t^{n+1})) \qquad \forall n \in [2,N] \\ s_{t+1}^1 &= (1-\varepsilon)s_t^1 + \varepsilon\sigma'(s_t^1) \odot (w_0 \cdot \sigma\left(x\right) + w_1^\top \cdot \sigma(s_t^2)) \end{cases},$$

During the second phase,

$$\forall t \in [0,T] : \begin{cases} \hat{y}_{t+1}^{\beta,\eta} &= (1-\varepsilon)\hat{y}_t^{\beta,\eta} + \varepsilon\sigma'(\hat{y}_t^{\beta,\eta}) \odot w_N \cdot \sigma(s_t^{N,\beta,\eta}) + \beta\varepsilon(y - \hat{y}_t^{\beta,\eta}) \\ s_{t+1}^{n,\beta,\eta} &= (1-\varepsilon)s_t^{n,\beta,\eta} + \sigma'(s_t^{n,\beta,\eta}) \odot \varepsilon(w_{n-1} \cdot \sigma\left(s_t^{n-1,\beta,\eta}\right) + w_n^\top \cdot \sigma(s_t^{n+1,\beta,\eta})) \\ & \forall n \in [2,N], \\ s_{t+1}^{1,\beta,\eta} &= (1-\varepsilon)s_t^{1,\beta,\eta} + \varepsilon\sigma'(s_t^{1,\beta,\eta}) \odot (w_0 \cdot \sigma\left(x\right) + w_1^\top \cdot \sigma(s_t^{2,\beta,\eta})) \\ \theta_{t+1}^{\beta,\eta} &= \theta_t^{\beta,\eta} + \eta\Delta_\theta^{\mathrm{EP}}(\beta,\eta,t) \qquad \forall \theta \in \{w_n\} \end{cases}$$

According to Eq. (2.3) and Eq. (3.7), we have:

$$\begin{cases} \Delta_{w_0}^{\mathrm{EP}}(\beta,\eta,t) &= \frac{1}{\beta}\left(\sigma\left(s_{t+1}^{1,\beta,\eta}\right) \cdot \sigma\left(x\right)^\top - \sigma\left(s_t^{1,\beta,\eta}\right) \cdot \sigma\left(x\right)^\top\right) \\ \Delta_{w_n}^{\mathrm{EP}}(\beta,\eta,t) &= \frac{1}{\beta}\left(\sigma\left(s_{t+1}^{n+1,\beta,\eta}\right) \cdot \sigma\left(s_{t+1}^{n,\beta,\eta}\right)^\top - \sigma\left(s_t^{n+1,\beta,\eta}\right) \cdot \sigma\left(s_t^{n,\beta,\eta}\right)^\top\right) \qquad \forall n \in [1,N] \end{cases}$$

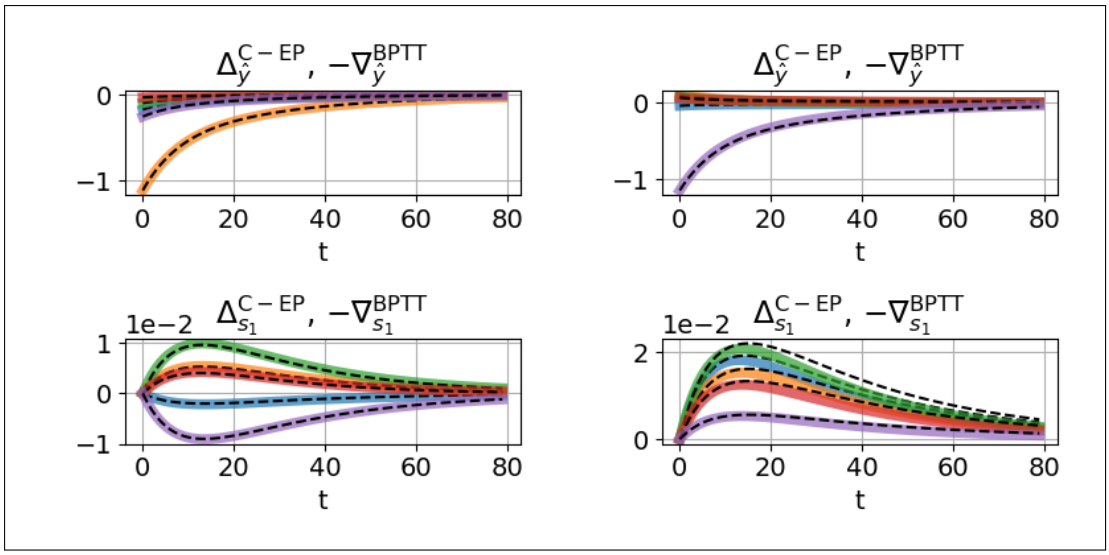

Figure 3.1: Real-Time RNN with symmetric weights. **Left**: $\Delta_s^{\mathrm{C-EP}}(t)$ normalized updates ($\eta \sim 10^{-6}$) and $-\nabla_s^{\mathrm{BPTT}}(t)$ gradients. **Right**: $\Delta_s^{\mathrm{C-EP}}(t)$ normalized updates ($\eta \sim 10^{-5}$) and $-\nabla_s^{\mathrm{BPTT}}(t)$ gradients.

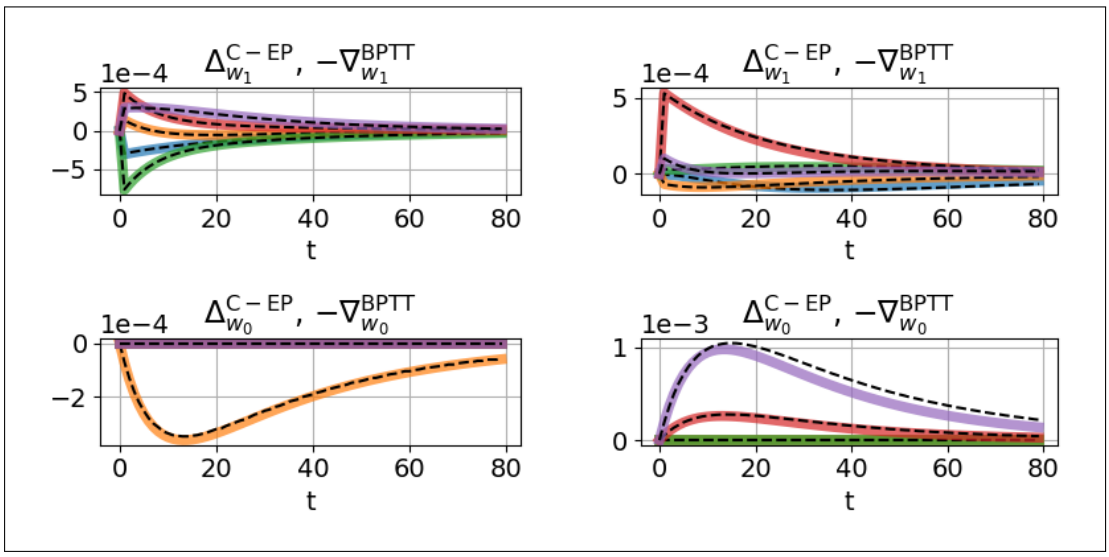

Figure 3.2: Real-Time RNN with symmetric weights. **Left**: $\Delta_\theta^{\mathrm{C-EP}}(t)$ normalized updates ($\eta \sim 10^{-6}$) and $-\nabla_\theta^{\mathrm{BPTT}}(t)$ gradients. **Right**: $\Delta_\theta^{\mathrm{C-EP}}(t)$ normalized updates ($\eta \sim 10^{-5}$) and $-\nabla_\theta^{\mathrm{BPTT}}(t)$ gradients.

3.2.2 Protypical model

Equations. Similarly to subsection 4.3.2, we consider a discrete-time model where the dynamics of the first phase are defined as:

$$\forall t \in [0,T] : \begin{cases} \hat{y}_{t+1} = \sigma\left(w_N \cdot s_t^N\right) \\ s_{t+1}^n = \sigma\left(w_{n-1} \cdot s_t^{n-1} + w_n^\top \cdot s_t^{n+1}\right) & \forall n \in [2,N] \\ s_{t+1}^1 = \sigma\left(w_0 \cdot x + w_1^\top \cdot s_t^2\right). \end{cases} \tag{3.8}$$

Again, we remind here that defining:

$$\Phi(x, s^1, \dots, \hat{y}) = \sum_{n=1}^{N} {s^{n+1}}^\top \cdot w_n \cdot s^n + s^1 \cdot w_0 \cdot x, \tag{3.9}$$

ignoring the activation function σ, Eq. (3.8) rewrites:

$$s_{t+1}^n \approx \frac{\partial \Phi}{\partial s^n}(x, s^1, \dots, \hat{y}) \qquad \forall n \in [1, N+1] \tag{3.10}$$

Thereby, applying Eq. (2.3) along with Eq. (3.9), the dynamics of the second phase read:

$$\forall t \in [0,T] : \begin{cases} \hat{y}_{t+1}^{\beta,\eta} = \sigma\left(w_N \cdot s_t^{N,\beta,\eta}\right) + \beta(y - \hat{y}_t^{\beta,\eta}) \\ s_{t+1}^{n,\beta,\eta} = \sigma\left(w_{n-1} \cdot s_t^{n-1,\beta,\eta} + w_n^\top \cdot s_t^{n+1,\beta,\eta}\right) & \forall n \in [2,N] \\ s_{t+1}^{1,\beta,\eta} = \sigma\left(w_0 \cdot x + w_1^\top \cdot s_t^{2,\beta,\eta}\right), \\ \theta_{t+1}^{\beta,\eta} = \theta_t^{\beta,\eta} + \eta \Delta_\theta^{\mathrm{EP}}(\beta,\eta,t) & \forall \theta \in \{w_n\} \end{cases} \tag{3.11}$$

where for every layer w_n and every $t \in [0,K]$:

$$\begin{cases} \Delta_{w_0}^{\mathrm{EP}}(\beta,\eta,t) = \frac{1}{\beta}\left(s_{t+1}^{1,\beta,\eta} \cdot x^\top - s_t^{1,\beta,\eta} \cdot x^\top\right), \\ \Delta_{w_n}^{\mathrm{EP}}(\beta,\eta,t) = \frac{1}{\beta}\left(s_{t+1}^{n+1,\beta,\eta} \cdot s_{t+1}^{n,\beta,\eta^\top} - s_t^{n+1,\beta,\eta} \cdot s_t^{n,\beta,\eta^\top}\right) & \forall n \in [1,N] \end{cases} \tag{3.12}$$

Defining $s = (s^1, s^2, \dots, \hat{y})^\top$ and taking again W and W_x as defined in Eq. (4.11) in the previous part, Eq. (3.8) can also be vectorized in a block-wise fashion as Eq. (3.1).

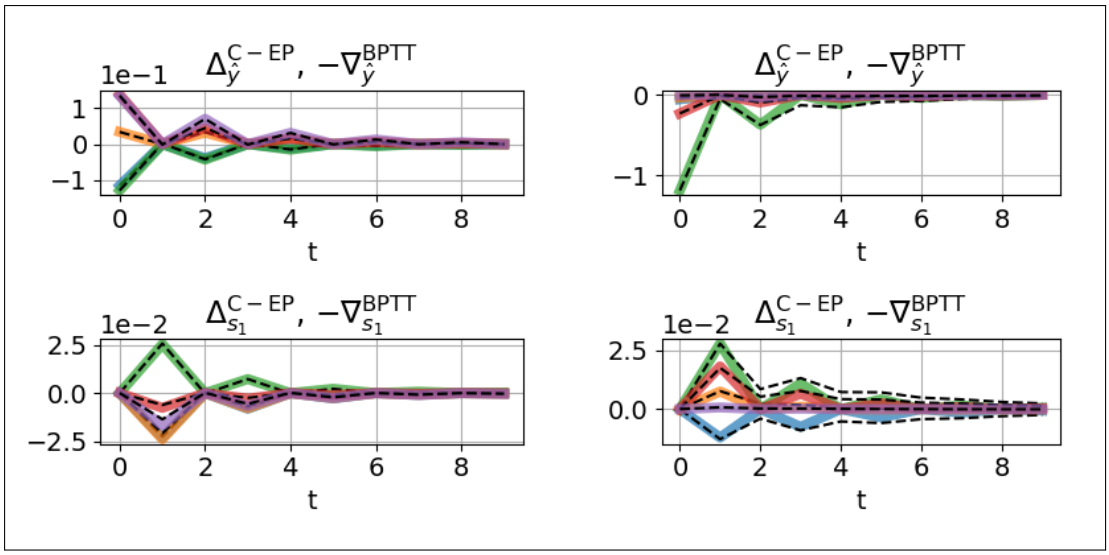

Figure 3.3: Prototypical model with symmetric weights. **Left**: $\Delta_s^{\mathrm{C-EP}}(t)$ normalized updates ($\eta \sim 10^{-6}$) and $-\nabla_s^{\mathrm{BPTT}}(t)$ gradients. **Right**: $\Delta_s^{\mathrm{C-EP}}(t)$ normalized updates ($\eta \sim 10^{-5}$) and $-\nabla_s^{\mathrm{BPTT}}(t)$ gradients.

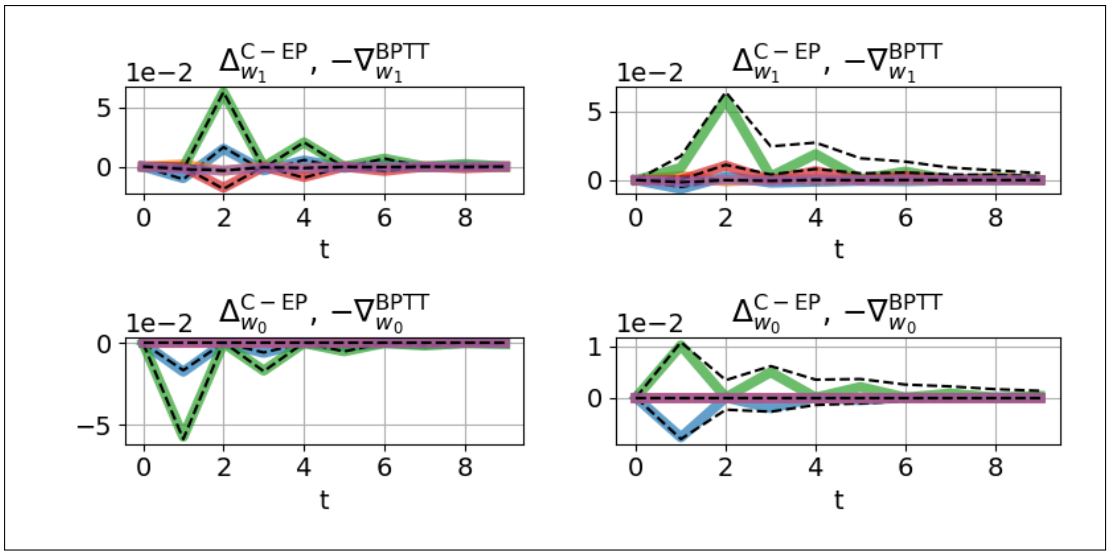

Figure 3.4: Prototypical model with symmetric weights. **Left**: $\Delta_\theta^{\mathrm{C-EP}}(t)$ normalized updates ($\eta \sim 10^{-6}$) and $-\nabla_\theta^{\mathrm{BPTT}}(t)$ gradients. **Right**: $\Delta_\theta^{\mathrm{C-EP}}(t)$ normalized updates ($\eta \sim 10^{-5}$) and $-\nabla_\theta^{\mathrm{BPTT}}(t)$ gradients.

3.3 Models with asymmetric weights trained by C-VF

In this section, we consider models with asymmetric connections: $w_{m,n}$ stands for the synapses connecting the layer s^n to the layer s^m.

3.3.1 Real-time model

Equations. For this model, the dynamics of the first phase are defined as:

$$\forall t \in [0,T]: \begin{cases} \hat{y}_{t+1} = (1-\varepsilon)\hat{y}_t + \varepsilon w_{\hat{y},N} \cdot \sigma\left(s_t^N\right) \\ s_{t+1}^n = (1-\varepsilon)s_t^n + \varepsilon\left(w_{n,n-1} \cdot \sigma\left(s_t^{n-1}\right) + w_{n,n+1} \cdot \sigma\left(s_t^{n+1}\right)\right) & \forall n \in [2,N] \\ s_{t+1}^1 = (1-\varepsilon)s_t^1 + \varepsilon\left(w_0 \cdot \sigma\left(x\right) + w_{1,2} \cdot \sigma\left(s_t^2\right)\right) \end{cases}$$

where ε is the time-discretization parameter.

Using Eq. (2.29) applied to these dynamics to derive the weight updates, the dynamics of the second phase read:

$$\forall t \in [0,K]: \begin{cases} \hat{y}_{t+1}^{\beta,\eta} = (1-\varepsilon)\hat{y}_t^{\beta,\eta} + \varepsilon w_{\hat{y},N} \cdot \sigma\left(s_t^{N,\beta,\eta}\right) + \beta\varepsilon(y - \hat{y}^{\beta,\eta}) \\ s_{t+1}^{n,\beta,\eta} = (1-\varepsilon)s_t^{n,\beta,\eta} + \varepsilon\left(w_{n,n-1} \cdot \sigma\left(s_t^{n-1,\beta,\eta}\right) + w_{n,n+1} \cdot \sigma\left(s_t^{n+1,\beta,\eta}\right)\right) \\ \qquad \forall n \in [2,N] \\ s_{t+1}^{1,\beta,\eta} = (1-\varepsilon)s_t^{1,\beta,\eta} + \varepsilon\left(w_0 \cdot \sigma\left(x\right) + w_{1,2} \cdot \sigma\left(s_t^{2,\beta,\eta}\right)\right) \\ \theta_{t+1}^{\beta,\eta} = \theta_t^{\beta,\eta} + \eta\, \Delta_\theta^{\mathrm{C-VF}}(\beta,\eta,t) \qquad \forall \theta \in \{w_{nn+1}, w_{n+1n}\} \end{cases} \tag{3.13}$$

where for every time step $t \in [0,K]$

$$\begin{cases} \Delta_{w_0}^{\mathrm{C-VF}}(\beta,\eta,t) = \frac{1}{\beta}(s_{t+1}^{1,\beta,\eta} - s_t^{1,\beta,\eta}) \cdot \sigma(x) \\ \Delta_{w_{n+1,n}}^{\mathrm{C-VF}}(\beta,\eta,t) = \dfrac{1}{\beta}\left(s_{t+1}^{n+1,\beta,\eta} - s_t^{n+1,\beta,\eta}\right) \cdot \sigma\left(s_t^{n,\beta,\eta}\right)^\top \\ \Delta_{w_{n,n+1}}^{\mathrm{C-VF}}(\beta,\eta,t) = \dfrac{1}{\beta}\left(s_{t+1}^{n,\beta,\eta} - s_t^{n,\beta,\eta}\right) \cdot \sigma\left(s_t^{n+1,\beta,\eta}\right)^\top \end{cases}$$

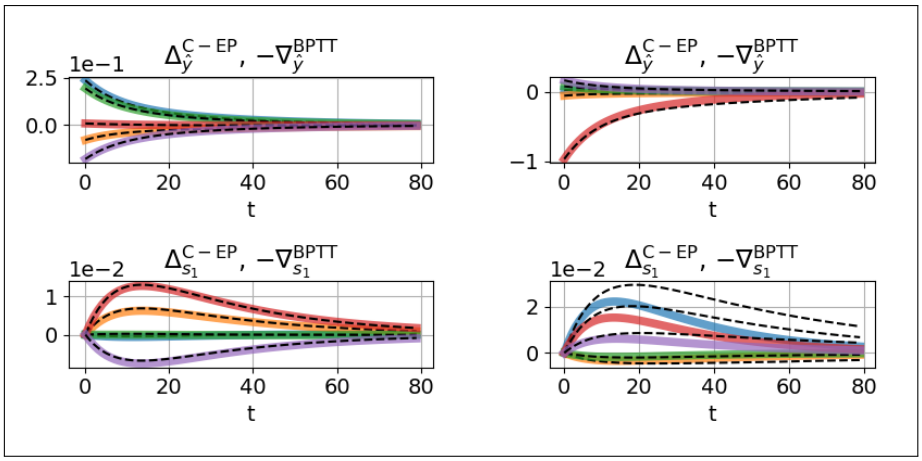

Figure 3.5: Real-Time RNN with asymmetric weights. **Left**: $\Delta_s^{\mathrm{C-VF}}(t)$ normalized updates ($\eta \sim 10^{-6}$) and $-\nabla_s^{\mathrm{BPTT}}(t)$ gradients. **Right**: $\Delta_s^{\mathrm{C-VF}}(t)$ normalized updates ($\eta \sim 10^{-5}$) and $-\nabla_s^{\mathrm{BPTT}}(t)$ gradients.

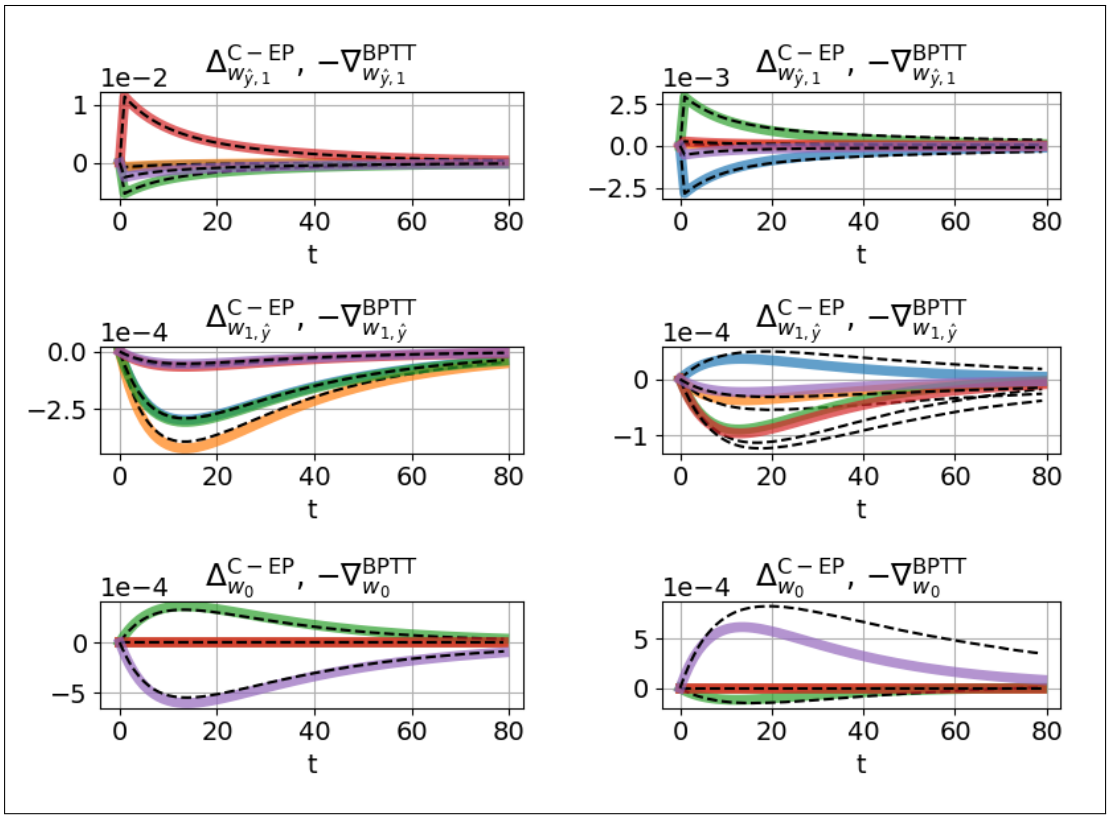

Figure 3.6: Real-Time RNN with asymmetric weights. **Left**: $\Delta_\theta^{\mathrm{C-VF}}(t)$ normalized updates ($\eta \sim 10^{-6}$) and $-\nabla_\theta^{\mathrm{BPTT}}(t)$ gradients. **Right**: $\Delta_\theta^{\mathrm{C-VF}}(t)$ normalized updates ($\eta \sim 10^{-5}$) and $-\nabla_\theta^{\mathrm{BPTT}}(t)$ gradients.

3.3.2 Prototypical model

Equations. For this model, the dynamics of the first phase are defined as:

$$\forall t \in [0,T] : \left\{ \begin{array}{ll} \hat{y}_{t+1} & = \sigma(w_{\hat{y},N} \cdot s_t^N) \\ s_{t+1}^n & = \sigma(w_{n,n-1} \cdot s_t^{n-1} + w_{n,n+1} \cdot s_t^{n+1}) \qquad \forall n \in [2,N] \\ s_{t+1}^1 & = \sigma\left(w_0 \cdot x + w_{1,2} \cdot s_t^2\right), \end{array} \right. \tag{3.14}$$

and the second phase as:

$$\forall t \in [0,K] : \left\{ \begin{array}{ll} \hat{y}_{t+1}^{\beta,\eta} & = \sigma(w_{\hat{y},N} \cdot s_t^{N,\beta,\eta}) \\ s_{t+1}^{n,\beta,\eta} & = \sigma(w_{n,n-1} \cdot s_t^{n-1,\beta,\eta} + w_{n,n+1} \cdot s_t^{n+1,\beta,\eta}) \qquad \forall n \in [2,N] \\ s_{t+1}^{1,\beta,\eta} & = \sigma\left(w_0 \cdot x + w_{1,2} \cdot s_t^{2,\beta,\eta}\right) \\ \theta_{t+1}^{\beta,\eta} & = \theta_t^{\beta,\eta} + \eta \Delta_\theta^{\text{C-VF}}(\beta,\eta,t) \qquad \forall \theta \in \{W_{nn+1}, W_{n+1n}\} \end{array} \right. \tag{3.15}$$

Note that Eq. (3.14) can also be in a vectorized block-wise fashion as Eq. (3.1) with $s = (s^0, s^1, \ldots, s^{N-1})^\top$ and provided that we define W and W_x as:

$$W = \begin{pmatrix} 0 & w_{1,2} & 0 & 0 & 0 \\ w_{2,1} & 0 & w_{2,3} & 0 & 0 \\ 0 & w_{3,2} & 0 & \ddots & 0 \\ 0 & 0 & \ddots & 0 & w_{N,\hat{y}} \\ 0 & 0 & 0 & w_{\hat{y},N} & 0 \end{pmatrix}, \qquad W_x = \begin{pmatrix} w_0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \tag{3.16}$$

For all layers $w_{n,n+1}$ and $w_{n+1,n}$, and every $t \in [0,K]$, we define the weight updates as:

$$\left\{ \begin{array}{ll} \Delta_{w_0}^{\text{C-VF}}(\beta,\eta,t) & = \frac{1}{\beta}(s_{t+1}^{1,\beta,\eta} - s_t^{1,\beta,\eta}) \cdot x \\ \Delta_{w_{n+1,n}}^{\text{C-VF}}(\beta,\eta,t) & = \frac{1}{\beta}(s_{t+1}^{n+1,\beta,\eta} - s_t^{n+1,\beta,\eta}) \cdot s_t^{n,\beta,\eta^\top} \\ \Delta_{w_{n,n+1}}^{\text{C-VF}}(\beta,\eta,t) & = \frac{1}{\beta}(s_{t+1}^{n,\beta,\eta} - s_t^{n,\beta,\eta}) \cdot s_t^{n+1,\beta,\eta^\top} \end{array} \right.$$

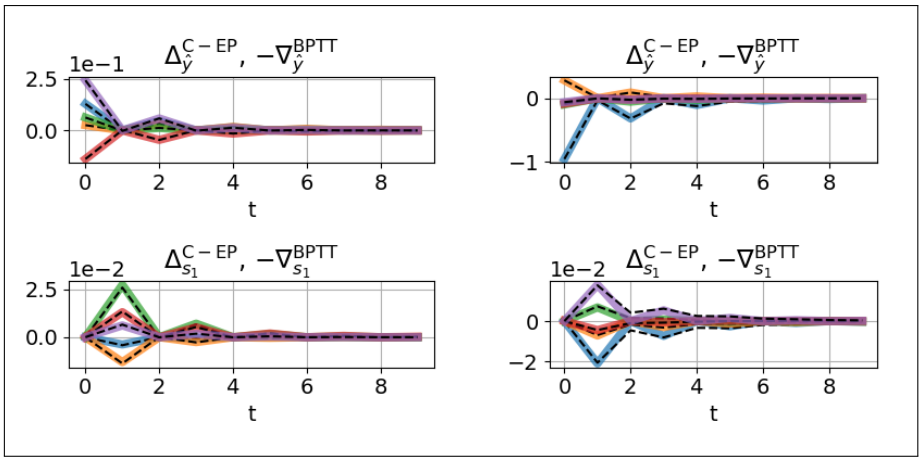

Figure 3.7: Prototypical model with asymmetric weights. **Left**: $\Delta_s^{\mathrm{C-VF}}(t)$ normalized updates ($\eta \sim 10^{-6}$) and $-\nabla_s^{\mathrm{BPTT}}(t)$ gradients. **Right**: $\Delta_s^{\mathrm{C-VF}}(t)$ normalized updates ($\eta \sim 10^{-5}$) and $-\nabla_s^{\mathrm{BPTT}}(t)$ gradients.

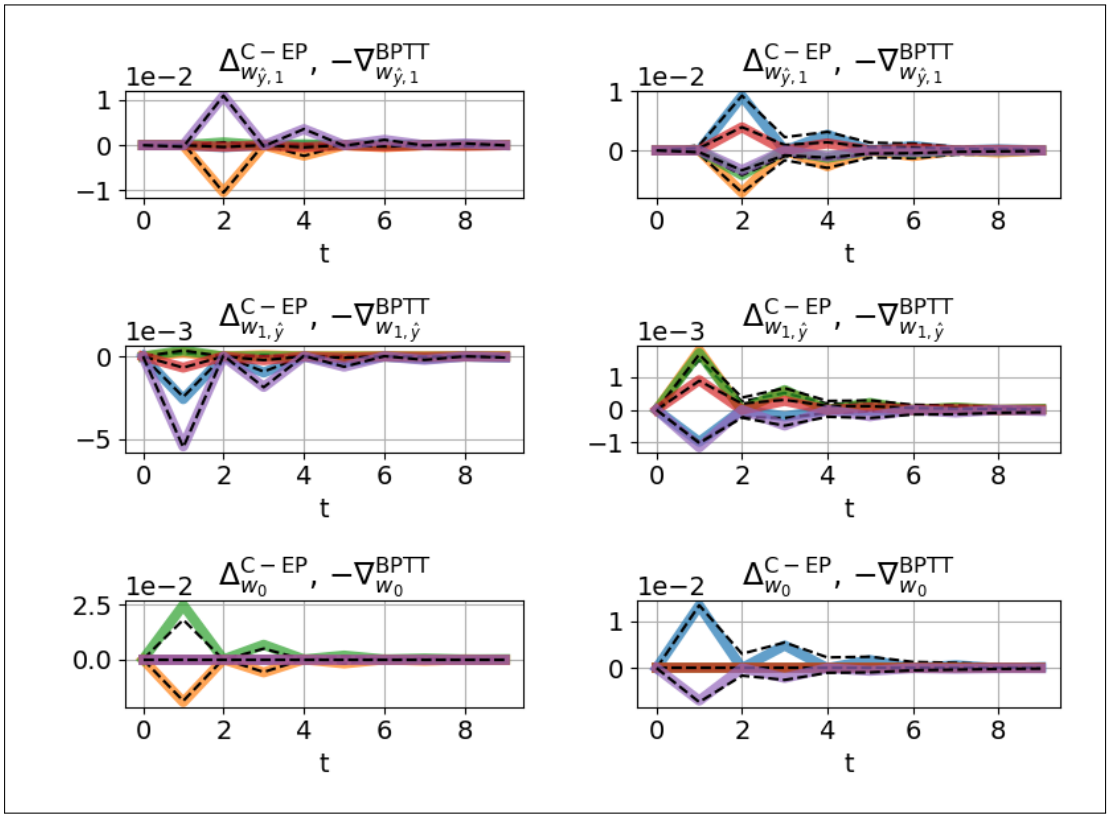

Figure 3.8: Prototypical model with asymmetric weights. **Left**: $\Delta_\theta^{\mathrm{C-VF}}(t)$ normalized updates ($\eta \sim 10^{-6}$) and $-\nabla_\theta^{\mathrm{BPTT}}(t)$ gradients. **Right**: $\Delta_\theta^{\mathrm{C-VF}}(t)$ normalized updates ($\eta \sim 10^{-5}$) and $-\nabla_\theta^{\mathrm{BPTT}}(t)$ gradients.

Chapter 4

Training experiments

4.1 C-EP training experiments

Table 4.1: Training results on MNIST with EP, C-EP and C-VF. "#h" stands for the number of hidden layers. We indicate over 5 trials the mean and standard deviation for the test error (mean train error in parenthesis). T (resp. K) is the number of iterations in the 1st (resp. 2nd) phase. For C-VF results, the initial angle between forward (θ_{f}) and backward (θ_{b}) weights is $\Psi(\theta_{\mathrm{f}}, \theta_{\mathrm{b}}) = 0°$.

	Error (%)		T	K	Epochs
	Test	Train			
EP-1h	2.00 ± 0.13	(0.20)	30	10	30
EP-2h	1.95 ± 0.10	(0.14)	100	20	50
C-EP-1h	$\mathbf{2.28 \pm 0.16}$	(0.41)	40	15	100
C-EP-2h	$\mathbf{2.44 \pm 0.14}$	(0.31)	100	20	150
C-VF-1h	$\mathbf{2.43 \pm 0.08}$	(0.77)	40	15	100
C-VF-2h	$\mathbf{2.97 \pm 0.19}$	(1.58)	100	20	150

Experiments are first performed with multi-layered prototypical RNNs (with symmetric weights) on MNIST. Table 4.1 presents the results obtained with C-EP training benchmarked against standard EP training results of part IV - see Appendix 3.4.1 for training conditions. The test error of C-EP approaches that of EP, with a slight degradation in accuracy. This is because although Theorem 10 guarantees Gradient Descending Dynamics (GDD) in the limit of infinitely small learning rates, in practice we have to strike a balance between having a learning rate that is small enough to ensure this condition but not too small to observe convergence within a reasonable number of epochs. As seen in Fig. 4.1 (b), the finite learning

rate η of continual updates leads to $\Delta^{\text{C}-\text{EP}}(\beta, \eta, t)$ curves splitting apart from the $-\nabla^{\text{BPTT}}(t)$ curves. As seen per Fig. 4.1 (a), this effect is emphasized with the depth: before training, angles between the normalized updates of C-EP and the gradients of BPTT reach 50 degrees for two hidden layers. The deeper the network, the more difficult it is for the C-EP dynamics to follow the gradients provided by BPTT. As an evidence, we show in section 4.2 that when we use extremely small learning rates throughout the second phase ($\theta \leftarrow \theta + \eta_{\text{tiny}} \Delta_\theta^{\text{C}-\text{EP}}$) and rescale up the resulting total weight update ($\theta \leftarrow \theta - \Delta\theta_{\text{tot}} + \frac{\eta}{\eta_{\text{tiny}}} \Delta\theta_{\text{tot}}$), we recover standard EP results.

4.2 Why C-EP does not perform as well as standard EP?

We provide here further ground for the training performance degradation observed on the MNIST task when implementing C-EP compared to standard EP. In practice, when training with C-EP, we have to make a trade-off between:

1. having a learning rate that is small enough so that C-EP normalized updates are subsequently close enough to the gradients of BPTT (Theorem 10),
2. having a learning rate that is large enough to ensure convergence within a reasonable number of epochs.

In other words, the degradation of accuracy observed in the table of Fig. 4.1 is due to using a learning rate that is too large to observe convergence within 100 epochs. To demonstrate this, we implement Alg. 8 which consists simply in using a very small learning rate throughout the second phase (denoted as η_{tiny}), and artificially rescaling the resulting weight update by a bigger learning rate (denoted as η). Applying Alg. 8 to a fully connected layered architecture with one hidden layer, $T = 30$, $K = 10$, $\beta = 0.1$, yields $2.06 \pm 0.13\%$ test error and $0.18 \pm 0.01\%$ train error over 5 trials, where we indicate mean and standard deviation. Similarly, applying Alg. 8 to a fully connected layered architecture with two hidden layers, $T = 100$, $K = 20$, $\beta = 0.5$, yields $1.89 \pm 0.22\%$ test error and $0.02 \pm 0.02\%$ train error. These results exactly match the ones provided by standard EP - see Table 3.3.

4.3 Continual Vector Field (C-VF) training experiments

Depending on whether the updates occur continuously during the second phase and the system obey general dynamics with untied forward and backward weights, we can span a large range of deviations from the ideal conditions of Theorem 10. Fig. 4.1 (b) depicts qualitatively these deviations with a model for which the normalized updates of EP match the gradients of BPTT (EP); with continual weight updates, the normalized updates and gradients start splitting apart (C-EP), and even more so if the weights are untied (C-VF).

Algorithm 8 Debugging procedure of C-EP

Input: x, y, θ, β, η, $\eta_{\text{tiny}} = 10^{-5}\eta$.
Output: θ.

$s_0 \leftarrow 0$ ▷ First Phase
$\Delta\theta \leftarrow 0$ ▷ Temporary variable accumulating parameter updates
repeat
    $s_{t+1} \leftarrow \frac{\partial \Phi}{\partial s}\left(x, s_t, \theta\right)$
until $s_t = s_*$
$s_0^\beta \leftarrow s_*$ ▷ Second Phase
repeat
    $s_{t+1}^\beta \leftarrow \frac{\partial \Phi}{\partial s}\left(x, s_t^\beta, \theta\right) - \beta \frac{\partial \ell}{\partial s}\left(s_t^\beta, y\right)$

    $\theta \leftarrow \theta + \frac{\eta_{\text{tiny}}}{\beta}\left(\frac{\partial \Phi}{\partial \theta}\left(s_{t+1}^\beta\right) - \frac{\partial \Phi}{\partial \theta}\left(s_t^\beta\right)\right)$
    $\Delta\theta \leftarrow \Delta\theta + \frac{\eta_{\text{tiny}}}{\beta}\left(\frac{\partial \Phi}{\partial \theta}\left(s_{t+1}^\beta\right) - \frac{\partial \Phi}{\partial \theta}\left(s_t^\beta\right)\right)$
until s_t^β and θ are converged.
$\theta \leftarrow \theta - \Delta\theta + \frac{\eta}{\eta_{\text{tiny}}}\Delta\theta$ ▷ Rescale the total parameter update by $\frac{\eta}{\eta_{\text{tiny}}}$

Protocol. In order to create these deviations from Theorem 10 and study the consequences in terms of training, we proceed as follows. For each C-VF simulations, we tune the initial angle between forward weights (θ_{f}) and backward weights (θ_{b}) between 0 and 180°. We denote this angle $\Psi(\theta_{\text{f}}, \theta_{\text{b}})$ - see Appendix 3.4.1 for the angle definition and the angle tuning technique employed. For each of these weight initialization, we compute the angle between the *total* normalized update provided by C-VF, i.e. $\Delta^{\text{C-VF}}(\beta, \eta, \text{tot}) = \sum_{t=0}^{K-1} \Delta^{\text{C-VF}}(\beta, \eta, t)$ and the *total* gradient provided by BPTT, i.e. $\nabla^{\text{BPTT}}(\text{tot}) = \sum_{t=0}^{K-1} \nabla^{\text{BPTT}}(t)$ on random mini-batches *before training*. We denote this angle $\Psi\left(\Delta^{\text{C-VF}}(\text{tot}), -\nabla^{\text{BPTT}}(\text{tot})\right)$. Finally for each weight initialization, we perform training on the prototypical models previously introduced. We proceed in the same way for EP and C-EP simulations, computing $\Psi\left(\Delta^{\text{EP}}(\text{tot}), -\nabla^{\text{BPTT}}(\text{tot})\right)$ and $\Psi\left(\Delta^{\text{C-EP}}(\text{tot}), -\nabla^{\text{BPTT}}(\text{tot})\right)$ before training. We use the generic notation $\Delta(\text{tot})$ to denote the total normalized update. This procedure yields (x, y) data points with $x = \Psi\left(\Delta(\text{tot}), -\nabla^{\text{BPTT}}(\text{tot})\right)$ and $y =$ test error, which are reported on Fig. 4.1 (a) - see Table 3.3 of Appendix 3.4.1 for the full table of results.

Results. Fig. 4.1 (a) shows the test error achieved on MNIST by EP, C-EP on the prototypical model with symmetric weights and C-VF on the prototypical model with asymmetric weights for different number of hidden layers as a function of the angle $\Psi\left(\Delta(\text{tot}), -\nabla^{\text{BPTT}}(\text{tot})\right)$ before training. This graphical representation spreads the algorithms between EP which best satisfies the GDD property (leftmost point in green at $\sim 20°$) to C-VF which satisfies the less the GDD property (rightmost points in red and orange at $\sim 100°$). As expected, high angles between gradients of C-VF and BPTT lead to high error rates that can reach 90% for $\Psi\left(\Delta^{\text{C-VF}}(\text{tot}), -\nabla^{\text{BPTT}}(\text{tot})\right)$ over 100°. More precisely, the inset of Fig. 4.1 shows the same data but focusing only on results generated by initial weight angles lying below 90°,

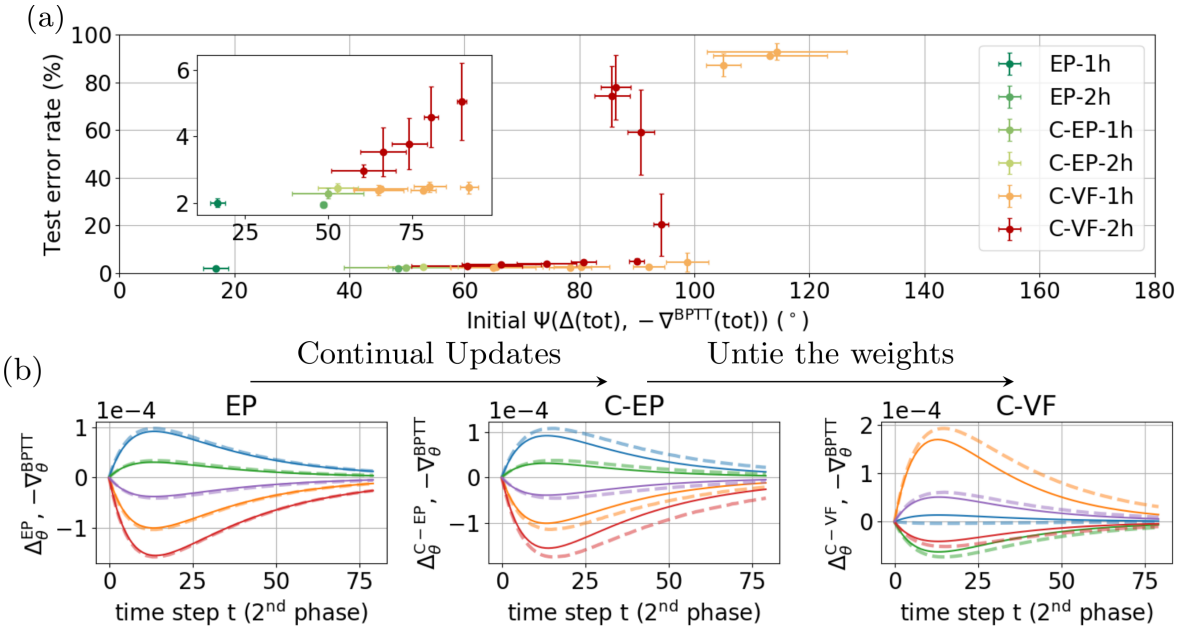

Figure 4.1: **Three versions of EP:** standard Equilibrium Propagation (EP), Continual Equilibrium Propagation (C-EP) and Continual Vector Field EP (C-VF). #-h denotes the number of hidden layers. **(a)**: test error rate on MNIST as a function of the initial angle Ψ between the total normalized update of EP and the total gradient of BPTT. **(b)**: Dashed and continuous lines respectively represent the normalized updates $\Delta_\theta(t)$ (i.e. $\Delta_\theta^{\mathrm{EP}}(t)$, $\Delta_\theta^{\mathrm{C-EP}}(t)$, $\Delta_\theta^{\mathrm{C-VF}}(t)$) and the gradients $-\nabla_\theta^{\mathrm{BPTT}}(t)$. Each randomly selected synapse corresponds to one color. While dashed and continuous lines coincide for standard EP, they split apart upon untying the weights and using continual updates.

i.e. $\Psi(\theta_{\mathrm{f}}, \theta_{\mathrm{b}}) = \{0°, 22.5°, 45°, 67.5°, 90°\}$. From standard EP with one hidden layer to C-VF with two hidden layers, the test error increases monotonically with $\Psi\left(\Delta(\mathrm{tot}), -\nabla^{\mathrm{BPTT}}(\mathrm{tot})\right)$ but does not exceed 5.05% on average. This result confirms the importance of proper weight initialization when weights are untied, also discussed in other context [83]. When the initial weight angle is of 0°, the impact of untying the weights on classification accuracy remains constrained, as shown in Table 4.1. Upon untying the forward and backward weights, the test error increases by $\sim 0.2\%$ with one hidden layer and by $\sim 0.5\%$ with two hidden layers compared to standard C-EP.

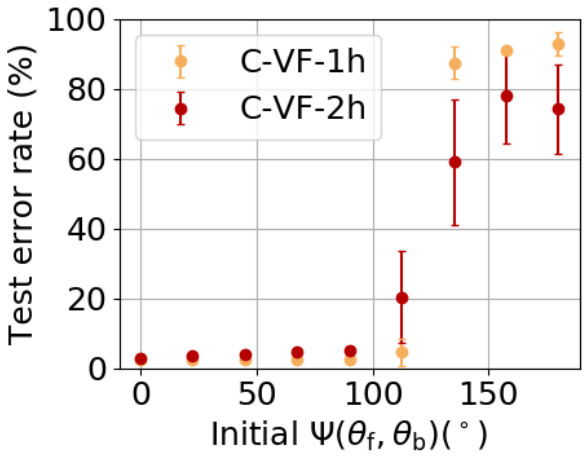

Figure 4.2: Test error rate on MNIST achieved by C-VF as a function of the initial angle $\Psi(\theta_{\mathrm{f}}, \theta_{\mathrm{b}})$ between feedforward weights θ_{f} and feedback weights θ_{b}.

Discussion

Equilibrium Propagation is an algorithm that leverages the dynamical nature of neurons to compute weight gradients through the physics of the neural network. C-EP embraces simultaneous synapse and neuron dynamics, getting rid of the need for artificial memory units for storing the neuron values between different phases. The C-EP framework preserves the equivalence with Backpropagation Through Time: in the limit of sufficiently slow synaptic dynamics (i.e. small learning rates), the system satisfies Gradient Descending Dynamics (Theorem 10).

Our experimental results confirm this theorem. When training our prototypical model with symmetric weights with C-EP while ensuring convergence in 100 epochs, a modest reduction in MNIST accuracy is seen with regards to standard EP. This accuracy reduction can be eliminated by using smaller learning rates and rescaling up the total weight update at the end of the second phase (section 4.2). On top of extending the theory of part IV, Theorem 10 also appears to provide a statistically robust approach for C-EP based learning: our experimental results show, as in part IV, that for a given network with specified neuron and synapse dynamics, the more the updates of Equilibrium Propagation follow the gradients provided by Backpropagation Through Time before training (in terms of angle between weight vectors, in this work), the better this network can learn. Specifically, Fig. 4.1 (a) shows that hyperparameters should be tuned so that before training, C-EP updates stay within 90° of the gradients provided by BPTT. In practice, it amounts to tune the degree of symmetry of the dynamics, for instance the angle between forward and backward weights - see Fig. 4.2.

Our C-EP and C-VF algorithms exhibit features reminiscent of biology. C-VF extends C-EP training to RNNs with asymmetric weights between neurons, as is the case in biology. Its learning rule, local in space and time, is furthermore closely acquainted to Spike Timing Dependent Plasticity (STDP), which we presented in subsection 3.3.2 of part I. Strikingly, the same rule that we use for C-VF learning can approximate STDP correlations in a rate-based formulation, as we also showed in section 2.1 of part III. From this viewpoint, our work brings EP a step closer to biology. This being said, C-EP and C-VF do not aim at being end-to-end models of biological learning. EP and its variants are meant to optimize any given loss function, and this loss function could be for supervised learning (as in our

experiments) or for any differentiable loss function in general. The core motivation of this work is to focus on and propose a fully local implementation of EP, in particular to foster its hardware implementation. When computed on a standard computer, due to the use of small learning rates to mimic analog dynamics within a finite number of epochs, training our models with C-EP and C-VF entail long simulation times. With a Titan RTX GPU, training a fully connected architecture on MNIST takes 2 hours 39 mins with 1 hidden layer and 10 hours 49 mins with 2 hidden layers. On the other hand, C-EP and C-VF might be particularly efficient in terms of speed and energy consumption when operated on neuromorphic hardware that employs analog device physics. To this purpose, our work can provide an engineering guidance to map Equilibrium Propagation onto a neuromorphic system. This is one step towards bridging Equilibrium Propagation with neuromorphic computing and thereby energy efficient hardware implementations of gradient-based learning algorithms.

Part VI

Conclusion and perspectives

Summary of the results

Overall, this thesis emphasizes two components of on-chip learning:

- The computation of the loss gradient: $\frac{\partial \mathcal{L}}{\partial \theta}$.
- Given the loss gradient, the resulting weight update: $\theta \leftarrow \theta - \alpha \frac{\partial \mathcal{L}}{\partial \theta}$.

Part II investigates the second component in several variants of Restricted Boltzmann Machines. With typical values of device imperfections for non-linearity, cycle-to-cycle and device-to-device variabilities, we have shown that the Discriminative RBM is the best candidate architecture in terms of the resulting training performance on MNIST. Also, our simulations highlight how device imperfections influence the optimal pulse width: non-linearity favors small pulse widths and cycle-to-cycle variability large pulse widths conversely. Importantly, a stack of memristive RBMs, when being greedily learnt does not benefit from depth: on the contrary the effect of the device imperfections cumulate when passing features from a RBM to the next one in the stack. This limitation comes from not propagating error signals across layers using backpropagation for instance, to preserve the locality of the learning rule employed. We have also shown that averaging Contrastive Divergence across samples and stochastic realizations of the binary states of the neurons considerably improves the resilience of Discriminative RBMs versus device imperfections. Since this method selects smaller pulse widths, it mitigates both non-linearity and variability effects. We also propose the use of Resilient Propagation (RProp) to facilitate the tuning of the pulse width. While not affecting the resilience of the Discriminative RBM with respect to imperfections and obeying to very simple logics, RProp allows to enlarge the range of usable pulse widths by up to two order of magnitudes. Overall, our study proposes strategies to make Restricted Boltzmann Machines amenable to on-chip training with highly imperfect memristive devices, thereby addressing some of the major challenges of embedded environments.

In part IV, we focus on the first component of learning: the computation of the loss gradient with Equilibrium Propagation. We have proposed a discrete-time formulation of Equilibrium Propagation and in this framework, the original real-time formulation of Equilibrium Propagation can be seen as a particular choice of primitive function Φ. We have shown that our

discrete-time version of Equilibrium Propagation is equivalent to Backpropagation Through Time provided that the Jacobian of the dynamics is symmetric (which is equivalent to the requirement of a primitive function), and equilibrium is reached at the end of the first phase. More precisely, the synapse updates computed in a *forward-time* fashion during the second phase of Equilibrium Propagation are step-by-step equal to the gradients computed by Backpropagation Through Time in a *backward-time* fashion. We call this property the *Gradient Descending Update* (GDU) property and check it on two classes of models: energy-based models and prototypical models. After defining theoretically the fully connected architectures in both settings, we show that the GDU property is generally very well satisfied numerically. More quantitatively, using the Relative Mean Squared (RelMSE) metric, we show that the deeper the network, the larger the RelMSE, thereby suggesting the difficulty to train deep architectures with Equilibrium Propagation. Finally, we propose a Convolutional model in the prototypical setting trainable by Equilibrium Propagation. We show that this architecture also satisfies well the GDU property and achieves the best training performance ever reported on MNIST in the literature of Equilibrium Propagation ($\sim 1\%$ test error). Finally, we show that the use of our prototypical setting speeds up training by a factor 5 to 8 compared to the energy-based setting. This work facilitates the design of models trainable by Equilibrium Propagation both by the practical use of Theorem 4 and the training speed-up offered by the prototypical setting, which is of interest to map Equilibrium Propagation onto neuromorphic systems. These results bring Equilibrium Propagation closer to conventional machine learning and should help the algorithm to scale to more complex problems.

Finally in part V, we extend the study of part IV to a more biologically realistic - and hardware-friendly - setting where the learning rule prescribed by Equilibrium Propagation becomes local in time, a definite asset for future hardware implementations of Equilibrium Propagation. In this new version of Equilibrium Propagation that we call *Continual Equilibrium Propagation* (C-EP), the synapses evolve dynamically along with the neurons during the second phase of Equilibrium Propagation. In this framework, we show that the theorem of the previous part can be extended: in the limit of slow synaptic dynamics, we say that *Gradient Descending Dynamics* (GDD) are satisfied (Theorem 10). We numerically show the GDD property on various models and demonstrate C-EP training on a prototypical model with symmetric weights, therefore benefiting from the subsequent acceleration for training simulations mentioned before. We observe a slight training accuracy reduction compared to standard Equilibrium Propagation which we can directly account for: the learning rate should be small enough so that GDD is sufficiently well satisfied, but not too small so that convergence occurs within a reasonable number of epochs. We extend C-EP training to neural networks whose synaptic connections are asymmetric and we call this version of the algorithm *Continual Vector Field* Equilibrium Propagation (C-VF). We show that C-VF successfully trains neural networks with asymmetric connections on MNIST. Furthermore we show that, given a model with specified dynamics and connectivity, the more Theorem 10 is satisfied *before training* (in terms of the angle between the total weight update over the second phase of C-EP and the negative gradient provided by BPTT), the better the resulting training

performance. This work brings Equilibrium Propagation one step closer to hardware and biology: C-VF can be seen as a rate-based equivalent of Spike Timing Dependent Plasticity (STDP), which is amenable to energy efficient analog hardware. Finally here again, Theorem 10 provides a guidance to map Equilibrium Propagation onto neuromorphic systems.

Other research projects & collaborations

This section describes research projects on Equilibrium Propagation that I have been involved in at the end of my PhD project.

mEqProp: Equilibrium Propagation with memristors in spiking neural networks

In subsection 3.3.2 of the Introduction, we have shown that the use of memristive devices along with spiking voltage pulses with appropriate shapes could emulate Spike Timing Dependent Plasticity - Fig. 3.2 and Fig. 3.3. Given a memristive device emulating a synapse, whenever the pre or post synaptic neuron spikes, a voltage pulse is created and depending on the relative timings of the pre and post synaptic resulting voltages, the device undergoes a subsequent voltage difference that may or may not cross the programming threshold of the device. On the other hand, we have shown in subsection 2.1 the connection existing between Equilibrium Propagation and STDP. Finally, part V provides theoretical guarantees on the rate-based equivalent of an STDP-like implementation of Equilibrium Propagation.

Somehow, combining these three highlights, there comes here some intuition that the learning rule of Equilibrium Propagation could be emulated in an event-based fashion with memristive devices. One possible approach in this purpose is to engineer the shape of the spikes that program the memristive device so that the resulting conductance updates undergone by the device correlate well enough at any time with the weight updates prescribed by Equilibrium Propagation, namely $\Delta_\theta^{\mathrm{C-EP}}(t) \sim \dot{s}_{\mathrm{post}} \rho(s_{\mathrm{pre}})$ to reuse the notations of the last part. Note that this event-based setting where synapses evolve along with neurons throughout the second phase falls into our Continual Equilibrium Propagation setting.

I have worked with **Erwann Martin** (PhD student under the joint supervision of Julie Grollier and Teodora Petrisor at Thalès lab) on this event-based version of Equilibrium

Propagtion to be known as *mEqProp*. Erwann will present the results of this work at the NAISys conference as an oral contribution.

Equilibrium Propagation with physical artificial neurons

In the project previously described, traditional leaky-integrate-and-fire (LIF) neurons are assumed for spiking simulations with a strong focus on how to emulate appropriate synapse dynamics. How about the neurons? In other words, how would be go about implementing Equilibrium Propagation on a physical substrate where the neurons are governed by the equations of their very own physics and not simply LIF equations? At the end of my thesis, I have worked with **Jérémie Laydevant** (a PhD student of Julie Grollier) in this regard and help him make Equilibrium Propagation comply the best with our hardware constraints. To only mention a few of these constraints: how do we encode inputs and outputs? How should they be scaled appropriately? What are the conditions for the system to achieve a stable steady state (a highly non trivial question for non-linear systems for instance)? How do we physically nudge the system during the second phase ? Which physical quantity stores the value of the synapse and what is the subsequent learning rule prescribed by Equilibrium Propagation? What if the neurons or the synapses are constrained to binary values, as it is the case for a lot of candidate technologies? Jérémie's work and results achieved so far are extremely exciting and will hopefully contribute to leading to the first experimental demonstration of Equilibrium Propagation-based training with our technologies.

Scaling Equilibrium Propagation to deeper architectures

Although Equilibrium Propagation has been brought closer to standard machine learning through the "prototypical" setting presented in part IV, it has yet to be scaled up to more complex visual tasks like CIFAR-10 or ImageNet. When working on chapter IV, I could not achieve better than 65-70% test accuracy on CIFAR-10. I believe there are three main reasons for this:

- The first reason why Equilibrium Propagation may scale poorly to deeper architectures is the credit assignment itself used: the requirement of equilibrium. And the deeper the architecture, the longer it takes to reach equilibrium. When running training simulations, a trade-off has to be find between having a first phase that is long enough to ensure equilibrium, but not too long to make simulations achievable within a reasonable time. So it might be that equilibrium is not perfectly achieved when training deep networks.
- Another reason is rather subtle. One should bear in mind that the ability of the system to reach a steady state is an *hypothesis* of the theory of Equilibrium Propagation.

In practice, we use dynamics deriving from an energy function or equivalently of a primitive function in the prototypical setting. But what actually helps the system to reach convergence at the end of the first and second phase is to *clip* the activations of the neurons between 0 and 1. Consequently, a large proportion of neurons saturate at the end of the first phase at equilibrium. Therefore, when nudging the system during the second phase, error signals do not pass through saturated neurons.

- Finally, the learning given by Equilibrium Propagation in Eq. (1.5) in part III is biased in the way the underlying derivative with respect to β is estimated. More precisely, in [97] EP learning rule is more generally expressed in terms of $\frac{d}{d\beta}\left(\frac{\partial E}{\partial \theta}\right)\Big|_{\beta=0}$. Eq. (1.5) is the discrete forward-time estimate of this derivative. Cancelling out this bias, for instance by flipping the sign of β across mini-batches, could also dramatically improve learning in deep architectures.

At the end of my PhD project, I have been working with **Axel Laborieux** to pursue this work and attempt to achieve $> 90\%$ test accuracy on CIFAR-10 using techniques to mitigate what we believe hinder learning in deep networks. We believe that carrying out this work, leading to positive or negative conclusions, could in any case benefit the community, as a continuation of Bartunov et al work [85] on the scalability of biologically inspired learning algorithms and architectures - which, in its study, mainly investigated Feedback Alignment and Target Propagation.

Equilibrium Propagation on sequential data

One natural question that arises is: could it extend to temporal data? At first sight, it seems that the answer is no, again because of the credit assignment scheme itself. Upon presenting a *static* input, the system reaches a steady state so that its subsequent motion triggered by the nudging strength during the second phase encode error signals: $\dot{s} \sim \frac{\partial \mathcal{L}}{\partial s}$. In other words, when applying the nudge, the system should *only* move because of the error signal. Thereby, if the system receives a sequential input, the system may be influenced by both the nudging strength applied on the output layer *and* changing inputs on the visible layer, so that the motion may no longer encode error signals. This being said, we have some intuitions that there are clever ways to build neural network models that could both process temporal data and be trainable under Equilibrium Propagation: error signals could possibly propagate across different inputs.

Equilibrium Propagation without the equilibrium requirement

From the previous points, it appears clearly that the requirement of equilibrium hinders the scalability of Equilibrium Propagation. Would there be any way we could go round this

requirement at all, therefore being able to train deeper architectures, by rethinking the whole credit assignment of "Equilibrium" Propagation? Yoshua Bengio came to Benjamin and I suggesting that if each neuron could access *simultaneously* its free current state s_t (when it evolve freely, without nudging) and nudged state s_t^β (when it evolves under the influence of nudging), the error signal $s_t^\beta - s_t$ thereby created neuron-wise could be leveraged for learning. This idea raises two questions that can be treated independently:

- How could, *each neuron*, know about its free state s_t while being in its nudged state s_t^β?
- Once the error signal $s_t^\beta - s_t$ is extracted, how could it be used to achieve learning?

At the end of my thesis, I have been collaborating with **Yoshua Bengio**, **Blake Richards** and **Damjan Kalajdzievski** on this project. I went to the Mila after NeurIPS in December for a week to help kicking off this project, and hopefully pursue it after the thesis defense.

Some thoughts about longer-term directions of research

The wide spectrum of neuromorphic approaches. Neuromorphic computing is a complex area in itself, as it intertwines closely computer sciences, mathematics, statistics, optimization, neurosciences, general electrical engineering, condensed matter physics and circuit design. There are arguably as many conceptions of the field as there are researchers, as to the expertise, approaches, research endeavors and time scales involved in such research. Should it be? Clearly, the discrepancies of the existing approaches makes it especially hard not only to pick up on the missing skills and keep up with the forefront of miscellaneous research literatures, and even more to form a strong opinion about promising directions of research. From my research experience, I also think that although such diversity of point of views in the field can benefit the community, the numerous research efforts in neuromorphic computing could still benefit from a better coordination across different fields.

Better communication amongst neuromorphic researchers? Today, diverse communities are still working far apart, each coping separately with the technical challenges of their own research on neuromorphic computing. A lot of physicists or electronic researchers within neuromorphic computing work hard to emulate basic functions required for inference with their candidate hardware substrate, such as non-linearity, matrix multiplication, or to connect at all a significant number neurons. Other hardware researchers focus more on the physics of the synapse update, to achieve the most accurate conductance update given a gradient value, as already mentionned in this thesis. Computational neuroscientists are limited in the problems they can solve by the simulation time required to solve learning problems with realistic spiking neurons, motivating projects such as SpiNNaker [27]. Although overcoming these technological challenges is essential, better communication between communities might have the potential of saving a lot of time in the research. Tremendous technological advances have been realized in neuromorphic computing; however there is some risk that these research endeavours, which sometimes involve years of research, build upon learning paradigms that could have benefited from more neuroscience inspiration or mathematical guarantees from the very beginning, far ahead in the project design. For instance, a lot of neuromorphic researchers have designed systems building upon the Bi-Poo Spike Timing Dependent Plas-

ticity arguing about biological plausibility (STDP). Although this research led to extremely exciting results, STDP is a learning heuristic that was not thought at first to optimize an objective function. Also, STDP was fitted on *in vitro* measurements. It was shown that with realistic *in vivo* calcium concentration, Bi-Poo STDP could not be observed [169]. The same team subsequently proposed to infer learning rules from distribution of firing rates in cortical neurons. Such learning rules - based on burst synaptic mechanisms for instance - could also inspire neuromorphic engineering. Finally, although Hebbian learning rules such as STDP are extremely attractive for hardware designers, they should be endowed with strong theoretical guarantees optimization-wise to be able to scale to complex problems, as this thesis suggests. In this regard, better communication between mathematicians and neuromorphic engineers could strongly benefit research towards on-chip learning.

Mathematics versus biological inspiration? When developping efficient neuromorphic systems, should mathematics prevail over biological inspiration or the other way around? Is there such a thing as "mathematics versus biology" when doing neuromorphic research? For the neuromorphic approach to succeed in the long run and scale to complex systems, we very likely need both: close biological inspiration *and* mathematics. The first most obvious example supporting this necessity is Convolutional neural networks. The topology of the convolution operation applied in neural networks is directly inspired by the primary visual cortex. Still, this biological inspiration is not sufficient by itself: the parameters of the convolution are not given by neuroscience measurements but gradually adjusted by back-propagation and gradient descent, whose goal is to minimize a mathematically well-defined objective function. Quite often and intriguingly, the optimization setting of deep learning often enabled to recover some neurophysiological features of the brain like grid cells, Gabor filters, shape tuning or temporal receptive fields [60].

Richards et al more generally support this vision in their Nature paper where they advocate the use of a deep learning framework to fuel progress in neuroscience [60]: this way of thinking could potentially be transferred to neuromorphic computing. More precisely, Richards et al argue that such a framework should rely on three fundamental components: the neural architectures, the objective function and the learning rule. Evaluating deep learning models with an emphasis on these three components on benchmark "Brain tasks" that could also be experimentally carried out on living beings, this framework would enable to create complex models with testable hypothesis. They insist in particular on the necessity of *inductive biases*, as opposed to the properties that emerge during learning. Inductive biases are the priors of the model that are chosen appropriately to help this model learn specific tasks. For example, translation invariance is an inductive bias of convolutional architectures, which is especially suited on visual tasks where the system should detect redundant visual features. This is how both mathematics *and* biology come into play: biology provides inspiration for the inductive biases of the models, the emergent properties of the models come from the mathematical optimization framework itself. The authors beautifully illustrate this trade-off as follows: "[...]*many of the successes of deep learning have grown out of a balance between*

useful ***inductive biases*** *and* ***emergent computation****, echoing the blend of nature and nurture which underpins the adult brain*". Similarly, the design of neuromorphic systems should maybe result from the same combination. More than only bringing memory the closest to computation, the topology of the circuit that achieves inference and gradient computation should directly take inspiration from physiological features of the brain, while still obeying strong mathematical principles.

Evolution? How about biological biases that are endowed by evolution? Could neuromorphic systems somehow inherit "evolved" features? Many experiments suggest that the optimization framework is not sufficient to pick up evolution. It is worth mentioning here a conflicting view that was presented at NeurIPS in a wonderful talk by Blaise Aguera y Arcas from Google *. In his view, life at all scales may not be governed by optimization-based principles. He first underpins this opinion on a bacteria population which he considers as the most simple biological system. What he shows is that when such a population strives to find food and survive with pure evolutionary principles, it elaborates a strategy in terms of displacement given a chemical stimulus - a phenomenon known as bacteria chemotaxis. Whether the strategy eventually retained by the bacteria after a sufficiently long time is *optimal* in some mathematical sense is not simple at all and pertains to inverse reinforcement learning. Rather than pure optimization, Aguera y Arcas rather argues that evolution decides on its own the notion of "good" or "bad" throughout time, so that "what persists exists" in his own words. Similarly, he puts forward that a GAN optimization problem is another case in point where each of the actor - the discriminator and the generator - does a gradient descent of its own well defined loss function, but the combined system does not [170]. At all scales, life operates with modular agents so that intelligence would emerge collectively from the interaction of these agents. Therefore, evolution from this prospective appears as a definite key component of intelligence and results from a combination of complex phenomena that the optimization framework is not sufficient on its own to account for. This being said, there are specific situations where evolution can be emulated in an optimization setting.

Metalearning as a way to emulate evolution Similarly according to Aguera y Arcas, the learning rules themselves should be "evolved". Traditional learning rules, such as backpropagation, act on parameters that model the *connectome*: looping several times over a dataset until convergence of these parameters as we are used to can be seen as a learning episode. One way to model evolution is to parametrise the learning rule itself: these parameters model the *genome*. The way we proceed generally to train these parameters is *metalearning*: each learning episode (as defined before) is part of a outer loop that optimizes the parameters of the genome, or *metaparameters*. A metalearning approach has been used for instance to achieve few shot learning [171], where the learning agent can learn out of a very few samples of each class, pretty much like human beings can. A learning episode consists in learning from a dataset with a few samples, where the learning rule does not read

*The full talk can be found: https://slideslive.com/38922302/social-intelligence.

like gradient descent but LSTM-like dynamics with metaparameters: $\theta_{t+1} \leftarrow \text{LSTM}(\theta_t; \gamma)$. At the end of each learning episode, a *meta-loss* is evaluated on the current test set, and the gradients with respect to γ backpropagated through the *whole* training episode. Alternatively, the γ parameters can also be learnt in a evolutionary way. Eventually, at the end of the metalearning procedure, the γ parameters learnt in this way, or equivalently the learning rule, allow to perform few shot learning on unseen datasets: it has learn to learn with a few examples! Very recently, a metalearning approach - presented at the NeurIPS workshop and at Cosyne - was used to learn local learning rules able to perform few shot learning [172].

Similarly, these metalearning techniques could potentially benefit neuromorphic computing. For example, we could think of these metaparameters as adjustable physical quantities that we would generally tune by hand - e.g. a voltage or current threshold, the shape of a programming pulse, tunable parameters of a technology. Metalearning approaches could for instance be used on software to find good values for these tunable parameters. What if, given a physical substrate with topological constraints, a learning rule could be made local by adjusting properly the properties of the circuits? Also, conductance udpdate is energy costly and some devices are limited by their endurance. What if we could metalearn proper learning rules on software, which once transferred onto hardware would enable to learn with a very few examples, and thereby reduce the energy consumption of neuromorphic systems with a few conductance updates?

Take-away message. This piece of work, which intertwines closely neuroscience, mathematical and physical aspects, is one leap towards achieving such transdisciplinarity in neuromorphic computing. These results and the previous outlook have convinced me of the formidable prospects of this kind of approach. Neuroscientists seek for mathematical models of how the brain might work and neuromorphic researchers would tremendously benefit to be part of the same exciting quest. Intriguingly, neuroscientists may also well gain from the neuromorphic insight, as the emergent physics of the neuromorphic systems might reveal features of the brain that one may have not suspected otherwise. Therefore, bringing these two communities closer, to the extent of collaboration towards building learning theories, may well be a very promising path for neuromorphic computing, and perhaps for neuroscience as well. In the long run, the joint endeavour could potentially lead to building a formal umbrella framework for neuromorphic computing, and hopefully help neuromorphic systems scale to complex tasks.

List of publications

Publications linked to my Ph.D work

Peer-reviewed journal articles

- **Maxence Ernoult**, Julie Grollier, Damien Querlioz. "Using Memristors for Robust Local Learning of Hardware Restricted Boltzmann Machines", Scientific Reports 9(1), 1–15 (2019).
- Miguel Romera, Philippe Talatchian, Sumito Tsunegi, Flavio Abreu Araujo, Vincent Cros, Paolo Bortolotti, Juan Trastoy, Kay Yakushiji, Akio Fukushima, Hitoshi Kubota, Shinji Yuasa, **Maxence Ernoult**, Damir Vodenicarevic, Tifenn Hirtzlin, Nicolas Locatelli, Damien Querlioz, Julie Grollier. “Vowel recognition with four coupled spin-torque nano-oscillators”, Nature 563(7730), 230 (2018).

Pre-prints

- Axel Laborieux, **Maxence Ernoult**, Benjamin Scellier, Yoshua Bengio, Damien Querlioz, Julie Grollier. "Scaling Equilibrium Propagation to Deep ConvNets by Drastically Reducing its Gradient Estimator Bias", arXiv preprint arXiv:2006.03824 (2020).
- **Maxence Ernoult**, Julie Grollier, Damien Querlioz, Yoshua Bengio, Benjamin Scellier, "Equilibrium Propagation with Continual Weight Updates", arXiv preprint arXiv:2005.04168 (2020).
- Axel Laborieux, **Maxence Ernoult**, Tifenn Hirtzlin, Damien Querlioz. "Synaptic Metaplasticity in Binarized Neural Networks", arXiv preprint arXiv:2003.03533 (2020).
- Miguel Romera, Philippe Talatchian, Sumito Tsunegi, Kay Yakushiji, Akio Fukushima, Hitoshi Kubota, Shinji Yuasa, Vincent Cros, Paolo Bortolotti, **Maxence Ernoult**, Damien Querlioz, Julie Grollier. "Binding events through the mutual synchronization of spintronic nano-neurons". arXiv preprint arXiv:2001.08044.(2020)

Peer-reviewed conference articles

- **Maxence Ernoult**, Julie Grollier, Damien Querlioz. "Robust Local Learning with Memristor-Based Restricted Boltzmann Machines". Cognitive Computing, **poster** (2018).
- Tifenn Hirtzlin, Marc Bocquet, **Maxence Ernoult**, Jacques-Olivier Klein, /'Etienne Nowak, Elisa Vianello, Jean-Michel Portal, Damien Querlioz. "Hybrid Analog-Digital Learning with Differential RRAM Synapses". IEEE International Electron Devices Meeting (2019).
- **Maxence Ernoult**, Julie Grollier, Damien Querlioz, Yoshua Bengio and Benjamin Scellier. "Updates of Equilibrium Prop Match Gradients of Backprop Through Time in an RNN with Static Input". In Advances in Neural Information Processing Systems, pages 7079–7089, **oral** (2019).
- **Maxence Ernoult**, Julie Grollier, Damien Querlioz, Yoshua Bengio and Benjamin Scellier. "Continual Weight Updates and Convolutional Architectures for Equilibrium Propagation". Cosyne, **poster** (2020).
- Erwann Martin, **Maxence Ernoult**, Shuai Li, Damien Querlioz, Teodora Petrisor, Julie Grollier. "Spiking Equilibrium Propagation for Intrinsic Learning Hardware". NAISys, **oral** (2020).

Publications prior to my Ph.D

- Christian Hoecker, Jean de la Verpilliere, **Maxence Ernoult**, Brian Graves and Adam Boies. "Theoretical model of CNT aerogel formation". European Aerosol Conference, **poster**(2016).
- Gautier Lefebvre, Alexane Gondel, Marc Dubois, Michael Atlan, Florian Feppon, Aim/'e Labb/'e, Camille Gillot, Alix Garelli, **Maxence Ernoult**, Svitlana Mayboroda, Marcel Filoche, and Patrick Sebbah, "One Single Static Measurement Predicts Wave Localization in Complex Structures", Phys. Rev. Lett. 117, 074301 (2016).

References

References

[1] Alan M Turing. "Computing machinery and intelligence". In *Parsing the Turing Test*, pages 23–65. Springer, (2009).

[2] Warren S McCulloch and Walter Pitts, "A logical calculus of the ideas immanent in nervous activity", *The bulletin of mathematical biophysics* 5(4), 115–133 (1943).

[3] Donald Olding Hebb, *The organization of behavior: A neuropsychological theory*, Psychology Press (2005).

[4] Carla J Shatz, "The developing brain", *Scientific American* 267(3), 60–67 (1992).

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

[6] Bernard Widrow, "Thinking about thinking: the discovery of the LMS algorithm", *IEEE Signal Processing Magazine* 22(1), 100–106 (2005).

[7] Marvin Minsky and Seymour A Papert, *Perceptrons: An introduction to computational geometry*, MIT press (2017).

[8] Paul John Werbos, *The roots of backpropagation: from ordered derivatives to neural networks and political forecasting*, volume 1, John Wiley & Sons (1994).

[9] David B Parker, "Learning Logic Technical Report TR-47", *Center of Computational Research in Economics and Management Science, Massachusetts Institute of Technology, Cambridge, MA* (1985).

[10] Yann LeCun, "A learning scheme for asymmetric threshold networks", *Proceedings of COGNITIVA* 85(537), 599–604 (1985).

[11] David E Rumelhart, Geoffrey E Hinton and Ronald J Williams. "Learning internal representations by error propagation". Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, (1985).

[12] Kunihiko Fukushima and Sei Miyake, "Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position", *Pattern recognition* 15(6), 455–469 (1982).

[13] Yann LeCun, Léon Bottou, Yoshua Bengio and Patrick Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE* 86(11), 2278–2324 (1998).

[14] Sepp Hochreiter and Jürgen Schmidhuber, "Long short-term memory", *Neural computation* 9(8), 1735–1780 (1997).

[15] Alex Krizhevsky, Ilya Sutskever and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In *Advances in neural information processing systems*, pages 1097–1105, (2012).

[16] Diederik P Kingma and Jimmy Ba, "Adam: A method for stochastic optimization", *arXiv preprint arXiv:1412.6980* (2014).

[17] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous and Andre R LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions", *IEEE Journal of Solid-State Circuits* 9(5), 256–268 (1974).

[18] Laszlo B Kish, "End of Moore's law: thermal (noise) death of integration in micro and nano electronics", *Physics Letters A* 305(3-4), 144–149 (2002).

[19] John Von Neumann, "First Draft of a Report on the EDVAC", *IEEE Annals of the History of Computing* 15(4), 27–75 (1993).

[20] "1.1 Computing's energy problem (and what we can do about it)", *Digest of Technical Papers - IEEE International Solid-State Circuits Conference* 57, 10–14 (2014).

[21] John Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", *Communications of the ACM* 21(8), 613–641 (1978).

[22] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard et al. "Tensorflow: A system for large-scale machine learning". In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, (2016).

[23] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga and Adam Lerer, "Automatic differentiation in pytorch", (2017).

[24] Carver Mead, "Neuromorphic electronic systems", *Proceedings of the IEEE* 78(10), 1629–1636 (1990).

[25] Srinjoy Mitra, Stefano Fusi and Giacomo Indiveri, "Real-time classification of complex patterns using spike-based learning in neuromorphic VLSI", *IEEE transactions on biomedical circuits and systems* 3(1), 32–42 (2008).

[26] Ben Varkey Benjamin, Peiran Gao, Emmett McQuinn, Swadesh Choudhary, Anand R Chandrasekaran, Jean-Marie Bussat, Rodrigo Alvarez-Icaza, John V Arthur, Paul A Merolla and Kwabena Boahen, "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations", *Proceedings of the IEEE* 102(5), 699–716 (2014).

[27] Eustace Painkras, Luis A Plana, Jim Garside, Steve Temple, Francesco Galluppi, Cameron Patterson, David R Lester, Andrew D Brown and Steve B Furber, "SpiNNaker: A 1-W 18-core system-on-chip for massively-parallel neural network simulation", *IEEE Journal of Solid-State Circuits* 48(8), 1943–1953 (2013).

[28] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam 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* 34(10), 1537–1557 (2015).

[29] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain et al., "Loihi: A neuromorphic manycore processor with on-chip learning", *IEEE Micro* 38(1), 82–99 (2018).

[30] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers et al. "In-datacenter performance analysis of a tensor processing unit". In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, (2017).

[31] Giacomo Indiveri, Bernabé Linares-Barranco, Robert Legenstein, George Deligeorgis and Themistoklis Prodromakis, "Integration of nanoscale memristor synapses in neuromorphic computing architectures", *Nanotechnology* 24(38), 384010 (2013).

[32] Tifenn Hirtzlin, Marc Bocquet, J-O Klein, Etienne Nowak, Elisa Vianello, J-M Portal and Damien Querlioz. "Outstanding bit error tolerance of resistive ram-based binarized neural networks". In *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 288–292. IEEE, (2019).

[33] Geoffrey W Burr, Robert M Shelby, Abu Sebastian, Sangbum Kim, Seyoung Kim, Severin Sidler, Kumar Virwani, Masatoshi Ishii, Pritish Narayanan, Alessandro Fumarola et al., "Neuromorphic computing using non-volatile memory", *Advances in Physics: X* 2(1), 89–124 (2017).

[34] Stefano Ambrogio, Pritish Narayanan, Hsinyu Tsai, Robert M Shelby, Irem Boybat, Carmelo di Nolfo, Severin Sidler, Massimo Giordano, Martina Bodini, Nathan CP Farinha et al., "Equivalent-accuracy accelerated neural-network training using analogue memory", *Nature* 558(7708), 60–67 (2018).

[35] Leon Chua, "Memristor-the missing circuit element", *IEEE Transactions on circuit theory* 18(5), 507–519 (1971).

[36] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart and R Stanley Williams, "The missing memristor found", *nature* 453(7191), 80–83 (2008).

[37] Massimiliano Di Ventra and Yuriy V Pershin, "On the physical properties of memristive, memcapacitive and meminductive systems", *Nanotechnology* 24(25), 255201 (2013).

[38] Paul Meuffels and Rohit Soni, "Fundamental issues and problems in the realization of memristors", *arXiv preprint arXiv:1207.7319* (2012).

[39] Sascha Vongehr and Xiangkang Meng, "The missing memristor has not been found", *Scientific reports* 5, 11657 (2015).

[40] C Garling, "Wonks question HP's claim to computer-memory missing link", *Wired. com, retrieved* pages 09–23 (2012).

[41] GW Burr, P Narayanan, RM Shelby, Severin Sidler, Irem Boybat, Carmelo di Nolfo and Yusuf Leblebici. "Large-scale neural networks implemented with non-volatile memory as the synaptic weight element: Comparative performance analysis (accuracy, speed, and power)". In *2015 IEEE International Electron Devices Meeting (IEDM)*, pages 4–4. IEEE, (2015).

[42] SR Nandakumar, Manuel Le Gallo, Christophe Piveteau, Vinay Joshi, Giovanni Mariani, Irem Boybat, Geethan Karunaratne, Riduan Khaddam-Aljameh, Urs Egger, Anastasios Petropoulos et al., "Mixed-precision deep learning based on computational memory", *arXiv preprint arXiv:2001.11773* (2020).

[43] T Hirtzlin, Marc Bocquet, M Ernoult, J-O Klein, E Nowak, E Vianello, J-M Portal and D Querlioz. "Hybrid analog-digital learning with differential rram synapses". In *2019 IEEE International Electron Devices Meeting (IEDM)*, (2019).

[44] Hyeongsu Kim, Jong-Ho Bae, Suhwan Lim, Sung-Tae Lee, Young-Tak Seo, Dongseok Kwon, Byung-Gook Park and Jong-Ho Lee, "Efficient precise weight tuning protocol considering variation of the synaptic devices and target accuracy", *Neurocomputing* 378, 189–196 (2020).

[45] Fabien Alibart, Elham Zamanidoost and Dmitri B Strukov, "Pattern classification by memristive crossbar circuits using ex situ and in situ training", *Nature communications* 4(1), 1–7 (2013).

[46] Dirk J Wouters, Rainer Waser and Matthias Wuttig, "Phase-change and redox-based resistive switching memories", *Proceedings of the IEEE* 103(8), 1274–1288 (2015).

[47] Geoffrey W Burr, Robert M Shelby, Severin Sidler, Carmelo Di Nolfo, Junwoo Jang, Irem Boybat, Rohit S Shenoy, Pritish Narayanan, Kumar Virwani, Emanuele U Giacometti et al., "Experimental demonstration and tolerancing of a large-scale neural network (165 000 synapses) using phase-change memory as the synaptic weight element", *IEEE Transactions on Electron Devices* 62(11), 3498–3507 (2015).

[48] Damien Querlioz, Olivier Bichler, Philippe Dollfus and Christian Gamrat, "Immunity to device variations in a spiking neural network with memristive nanodevices", *IEEE Transactions on Nanotechnology* 12(3), 288–295 (2013).

[49] Irem Boybat, Manuel Le Gallo, SR Nandakumar, Timoleon Moraitis, Thomas Parnell, Tomas Tuma, Bipin Rajendran, Yusuf Leblebici, Abu Sebastian and Evangelos Eleftheriou, "Neuromorphic computing with multi-memristive synapses", *Nature communications* 9(1), 1–12 (2018).

[50] Jun-Woo Jang, Sangsu Park, Geoffrey W Burr, Hyunsang Hwang and Yoon-Ha Jeong, "Optimization of conductance change in Pr 1–x Ca x MnO 3-based synaptic devices for neuromorphic systems", *IEEE Electron Device Letters* 36(5), 457–459 (2015).

[51] Pai-Yu Chen, Binbin Lin, I-Ting Wang, Tuo-Hung Hou, Jieping Ye, Sarma Vrudhula, Jae-sun Seo, Yu Cao and Shimeng Yu. "Mitigating effects of non-ideal synaptic device characteristics for on-chip learning". In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 194–199. IEEE, (2015).

[52] Ligang Gao, Pai-Yu Chen and Shimeng Yu, "Programming protocol optimization for analog weight tuning in resistive memories", *IEEE Electron Device Letters* 36(11), 1157–1159 (2015).

[53] Shimeng Yu, Pai-Yu Chen, Yu Cao, Lixue Xia, Yu Wang and Huaqiang Wu. "Scaling-up resistive synaptic arrays for neuro-inspired architecture: Challenges and prospect". In *2015 IEEE International Electron Devices Meeting (IEDM)*, pages 17–3. IEEE, (2015).

[54] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli and Anima Anandkumar, "signSGD: Compressed optimisation for non-convex problems", *arXiv preprint arXiv:1802.04434* (2018).

[55] Lukas Balles and Philipp Hennig, "Dissecting adam: The sign, magnitude and variance of stochastic gradients", *arXiv preprint arXiv:1705.07774* (2017).

[56] W Schiffmann, M Joost and R Werner, "Optimization of the backpropagation algorithm for training multilayer perceptrons", *University of Koblenz: Institute of Physics* (1994).

[57] Christopher H Bennett, Vivek Parmar, Laurie E Calvet, Jacques-Olivier Klein, Manan Suri, Matthew J Marinella and Damien Querlioz, "Contrasting advantages of learning with random weights and backpropagation in non-volatile memory neural networks", *IEEE Access* 7, 73938–73953 (2019).

[58] Alaa Saade, Francesco Caltagirone, Igor Carron, Laurent Daudet, Angélique Drémeau, Sylvain Gigan and Florent Krzakala. "Random projections through multiple optical scattering: Approximating kernels at the speed of light". In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6215–6219. IEEE, (2016).

[59] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen and Hai Li. "Terngrad: Ternary gradients to reduce communication in distributed deep learning". In *Advances in neural information processing systems*, pages 1509–1519, (2017).

[60] Blake A Richards, Timothy P Lillicrap, Philippe Beaudoin, Yoshua Bengio, Rafal Bogacz, Amelia Christensen, Claudia Clopath, Rui Ponte Costa, Archy de Berker, Surya Ganguli et al., "A deep learning framework for neuroscience", *Nature neuroscience* 22(11), 1761–1770 (2019).

[61] Yang Dan and Mu-ming Poo, "Spike timing-dependent plasticity of neural circuits", *Neuron* 44(1), 23–30 (2004).

[62] José Antonio Pérez-Carrasco, Carlos Zamarreno-Ramos, Teresa Serrano-Gotarredona and Bernabé Linares-Barranco. "On neuromorphic spiking architectures for asynchronous stdp memristive systems". In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 1659–1662. IEEE, (2010).

[63] Olivier Bichler, Damien Querlioz, Simon J Thorpe, Jean-Philippe Bourgoin and Christian Gamrat. "Unsupervised features extraction from asynchronous silicon retina through spike-timing-dependent plasticity". In *The 2011 International Joint Conference on Neural Networks*, pages 859–866. IEEE, (2011).

[64] Michael Beyeler, Nikil D Dutt and Jeffrey L Krichmar, "Categorization and decision-making in a neurobiologically plausible spiking network using a STDP-like learning rule", *Neural Networks* 48, 109–124 (2013).

[65] Milad Mozafari, Mohammad Ganjtabesh, Abbas Nowzari-Dalini, Simon J Thorpe and Timothée Masquelier, "Bio-inspired digit recognition using reward-modulated spike-timing-dependent plasticity in deep convolutional networks", *Pattern Recognition* 94, 87–95 (2019).

[66] Jean-Pascal Pfister, Taro Toyoizumi, David Barber and Wulfram Gerstner, "Optimal spike-timing-dependent plasticity for precise action potential firing in supervised learning", *Neural computation* 18(6), 1318–1348 (2006).

[67] Miguel Romera, Philippe Talatchian, Sumito Tsunegi, Flavio Abreu Araujo, Vincent Cros, Paolo Bortolotti, Juan Trastoy, Kay Yakushiji, Akio Fukushima, Hitoshi Kubota et al., "Vowel recognition with four coupled spin-torque nano-oscillators", *Nature* 563(7730), 230 (2018).

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

[69] David H Ackley, Geoffrey E Hinton and Terrence J Sejnowski, "A learning algorithm for Boltzmann machines", *Cognitive science* 9(1), 147–169 (1985).

[70] Paul Smolensky. "Information processing in dynamical systems: Foundations of harmony theory". Technical report, Colorado Univ at Boulder Dept of Computer Science, (1986).

[71] Geoffrey E Hinton, "Training products of experts by minimizing contrastive divergence", *Neural computation* 14(8), 1771–1800 (2002).

[72] Javier R Movellan. "Contrastive hebbian learning in the continuous hopfield model". In *Connectionist models*, pages 10–17. Elsevier, (1991).

[73] Emre Neftci, Srinjoy Das, Bruno Pedroni, Kenneth Kreutz-Delgado and Gert Cauwenberghs, "Event-driven contrastive divergence for spiking neuromorphic systems", *Frontiers in neuroscience* 7, 272 (2014).

[74] S Burc Eryilmaz, Emre Neftci, Siddharth Joshi, SangBum Kim, Matthew BrightSky, Hsiang-Lan Lung, Chung Lam, Gert Cauwenberghs and H-S Philip Wong, "Training a Probabilistic Graphical Model with Resistive Switching Electronic Synapses", *arXiv preprint arXiv:1609.08686* (2016).

[75] M Ishii, S Kim, S Lewis, A Okazaki, J Okazawa, M Ito, M Rasch, W Kim, A Nomura, U Shin et al. "On-chip trainable 1.4 m 6t2r pcm synaptic array with 1.6 k stochastic lif neurons for spiking rbm". In *2019 IEEE International Electron Devices Meeting (IEDM)*, pages 14–2. IEEE, (2019).

[76] Pieter R Roelfsema and Arjen van Ooyen, "Attention-gated reinforcement learning of internal representations for classification", *Neural computation* 17(10), 2176–2214 (2005).

[77] Isabella Pozzi, Sander Bohté and Pieter Roelfsema, "A biologically plausible learning rule for deep learning in the brain", *arXiv preprint arXiv:1811.01768* (2018).

[78] Justin Werfel, Xiaohui Xie and H Sebastian Seung. "Learning curves for stochastic gradient descent in linear feedforward networks". In *Advances in neural information processing systems*, pages 1197–1204, (2004).

[79] Benjamin James Lansdell and Konrad Paul Kording, "Spiking allows neurons to estimate their causal effect", *bioRxiv* page 253351 (2019).

[80] Dong-Hyun Lee, Saizheng Zhang, Asja Fischer and Yoshua Bengio. "Difference target propagation". In *Joint european conference on machine learning and knowledge discovery in databases*, pages 498–515. Springer, (2015).

[81] James CR Whittington and Rafal Bogacz, "An approximation of the error backpropagation algorithm in a predictive coding network with local hebbian synaptic plasticity", *Neural computation* 29(5), 1229–1262 (2017).

[82] Benjamin James Lansdell, Prashanth Prakash and Konrad Paul Kording, "Learning to solve the credit assignment problem", *arXiv preprint arXiv:1906.00889* (2019).

[83] Timothy P Lillicrap, Daniel Cownden, Douglas B Tweed and Colin J Akerman, "Random synaptic feedback weights support error backpropagation for deep learning", *Nature communications* 7(1), 1–10 (2016).

[84] Arild Nøkland. "Direct feedback alignment provides learning in deep neural networks". In *Advances in neural information processing systems*, pages 1037–1045, (2016).

[85] Sergey Bartunov, Adam Santoro, Blake Richards, Luke Marris, Geoffrey E Hinton and Timothy Lillicrap. "Assessing the scalability of biologically-motivated deep learning algorithms and architectures". In *Advances in Neural Information Processing Systems*, pages 9368–9378, (2018).

[86] Will Xiao, Honglin Chen, Qianli Liao and Tomaso Poggio, "Biologically-plausible learning algorithms can scale to large datasets", *arXiv preprint arXiv:1811.03567* (2018).

[87] Mohamed Akrout, Collin Wilson, Peter Humphreys, Timothy Lillicrap and Douglas B Tweed. "Deep learning without weight transport". In *Advances in Neural Information Processing Systems*, pages 974–982, (2019).

[88] Johannes Christian Thiele, Olivier Bichler and Antoine Dupret, "SpikeGrad: An ANN-equivalent Computation Model for Implementing Backpropagation with Spikes", *arXiv preprint arXiv:1906.00851* (2019).

[89] João Sacramento, Rui Ponte Costa, Yoshua Bengio and Walter Senn. "Dendritic cortical microcircuits approximate the backpropagation algorithm". In *Advances in Neural Information Processing Systems*, pages 8721–8732, (2018).

[90] Richard Naud and Henning Sprekeler, "Sparse bursts optimize information transmission in a multiplexed neural code", *Proceedings of the National Academy of Sciences* 115(27), E6329–E6338 (2018).

[91] Paul J Werbos, "Generalization of backpropagation with application to a recurrent gas market model", *Neural networks* 1(4), 339–356 (1988).

[92] Ronald J Williams and David Zipser, "A learning algorithm for continually running fully recurrent neural networks", *Neural computation* 1(2), 270–280 (1989).

[93] L. B. Almeida. "A learning rule for asynchronous perceptrons with feedback in a combinatorial environment". volume 2, pages 609–618, San Diego 1987, (1987). IEEE, New York.

[94] F. J. Pineda, "Generalization of Back-Propagation to Recurrent Neural Networks", 59, 2229–2232 (1987).

[95] Guillaume Bellec, Franz Scherr, Elias Hajek, Darjan Salaj, Robert Legenstein and Wolfgang Maass, "Biologically inspired alternatives to backpropagation through time for learning in recurrent neural nets", *arXiv preprint arXiv:1901.09049* (2019).

[96] Dongsung Huh and Terrence J Sejnowski. "Gradient descent for spiking neural networks". In *Advances in Neural Information Processing Systems*, pages 1433–1443, (2018).

[97] Benjamin Scellier and Yoshua Bengio, "Equilibrium propagation: Bridging the gap between energy-based models and backpropagation", *Frontiers in computational neuroscience* 11, 24 (2017).

[98] Sung Hyun Jo, Ting Chang, Idongesit Ebong, Bhavitavya B. Bhadviya, Pinaki Mazumder and Wei Lu, "Nanoscale memristor device as synapse in neuromorphic systems", *Nano Letters* 10(4), 1297–1301 (2010).

[99] Fabien Alibart, Bernabé Linares-Barranco, Robert Legenstein, George Deligeorgis and Themistoklis Prodromakis, "Integration of nanoscale memristor synapses in neuromorphic computing architectures", *Nanotechnology* 24(38) (2013).

[100] J Joshua Yang, Byung Joon Choi, Min-xian Zhang, Antonio C Torrezan, John Paul Strachan, Stanley Williams and R Stanley Williams, "Memristive Devices for Computing : Mechanisms , Applications and Challenges", *The Electrochemical Society* (2013).

[101] Davide Sacchetto, Pierre-emmanuel Gaillardon, Michael Zervas, Sandro Carrara, Giovanni De Micheli and Yusuf Leblebici, "Applications of Multi-Terminal Memristive Devices : A Review", *IEEE Circuits and Systems Magazine* pages 23–41 (2013).

[102] Fabien Alibart, Elham Zamanidoost and Dmitri B. Strukov, "Pattern classification by memristive crossbar circuits using ex situ and in situ training", *Nature Communications* 4(May), 1–7 (2013).

[103] X Guo, F Merrikh Bayat, M Bavandpour, M Klachko, M R Mahmoodi, M Prezioso, Santa Barbara, Santa Barbara and Stony Brook. "Mixed-Signal Neuromorphic Classifier Based on Embedded NOR Flash Memory Technology". In *Electron Devices Meeting (IEDM), 2017 IEEE International*, pages 151–154, (2017).

[104] G.W. Burr, R.M. Shelby, C. di Nolfo, J.W. Jang, R.S. Shenoy, P. Narayanan, K. Virwani, E.U. Giacometti, B. Kurdi and H. Hwang. "Experimental demonstration and tolerancing of a large-scale neural network (165,000 synapses), using phase-change memory as the synaptic weight element". In *2014 IEEE International Electron Devices Meeting*, (2014).

[105] "Efficient training algorithms for neural networks based on memristive crossbar circuits", *Proceedings of the International Joint Conference on Neural Networks* 2015-Septe (2015).

[106] D Soudry, D Di Castro, A Gal, A Kolodny and S Kvatinsky, "Memristor-based multilayer neural networks with online gradient descent training", *IEEE transactions on neural networks and learning systems* 26(10), 2408–2421 (2015).

[107] "Physical Realization of a Supervised Learning System Built with Organic Memristive Synapses", *Scientific Reports* 6(September), 1–12 (2016).

[108] Suhwan Lim, Jong-Ho Bae, Jai-Ho Eum, Sungtae Lee, Chul-Heung Kim, Byung-Gook Park and Jong-Ho Lee, "Adaptive Learning Rule for Hardware-based Deep Neural Networks Using Electronic Synapse Devices", *arXiv:1707.06381* (2017).

[109] Nandakumar S. R., Manuel Le Gallo, Irem Boybat, Bipin Rajendran, Abu Sebastian and Evangelos Eleftheriou, "Mixed-precision training of deep neural networks using computational memory", *arXiv:1712.01192* (2017).

[110] Anakha V Babu and Bipin Rajendran, "Stochastic Deep Learning in Memristive Networks", *arXiv:1711.03640* (2017).

[111] M Prezioso, B Hoskins, G Adam, K K Likharev and D B Strukov, "Training and operation of an integrated neuromorphic network based on metal-oxide memristors", *Nature* (December), 1–21 (2015).

[112] Koki Kawasaki, Tomohiro Yoshikawa and Takeshi Furuhashi, "A Study on Visualizing Feature Extracted from Deep Restricted Boltzmann Machine using PCA", *International Journal of Computer Information Systems and Industrial Management Applications.* 8, 67–76 (2016).

[113] Na Lu, Tengfei Li, Xiaodong Ren and Hongyu Miao, "A Deep Learning Scheme for Motor Imagery Classification based on Restricted Boltzmann Machines", *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 25(6), 566–576 (2017).

[114] Hugo Larochelle and Yoshua Bengio, "Classification using Discriminative Restricted Boltzmann Machines", *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)* (2008).

[115] Geoffrey E. Hinton, "Training Products of Experts by Minimizing Contrastive Divergence", *Neural Computation* (2002).

[116] Manan Suri, Vivek Parmar, Ashwani Kumar, Damien Querlioz and Fabien Alibart, "Neuromorphic Hybrid RRAM-CMOS RBM Architecture", *Non-Volatile Memory Technology Symposium (NVMTS), 2015 15th* (2015).

[117] Ahmad Muqeem Sheri, Aasim Rafique, Witold Pedrycz and Moongu Jeon, "Contrastive divergence for memristor-based restricted Boltzmann machine", *Engineering Applications of Artificial Intelligence* (2015).

[118] Mahdi Nazm Bojnordi and Engin Ipek. "Memristive Boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning". In *Proceedings - International Symposium on High-Performance Computer Architecture*, (2016).

[119] Vivek Parmar and Manan Suri, "Design Exploration of Hybrid CMOS-OxRAM Deep Generative Architectures", *https://arxiv.org/pdf/1801.02003* (2018).

[120] "A novel memristor-based restricted Boltzmann machine for contrastive divergence", *IEICE Electronics Express* (2018).

[121] Y. Lecun and C. Cortes, "The MNIST database of handwritten digits [Online].", *http://yann.lecun.com/exdb/mnist/* (1998).

[122] W Schiffmann, M Joost and R Werner, "Optimization of the Backpropagation Algorithm for Training Multilayer Perceptrons", *Physics* (\mbox{}), 1–36 (1994).

[123] Christian Igel and Michael Hüsken, "Empirical evaluation of the improved Rprop learning algorithms"(2003).

[124] Alan Mosca and George D. Magoulas, "Adapting Resilient Propagation for Deep Learning", *https://arxiv.org/pdf/1509.04612* (2015).

[125] Maxence Ernoult, Julie Grollier and Damien Querlioz, "Using memristors for robust local learning of hardware restricted Boltzmann machines", *Scientific reports* 9(1), 1–15 (2019).

[126] Asja Fischer and Christian Igel, "Training restricted Boltzmann machines: An introduction", *Pattern Recognition* 47(1), 25–39 (2014).

[127] Damien Querlioz, Philippe Dollfus, Olivier Bichler and Christian Gamrat. "Learning with memristive devices: How should we model their behavior?". In *Proceedings of the 2011 IEEE/ACM International Symposium on Nanoscale Architectures, NANOARCH 2011*, (2011).

[128] S. La Barbera, D. R. B. Ly, G. Navarro, N. Castellani, O. Cueto, G. Bourgeois, B. De Salvo, E. Nowak and Elisa Vianello Querlioz, D., "Narrow Heater Bottom Electrode-Based Phase Change Memory as a Bidirectional Artificial Synapse", *Advanced Electronic Materials* (2018).

[129] Alexander Serb, Johannes Bill, Ali Khiat, Radu Berdan, Robert Legenstein and Themis Prodromakis, "Unsupervised learning in probabilistic neural networks with multi-state metal-oxide memristive synapses", *Nature communications* 7, 12611 (2016).

[130] Ziegler Martin Kohlstedt Hermann Hansen Mirko, Zahari Finn, "Double-barrier memristive devices for unsupervised learning and pattern recognition", *Frontiers in Neuroscience* 11, 1–11 (2017).

[131] Alessandro Fumarola, Pritish Narayanan, Lucas L. Sanches, Severin Sidler, Junwoo Jang, Kibong Moon, Robert M. Shelby, Hyunsang Hwang and Geoffrey W. Burr, "Accelerating machine learning with Non-Volatile Memory: Exploring device and circuit tradeoffs", *2016 IEEE International Conference on Rebooting Computing, ICRC 2016 - Conference Proceedings* (2016).

[132] V Nair Manu and Piotr Dudek. "Gradient-descent-based learning in memristive crossbar arrays". In *International Joint Conference on Neural Networks (IJCNN)*. IEEE, (2015).

[133] Lukas Balles and Philipp Hennig. "Dissecting Adam: The Sign, Magnitude and Variance of Stochastic Gradients". (2017).

[134] Lorenz K Müller, Manu V Nair and Giacomo Indiveri, "Randomized Unregulated Step Descent for Limited Precision Synaptic Elements", *IEEE* (2017).

[135] Geoffrey E Hinton, Simon Osindero and Yee-Whye Teh, "A Fast Learning Algorithm for Deep Belief Nets", *Neural computation* (2006).

[136] Shimeng Yu, "Neuro-Inspired Computing With Emerging Nonvolatile Memory", *Proceedings of the IEEE* 106(2) (2018).

[137] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever and Ruslan Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", *Journal of Machine Learning Research* 15, 1929–1958 (2014).

[138] "Rate-coded DBN: An online strategy for spike-based deep belief networks", *Biologically Inspired Cognitive Architectures* (April), 0–1 (2018).

[139] Emre O. Neftci, Bruno U. Pedroni, Siddharth Joshi, Maruan Al-Shedivat and Gert Cauwenberghs, "Stochastic synapses enable efficient brain-inspired learning machines", *Frontiers in Neuroscience* 10(JUN), 1–29 (2016).

[140] Samuel L.Smith, Pieter-Jan Kindermans, Chris Ying and Quoc V. Le. "Don't Decay the Learning Rate, Increase the Batch Size". In *ICLR 2018*, number 2017, pages 1–11, (2018).

[141] P. Narayanan, L.L. Sanches, A. Fumarola, R.M. Shelby, S. Ambrogio, J. Jang, H. Hwang, Y. Leblebici and G.W. Burr, "Reducing circuit design complexity for neuromorphic machine learning systems based on Non-Volatile Memory arrays", *Proceedings - IEEE International Symposium on Circuits and Systems* pages 4–7 (2017).

[142] Chen Pai-Yu, Lin Binbin, Wang I-Ting, Hou Tuo-Hung, Ye Jieping, Vrudhula Sarma, Seo Jaesun, Cao Yu and Yu Shimeng. "Mitigating Effects of Non-ideal Synaptic Device Characteristics for On-chip Learning". In *Computer-Aided Design (ICCAD)*, (2015).

[143] Matthew Jerry, Pai-yu Chen, Jianchi Zhang, Pankaj Sharma, Kai Ni, Shimeng Yu and Suman Datta, "Ferroelectric FET Analog Synapse for Acceleration of Deep Neural Network Training", *International Electron Devices Meeting, IEDM* 6(c) (2017).

[144] "Programming Protocol Optimization for Analog Weight Tuning in Resistive Memories", *IEEE Electron Device Letters* (2015).

[145] Irem Boybat, Carmelo Nolfo, Stefano Ambrogio, Martina Bodini, Nathan C P Farinha, Robert M Shelby, Pritish Narayanan, Severin Sidler, Hsinyu Tsai, Yusuf Leblebici, Geoffrey W Burr, Harry Road and San Jose. "Improved Deep Neural Network hardware-accelerators based on Non-Volatile-Memory : the Local Gains technique". In *Rebooting Computing (ICRC)*, (2017).

[146] Jason Yosinski, Jeff Clune, Yoshua Bengio and Hod Lipson. "How transferable are features in deep neural networks?". In *Advances in neural information processing systems*, pages 3320–3328, (2014).

[147] Benjamin Scellier, Anirudh Goyal, Jonathan Binas, Thomas Mesnard and Yoshua Bengio, "Generalization of equilibrium propagation to vector field dynamics", *arXiv preprint arXiv:1808.04873* (2018).

[148] Benjamin Scellier and Yoshua Bengio, "Equivalence of equilibrium propagation and recurrent backpropagation", *Neural computation* 31(2), 312–329 (2019).

[149] Ruslan Salakhutdinov and Geoffrey Hinton. "Deep boltzmann machines". In *Artificial intelligence and statistics*, pages 448–455, (2009).

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

[151] Yoshua Bengio and Asja Fischer, "Early inference in energy-based models approximates back-propagation", *arXiv preprint arXiv:1510.02777* (2015).

[152] Max Welling and Yee W Teh. "Bayesian learning via stochastic gradient langevin dynamics". In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 681–688, (2011).

[153] Yoshua Bengio, Thomas Mesnard, Asja Fischer, Saizheng Zhang and Yuhuai Wu, "STDP as presynaptic activity times rate of change of postsynaptic activity", *arXiv preprint arXiv:1509.05936* (2015).

[154] Luis B Almeida. "A learning rule for asynchronous perceptrons with feedback in a combinatorial environment". In *Artificial neural networks: concept learning*, pages 102–111. (1990).

[155] Fernando J Pineda, "Generalization of back-propagation to recurrent neural networks", *Physical review letters* 59(19), 2229 (1987).

[156] Maxence Ernoult, Julie Grollier, Damien Querlioz, Yoshua Bengio and Benjamin Scellier. "Updates of equilibrium prop match gradients of backprop through time in an rnn with static input". In *Advances in Neural Information Processing Systems*, pages 7079–7089, (2019).

[157] Yann LeCun, Yoshua Bengio and Geoffrey Hinton, "Deep learning", *Nature* 521(7553), 436 (2015).

[158] Francis Crick, "The recent excitement about neural networks", *Nature* 337(6203), 129–132 (1989).

[159] Editorial, "Big data needs a hardware revolution", *Nature* 554(7691), 145 February 2018.

[160] Benjamin Scellier and Yoshua Bengio, "Equilibrium propagation: Bridging the gap between energy-based models and backpropagation", *Frontiers in computational neuroscience* 11 (2017).

[161] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner and Gabriele Monfardini, "The graph neural network model", *IEEE Transactions on Neural Networks* 20(1), 61–80 (2009).

[162] Corentin Tallec and Yann Ollivier, "Unbiased online recurrent optimization", *arXiv preprint arXiv:1702.05043* (2017).

[163] P. O'Connor, E. Gavves and M. Welling. "Initialized equilibrium propagation for backprop-free training". In *Initialized Equilibrium Propagation for Backprop-Free Training*", (2018).

[164] Peter O'Connor, Efstratios Gavves and Max Welling. "Training a spiking neural network with equilibrium propagation". In Kamalika Chaudhuri and Masashi Sugiyama, editors, *Training a Spiking Neural Network with Equilibrium Propagation*", volume 89 of *Proceedings of Machine Learning Research*, pages 1516–1523. PMLR, 16–18 Apr 2019.

[165] Jacob Torrejon, Mathieu Riou, Flavio Abreu Araujo, Sumito Tsunegi, Guru Khalsa, Damien Querlioz, Paolo Bortolotti, Vincent Cros, Kay Yakushiji, Akio Fukushima et al., "Neuromorphic computing with nanoscale spintronic oscillators", *Nature* 547(7664), 428 (2017).

[166] J Feldmann, N Youngblood, CD Wright, H Bhaskaran and WHP Pernice, "All-optical spiking neurosynaptic networks with self-learning capabilities", *Nature* 569(7755), 208 (2019).

[167] Emma Strubell, Ananya Ganesh and Andrew McCallum, "Energy and Policy Considerations for Deep Learning in NLP", *arXiv preprint arXiv:1906.02243* (2019).

[168] John A Hertz, *Introduction to the theory of neural computation*, CRC Press (2018).

[169] Sukbin Lim, Jillian L McKee, Luke Woloszyn, Yali Amit, David J Freedman, David L Sheinberg and Nicolas Brunel, "Inferring learning rules from distributions of firing rates in cortical neurons", *Nature neuroscience* 18(12), 1804 (2015).

[170] Vaishnavh Nagarajan and J Zico Kolter. "Gradient descent gan optimization is locally stable". In *Advances in neural information processing systems*, pages 5585–5595, (2017).

[171] Sachin Ravi and Hugo Larochelle, "Optimization as a model for few-shot learning", (2016).

[172] Jack Lindsey, "Learning to Learn with Feedback and Local Plasticity", (2019).

[173] E. Cha, J. Woo, D. Lee, J.and Koo Y. ... Lee, S.and Song and K. Shiraishi, "Nanoscale (10nm) 3D vertical ReRAM and NbO 2 threshold selector with TiN electrode", *Electron Devices Meeting (IEDM), 2013 IEEE International* (2013).

[174] Y. Koo, K. Baek and H. Hwang, "Te-based amorphous binary OTS device with excellent selector characteristics for x-point memory applications", *VLSI Technology, 2016 IEEE Symposium on IEEE* pages 1–2 (2013).

[175] Geoffrey Hinton. "A Practical Guide to Training Restricted Boltzmann Machines A Practical Guide to Training Restricted Boltzmann Machines". In *Neural networks: Tricks of the trade ().* ., volume 9, chapter pp. 599-61, page 1. (2010).

[176] Benjamin Scellier and Yoshua Bengio, "Towards a biologically plausible backprop", *arXiv preprint arXiv:1602.05179v2* 914 (2016).

Part VII

Appendices

Chapter 1

Appendix of part II

1.1 Memristor model used

Integrating Eq. (1.5) between t_0 and $t_0 + \Delta t$ yields the explicit effective conductance update $G(t_0 + \Delta t) - G(t_0)$, which is equal to :

$$\begin{cases} f_p(G(t_0), \Delta t) = \frac{G_{max}-G_{min}}{\beta_p} \log\left(1 + \frac{\beta_p}{G_{max}-G_{min}} C_p \Delta t \exp\left(-\beta_p \frac{G(t_0)-G_{min}}{G_{max}-G_{min}}\right)\right) & \text{(potentiation)} \\ f_m(G(t_0), \Delta t) - \frac{G_{max}-G_{min}}{\beta_d} \log\left(1 + \frac{\beta_d}{G_{max}-G_{min}} C_d \Delta t \exp\left(-\beta_d \frac{G_{max}-G(t_0)}{G_{max}-G_{min}}\right)\right) & \text{(depression)} \end{cases}, \tag{1.1}$$

Eq. (1.1) is the explicit form of Eq. (1.6). In most of our simulations, we took $C_p = C_d = C$ and $\beta_p = \beta_d = \beta$. The single part of the study where this symmetry is broken is when studying device-to-device variability -see below. The constant C, encoding the voltage amplitude applied to the device, is fixed by the condition $\Delta G(t_0, \Delta t_{max}) = G_{max} - G_{min}$, $G(t_0) = G_{min}$ yielding with Eq. (1.1) $C = \frac{G_{max}-G_{min}}{\Delta t_{max}} \frac{\exp\beta - 1}{\beta}$. Injecting this C back into Eq.(1.1) shows that only the ratio $\Delta t/\Delta t_{max}$ is relevant. Note that C depends on β so that whenever β was changed, so was C. In all the simulations, G_{max} was taken to be 1 and $G_{min} = 1/13$. To model cycle-to-cycle variability, we simply added a Gaussian noise to each conductance update dictated by Eq.(1.1), e.g. $\Delta G_{tot}(t_0, \Delta t) = \Delta G(t_0, \Delta t) + \text{noise}$ with noise$\sim \mathcal{N}(0, \sigma^2_{intra})$ with $\sigma_{intra} = \varepsilon_{intra}(G_{max} - G_{min})$. The parameter ε_{intra} is the actual quantity we called 'cycle-to-cycle variability' throughout the paper: its value was swept through $\{0.001, 0.003, 0.006, 0.01, 0.02, 0.03\}$. For a given β, device-to-device variability was modeled by adding a dispersion on the coefficient C with $C \sim \log\mathcal{N}\left(\log\frac{\bar{C}}{\sqrt{1+\varepsilon^2_{inter}}}, \sqrt{\log(1+\varepsilon^2_{inter})}\right)$ so that $\langle C\rangle = \bar{C}$ $\sigma(C) = \varepsilon_{inter}\bar{C}$ with $\bar{C} = \frac{G_{max}-G_{min}}{\Delta t_{max}} \frac{\exp\beta - 1}{\beta}$. The parameter ε_{inter} is the actual quantity we called 'device-to-device variability' throughout the paper: its value was swept through $\{0.01, 0.1, 1, 2, 4\}$. Note that consequently in this very particular case: $C_{+,p} \neq C_{+,p}$ (one given device do not respond symmetrically to potentiation and depression) and $C_{+,p} \neq C_{-,p}$ (devices of the same pair do not respond symmetrically to potentiation).

Finally note that our model defined as Eq. (1.5) can look similar to [130]:

$$\frac{dG(t)}{dt} = \beta(G(t), n, \Delta V(t), \Delta t)\left(1 - \frac{G(t)}{G_{max}}\right), \tag{1.2}$$

with some important differences however. The coefficient β does not have the same meaning in the two models: while it is a constant in ours, it appears in theirs as a function of the applied voltage height ($\Delta V(t)$) and width (Δt) to model switching dynamics which we did not take into account. Morever, the number of pulses applied (n) appears explicitly in their model while it is implicit in ours: we treat equally a pulse of length Δt and n pulses of length $\Delta t/n$. Finally, the dependence of the conductance update with the current conductance is exponential in our model while it is polynomial in theirs. Expanding Eq. (1.5) for a small $\beta$and assuming $G_{max} >> G_{min}$ brings our model the closest to Eq. (1.2), up to the linear contribution in C:

$$\frac{dG(t)}{dt} \sim_{\beta\to 0} \begin{cases} C + \beta\frac{CG_{min}}{G_{max}}(1 - \frac{G(t)}{G_{min}}) \text{ (potentiation)} \\ -C + \beta C(1 - \frac{G(t)}{G_{max}}) \text{ (depression)} \end{cases}. \tag{1.3}$$

1.2 Simulations

All the simulations presented in this work have been carried out in the most simple way in the sense that it is abstracted from the realistic constraints inherent to the crossbar circuitry. We assumed that the memristive devices are associated with an access device (transistor [34] or resistive switching selector device [173, 174]), and therefore neglected sneak paths current effects. Sneak paths currents, if present, would significantly decrease the accuracy in programming the synaptic weights. Our goal is to focus on the effects of the weight update physics and the learning rules it enables on the different neural network architectures introduced above, so as to motivate further realistic investigations.

When training any architecture throughout the paper, we used $40,000$ samples for training and $10,000$ for test from the MNIST data base. The different neural network topologies (bias included) were set as follows: 785-301-10 for the RBM+softmax stack, 794(784+10)+300 for the Discriminative RBM and 785+501+511(501+10)+2001 for the Deep Belief Net. In practice, biases were concatenated to W as an extra column and row. Labels are one-hot encoded, e.g. the label "2" is encoded as
$(0, 1, 0, 0, 0, 0, 0, 0, 0, 0)$ out of 10 possible outcomes. If not stated otherwise, all simulations were carried out with a mini-batch size of 100.

The benchmark floating point software-based simulation results on the Discriminative with 300, 500 and 6000 hidden units have been obtained in specific training and test conditions apart from the memristor-based simulations. During training when computing Contrastive Divergence, visible units are binarized while hidden units are encoded by probabilities as it has been empirically prescribed [175]. Only one step of Gibbs sampling was used to compute Contrastive Divergence (CD-1). At test time and contrary to memristor-based simulations, the inference technique is deterministic: when clamping a digit the network was not trained on, the label which is selected is the one which minimizes the free-energy $F(v)$ of the RBM:

$$F(v) = -\log \sum_h \exp\left(h^T W v\right), \tag{1.4}$$

with $p(v) = \exp\left(-F(v)\right)/\sum_{\tilde{v}} \exp\left(-F(\tilde{v})\right)$: a minimal free-energy corresponds to a maximal likelihood. The Discriminative RBMs with 300, 500 and 6000 hidden units have respectively been tuned with a learning rate of 0.01, 0.01 and 0.04.

We now specify memristor-based simulation conditions. As in [105], the conductance were sampled from $\mathcal{N}(\frac{G_{max}-G_{min}}{2}, \left(0.1\frac{G_{max}-G_{min}}{2}\right)^2)$ and the conductance bias were set to $\frac{G_{max}-G_{min}}{2}$. In any architecture, the neurons values during inference (i.e. during the forward pass at a given training step) and training (i.e. in the weight update itself) were binarized, i.e. stochastically sampled from their Bernoulli probability given their upstream neurons, using sigmoid activation functions. When greedily training the Deep Belief Net, we trained the 785+501 RBM taking as inputs binarized MNIST inputs, then the 501+501 RBM taking as inputs binarized features extracted by the first RBM, finally the 511+2001 Discriminative RBM taking as inputs features extracted by the second RBM and target labels. To perform inference in the Discriminative RBM on the label units to calculate the test error rate (this is strictly analogous in the Deep Belief Net, taking extracted features instead of the MNIST samples as inputs), we proceeded in the following way: we initially clamp a given test sample on the first 784 units (or extracted features of this test sample on the first 500 hidden units) along with a label vector on the remaining 10 units initialized to $(1/10)(1, \ldots, 1)$. We subsequently perform 40 Gibbs chains in parallel over 2 steps (exactly as in [139]) and average the resulting 40 label vectors to determine which label was selected by the network.

If not stated otherwise, simulations were performed over 30 epochs, over 5 trials, with error bars indicating median, first quartile, third quartile with a mini-batch size of 100.

Finally, the multiplicative coefficient η_+ appearing in Alg. (2) describing RProp was set empirically set to 1.01 and we fixed $\eta_- = 1/\eta_+$. We selected η_+ by simply drawing the cumulative distribution function of the final pulse widths and ensured that it was spread enough over the whole range $[0, \Delta t_{max}]$ - with $\eta = 1.05$ conversely, 40% of the device were shut off after 30 epochs of learning ($\Delta t = 0$).

All simulation scripts can be found on: https://github.com/ernoult/mem-RBM.git.

Chapter 2

Appendix of part IV

2.1 Difference between $\mathcal{L}^*$ and $\mathcal{L}$

There is a difference between the loss at the steady state $\mathcal{L}^*$ and the loss after T iterations $\mathcal{L}$. To see why the functions $\mathcal{L}^*$ and $\mathcal{L}$ (as functions of θ) are different, we have to come back to the definitions of s_* and s_T. Recall that

- $\mathcal{L}^* = \ell(s_*, y)$ where s_* is the steady state, i.e. characterized by $s_* = F(x, s_*, \theta)$,
- $\mathcal{L} = \ell(s_T, y)$ where s_T is the state of the network after T time steps, following the dynamics $s_0 = 0$ and $s_{t+1} = F(x, s_t, \theta)$.

For the current value of the parameter θ, the hyperparameter T is chosen such that $s_T = s_*$, i.e. such that the network reaches steady state after T time steps. Thus, for this value of θ we have numerical equality $\mathcal{L}(\theta) = \mathcal{L}^*(\theta)$. However, two functions that have the same value at a given point are not necessarily equal. Similarly, two functions that have the same value at a given point don't necessarily have the same gradient at that point. Here we are in the situation where

1. the functions $\mathcal{L}$ and $\mathcal{L}^*$ (as functions of θ) have the same value at the current value of θ, i.e. $\mathcal{L}(\theta) = \mathcal{L}^*(\theta)$ numerically,
2. the functions $\mathcal{L}$ and $\mathcal{L}^*$ (as functions of θ) are analytically different, i.e. $\mathcal{L} \neq \mathcal{L}^*$.

Since the functions $\mathcal{L}$ and $\mathcal{L}^*$ (as functions of θ) are different, the gradients $\frac{\partial \mathcal{L}^*}{\partial \theta}$ and $\frac{\partial \mathcal{L}}{\partial \theta}$ are also different in general.

2.2 Index Shift in the definition of $\Delta_\theta^{\mathrm{EP}}$ and $\nabla_\theta^{\mathrm{BPTT}}$

The convention that we have chosen to define $\nabla_\theta^{\mathrm{BPTT}}(t)$ and $\Delta_\theta^{\mathrm{EP}}(t)$ could seem strange at first glance for two reasons:

- the state update $\Delta_s^{\mathrm{EP}}(t)$ is defined in terms of s_t^β and s_{t+1}^β, whereas the weight update $\Delta_\theta^{\mathrm{EP}}(t)$ is defined in terms of s_{t-1}^β and s_t^β,
- at time $t = 0$, the state gradient $\nabla_s^{\mathrm{BPTT}}(0)$ and the state update $\Delta_s^{\mathrm{EP}}(0)$ are defined, but the weight gradient $\nabla_\theta^{\mathrm{BPTT}}(0)$ and the weight update $\Delta_\theta^{\mathrm{EP}}(0)$ are not defined.

Here we explain why.

First, recall that we have defined the gradients of BPTT as

$$\forall t = 0, 1, \dots, T, \qquad \nabla_s^{\mathrm{BPTT}}(t) = \frac{\partial \mathcal{L}}{\partial s_{T-t}}, \tag{2.1}$$

$$\forall t = 1, 2, \dots, T, \qquad \nabla_\theta^{\mathrm{BPTT}}(t) = \frac{\partial \mathcal{L}}{\partial \theta_{T-t}}, \tag{2.2}$$

where

$$\forall t = 0, 1, \dots T-1, \qquad s_{t+1} = F\left(x, s_t, \theta_t = \theta\right), \qquad \mathcal{L} = \ell\left(s_T, y\right), \tag{2.3}$$

and that we have defined the neural and weight updates of EP as

$$\forall t \geq 0, \qquad \Delta_s^{\mathrm{EP}}(t) = \lim_{\beta \to 0} \frac{1}{\beta} \left(s_{t+1}^\beta - s_t^\beta\right), \tag{2.4}$$

$$\forall t \geq 1, \qquad \Delta_\theta^{\mathrm{EP}}(t) = \lim_{\beta \to 0} \frac{1}{\beta} \left(\frac{\partial \Phi}{\partial \theta}\left(x, s_t^\beta, \theta\right) - \frac{\partial \Phi}{\partial \theta}\left(x, s_{t-1}^\beta, \theta\right)\right), \tag{2.5}$$

where

$$s_0^\beta = s_*, \qquad \forall t \geq 0, \quad s_{t+1}^\beta = F\left(x, s_t^\beta, \theta\right) - \beta \, \frac{\partial \ell}{\partial s}\left(s_t^\beta, y\right). \tag{2.6}$$

Index Shift

Let us introduce

$$\Phi^\beta(x, s, y, \theta) = \Phi(x, s, \theta) - \beta \, \ell(s, y), \tag{2.7}$$

so that the dynamics in the second phase rewrites

$$s_{t+1}^\beta = \frac{\partial \Phi^\beta}{\partial s}\left(x, s_t^\beta, y, \theta\right). \tag{2.8}$$

It is then readily seen that the neural updates Δ_s^{EP} and the weight updates $\Delta_\theta^{\mathrm{EP}}$ both rewrite in the form

$$\Delta_s^{\mathrm{EP}}(0) = \lim_{\beta\to 0} \frac{1}{\beta}\left(\frac{\partial \Phi^\beta}{\partial s}\left(x, s_0^\beta, y, \theta\right) - \frac{\partial \Phi}{\partial s}\left(x, s_0^\beta, \theta\right)\right), \tag{2.9}$$

$$\forall t \geq 1, \qquad \Delta_s^{\mathrm{EP}}(t) = \lim_{\beta\to 0} \frac{1}{\beta}\left(\frac{\partial \Phi^\beta}{\partial s}\left(x, s_t^\beta, y, \theta\right) - \frac{\partial \Phi^\beta}{\partial s}\left(x, s_{t-1}^\beta, y, \theta\right)\right), \tag{2.10}$$

$$\forall t \geq 1, \qquad \Delta_\theta^{\mathrm{EP}}(t) = \lim_{\beta\to 0} \frac{1}{\beta}\left(\frac{\partial \Phi^\beta}{\partial \theta}\left(x, s_t^\beta, y, \theta\right) - \frac{\partial \Phi^\beta}{\partial \theta}\left(x, s_{t-1}^\beta, y, \theta\right)\right). \tag{2.11}$$

Written in this form, we see a symmetry between $\Delta_s^{\mathrm{EP}}(t)$ and $\Delta_\theta^{\mathrm{EP}}(t)$ and there is no more index shift.

Missing Weight Gradient $\nabla_\theta^{\mathrm{BPTT}}(0)$ and Weight Update $\Delta_\theta^{\mathrm{EP}}(0)$

We can naturally extend the definition of $\nabla_\theta^{\mathrm{BPTT}}(0)$ and $\Delta_\theta^{\mathrm{EP}}(0)$ following Eq. 2.2. In the setting studied in this paper, they both take the value 0 because the cost function $\ell(s, y)$ does not depend on the parameter θ. But suppose now that ℓ depends on θ, i.e. that ℓ is of the form $\ell(s, y, \theta)$. Then the loss of Eq. 2.3 takes the form $\mathcal{L} = \ell\left(s_T, y, \theta_T = \theta\right)$, so that:

$$\nabla_\theta^{\mathrm{BPTT}}(0) = \frac{\partial \mathcal{L}}{\partial \theta_T} = \frac{\partial \ell}{\partial \theta}\left(s_T, y, \theta\right). \tag{2.12}$$

As for the missing weight update $\Delta_\theta^{\mathrm{EP}}(0)$, we follow the definition of Eq. 2.9 and define:

$$\Delta_\theta^{\mathrm{EP}}(0) = \lim_{\beta\to 0} \frac{1}{\beta}\left(\frac{\partial \Phi^\beta}{\partial \theta}\left(x, s_0^\beta, y, \theta\right) - \frac{\partial \Phi}{\partial \theta}\left(x, s_0^\beta, \theta\right)\right) = -\frac{\partial \ell}{\partial \theta}\left(s_*, y, \theta\right). \tag{2.13}$$

Since $s_T = s_*$ (the state at the end of the first phase is the state at the beginning of the second phase, and it is the steady state), we have $\Delta_\theta^{\mathrm{EP}}(0) = -\nabla_\theta^{\mathrm{BPTT}}(0)$.

2.3 Experiments: demonstrating the GDU property

2.3.1 Hyperparameters

We provide in Table 2.1 a complete description of the hyperparameters that were used to demonstrate Theorem 4 on the different models.

Table 2.1: Table of hyperparameters used to demonstrate Theorem 4. "EB" and "P" respectively denote "energy-based" and "prototypical", "-#h" stands for the number of hidden layers.

	Activation	T	K	β	ε
Toy model	tanh	5000	80	0.01	0.08
EB-1h	tanh	800	80	0.001	0.08
EB-2h	tanh	5000	150	0.01	0.08
EB-3h	tanh	30000	200	0.02	0.08
P-1h	tanh	150	10	0.01	-
P-2h	tanh	1500	40	0.01	-
P-3h	tanh	5000	40	0.015	-
P-conv	hard sigmoid	5000	10	0.02	-

2.3.2 Definition of the Relative Mean Squared Error (RMSE)

We introduce a relative mean squared error (RelMSE) * between two continuous functions f and g in a given layer L as:

$$\text{RelMSE(f, g)} = \left\langle \frac{\|f-g\|_{2,K}}{\max(\|f\|_{2,K}, \|g\|_{2,K})} \right\rangle_L , \tag{2.14}$$

where $\|f\|_{2,K} = \sqrt{\frac{1}{K}\int_0^K f^2(t)dt}$ and $\langle\cdot\rangle_L$ denotes an average over all the elements of layer L. For example, $\text{RelMSE}(\Delta^{\text{EP}}_{W_{01}}, -\nabla^{\text{BPTT}}_{W_{01}})$ averages the squared distance between $\Delta^{\text{EP}}_{W_{01}}$ and $-\nabla^{\text{BPTT}}_{W_{01}}$ averaged over all the elements of W_{01}. Also, instead of computing Δ^{EP} and ∇^{BPTT} processes on a single sample presentation and bias the RelMSE by the choice of this sample, Δ^{EP} and ∇^{BPTT} processes have been averaged over a mini-batch of 20 samples before their distance in terms of RelMSE was measured.

2.3.3 Why are the ∇^{BPTT}_s and Δ^{EP}_s saw-teeth shaped in the prototypical setting ?

In the prototypical setting, in the case of a layered architecture (without lateral and skip-layer connections), the ∇^{BPTT} and Δ^{EP} processes are saw teeth shaped, i.e. they take the value zero every other time step (as seen per Fig. 4.12, Fig. 4.6, Fig. 4.7 and Fig. 4.8). We

*We choose the RelMSE metric rather than a more conventional one such as the cos metric. Indeed, although the cos metric is also meaningful, it lacks an important property in our context: the cos between f and g is maximal if and only if f and g are proportional, whereas we aim at reaching equality (Theorem 4). In contrast, our RelMSE metric is such that $\text{RelMSE}(f,g) = 0 \Leftrightarrow f(t) = g(t)$.

provide an explanation for this phenomenon both from the point of view of BPTT and from the point of view of EP. Fig. 2.1 illustrates this phenomenon in the case of a network with two layers: one output layer s^0 and one hidden layer s^1.

- **Point of view of BPTT.** In the forward-time pass (first phase), s^0_{t+1} is determined by s^1_t, while s^1_{t+1} is determined by s^0_t. This gives rise to a zig-zag shaped connectivity pattern in the computational graph of the the network unrolled in time (Fig. 2.1). In particular, the gray nodes of Fig. 2.1 are not involved in the computation of the loss $\mathcal{L}$, i.e. their gradients are equal to zero. In other words $\nabla^{\mathrm{BPTT}}_{s^1}(0) = 0$, $\nabla^{\mathrm{BPTT}}_{s^0}(1) = 0$, $\nabla^{\mathrm{BPTT}}_{s^1}(2) = 0$, etc.
- **Point of view of EP.** At the beginning of the second phase (at time step $t = 0$), the network is at the steady state ; in particular $s^{1,\beta}_0 = s^1_*$. At time step $t = 1$, only the output layer s^0 is influenced by y ; the hidden layer s^1 is still at the steady state, i.e. $s^{1,\beta}_1 = s^1_*$. From $s^{1,\beta}_0 = s^{1,\beta}_1$, it follows that $s^{0,\beta}_1 = s^{0,\beta}_2$. In turn, from $s^{0,\beta}_1 = s^{0,\beta}_2$ it follows that $s^{1,\beta}_2 = s^{1,\beta}_3$. Etc. In other words $\Delta^{\mathrm{EP}}_{s^1}(0) = 0$, $\Delta^{\mathrm{EP}}_{s^0}(1) = 0$, $\Delta^{\mathrm{EP}}_{s^1}(2) = 0$, etc.

The above argument can be generalized to an arbitrary number of layers. In this case we group the layers of even index (resp. odd index) together. We call $e_t = (s^0_t, s^2_t, s^4_t, \ldots)$ and $o_t = (s^1_t, s^3, t, s^5_t, \ldots)$. The crucial property is that o_{t+1} (resp. e_{t+1}) is determined by e_t (resp. o_t).

One consequence of this analysis is that, in the prototypical setting of EP, half of the computations are redundant and could be avoided. Avoiding such redundant computations would lead to an implementation where the layers of even indices and the layers of odd indices are updated alternatively, similar to the one proposed in section 4.3 of [176].

In contrast, the saw teeth shaped curves are not observed in the energy based setting. This is due to the different topology of the computational graph in this setting. In the energy-based setting, the assumptions under which we have shown the saw teeth shape are not satisfied since neurons are subject to leakage, e.g. s^1_{t+1} depends not just on s^0_t but also on s^1_t. Therefore the reasoning developed above no longer holds.

2.4 Training experiments

Simulation framework. Simulations have been carried out in Pytorch. The code has been attached to the supplementary materials upon submitting this work on the CMT interface. We have also attached a readme.txt with a specification of all dependencies, packages, descriptions of the python files as well as the commands to reproduce all the results presented in this paper.

Data set. Training experiments were carried out on the MNIST data set. Training set and test set include 60000 and 10000 samples respectively.

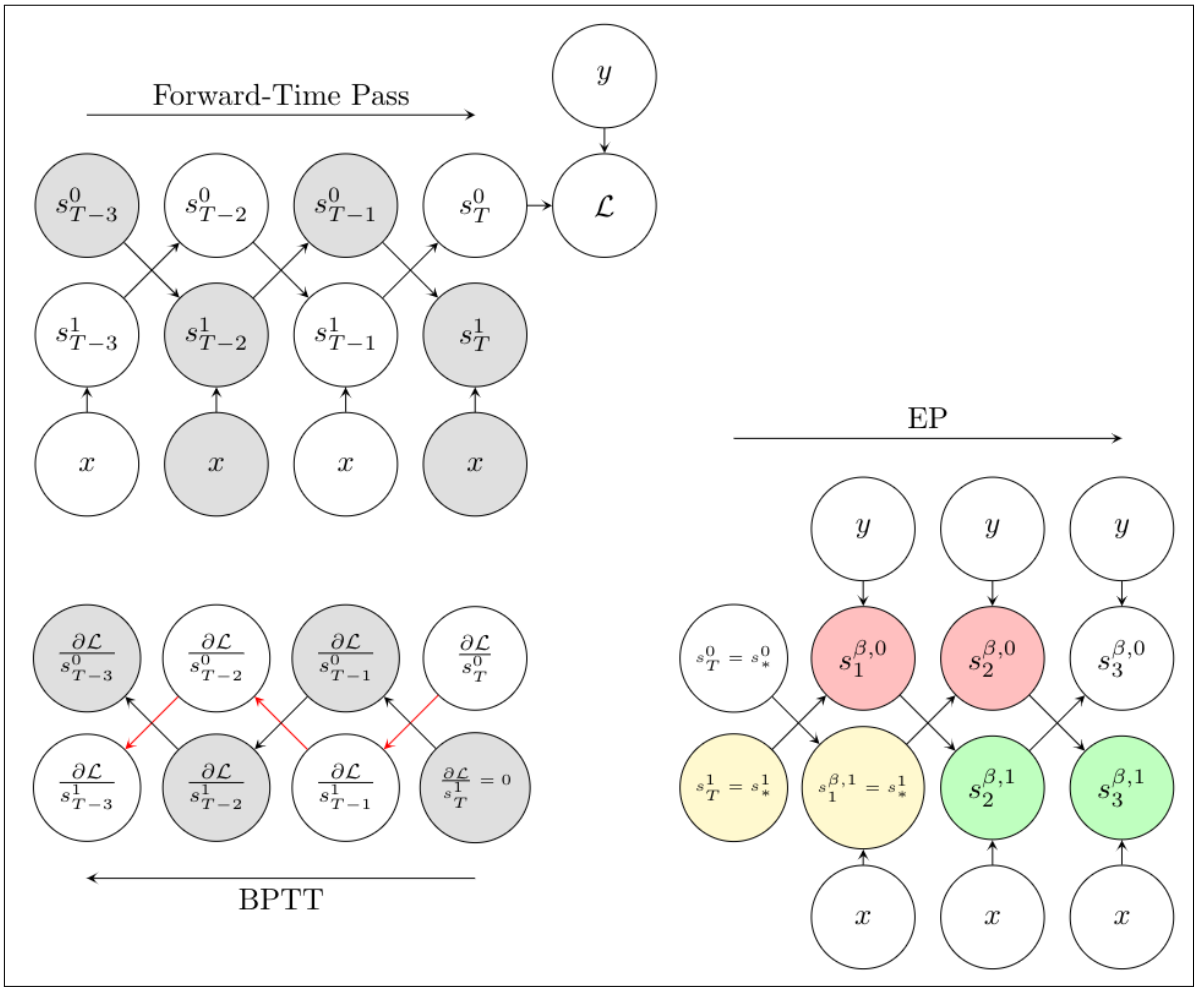

Figure 2.1: Explanation of the saw teeth shape of the ∇_s^{BPTT} and Δ_s^{EP} processes in the prototypical setting (layered architecture without lateral or skip-layer connections). **Forward-time pass (top left)**: gray nodes in the computational graph indicate nodes that are not involved in the computation of the loss $\mathcal{L}$. **BPTT (bottom left)**: red arrows indicate the differentiation path through the output units s^0. The gradients in the gray nodes are equal to 0. **EP (bottom right)**: nodes of the same color have the same value.

Optimization. Optimization was performed using stochastic gradient descent with mini-batches of size 20. For each simulation, weights were Glorot-initialized. No regularization technique was used and we did not use the persistent trick of caching and reusing converged states for each data sample between epochs as in [160].

Hyperparameter search for EP. We distinguish between two kinds of hyperparameters: the recurrent hyperparameters - i.e. T, K and β - and the learning rates. A first guess of the recurrent hyperparameters T and β is found by plotting the Δ^{EP} and ∇^{BPTT} processes associated to synapses and neurons to see qualitatively whether the theorem is approximately

satisfied, and by conjointly computing the proportions of synapses whose Δ_W^{EP} processes have the same sign as its ∇_W^{BPTT} processes. K can also be found out of the plots as the number of steps which are required for the gradients to converge. Morever, plotting these processes reveal that gradients are vanishing when going away from the output layer, i.e. they lose up to 10^{-1} in magnitude when going from a layer to the previous (i.e. upstream) layer. We subsequently initialized the learning rates with increasing values going from the output layer to upstreams layers. The typical range of learning rates is $[10^{-3}, 10^{-1}]$, $[10, 1000]$ for T, $[2, 100]$ for K and $[0.01, 1]$ for β. Hyperparameters where adjusted until having a train error the closest to zero. Finally, in order to obtain minimal recurrent hyperparameters - i.e. smallest T and K possible, both in the energy-based and prototypical setting for a fair comparison - we progressively decreased T and K until the train error increases again.

Activation functions, update clipping. For training, we used two kinds of activation functions:

- $\sigma(x) = \frac{1}{1+\exp(-4(x-1/2))}$. Although it is a shifted and rescaled sigmoid function, we shall refer to this activation function as 'sigmoid'.

- $\sigma(x) = \max(\min(x, 1), 0)$. It is the 'hard' version of the previous activation function so that we call it here for convenience 'hard sigmoid'.

The sigmoid function was used for all the training simulations except the convolutional architecture for which we used the hard sigmoid function - see Table 2.2. Also, similarly to [160], for the energy-based setting we clipped the neuron updates between 0 and 1 so that at each time step, when an update Δs was prescribed, we have implemented: $s \leftarrow \max(\min(s + \Delta s, 1), 0)$.

Benchmarking EP with respect to BPTT. In order to compare EP and BPTT directly, for each simulation trial we used the same weight initialization to train the network with EP on the one hand, and with BPTT on the other hand. We also used the same learning rates, and the same recurrent hyperparameters: we used the same T for both algorithms, and we truncated BPTT to K steps, as prescribed by the theory.

Table 2.2: Table of hyperparameters used for training. "EB" and "P" respectively denote "energy-based" and "prototypical", "-#h" stands for the number of hidden layers.

	Activation	T	K	β	ε	Epochs	Learning rates
EB-1h	sigmoid	100	12	0.5	0.2	30	0.1-0.05
EB-2h	sigmoid	500	40	0.8	0.2	50	0.4-0.1-0.01
P-1h	sigmoid	30	10	0.1	-	30	0.08-0.04
P-2h	sigmoid	100	20	0.5	-	50	0.2-0.05-0.005
P-3h	sigmoid	180	20	0.5	-	100	0.2-0.05-0.01-0.002
P-conv	hard sigmoid	200	10	0.4	-	40	0.15-0.035-0.015

Algorithm 9 Discrete-time Equilibrium Propagation (EP)

Input: static input x, parameter θ, learning rate α.
Output: parameter θ.

while θ not converged **do**
  for each mini-batch x **do**
    $\Delta\theta \leftarrow 0$
    for $t \in [1, T]$ **do**
      $s_{t+1} \leftarrow \frac{\partial \Phi}{\partial s}(x, s_t, \theta)$ ▷ 1st phase: common to EP and BPTT
    end for
    for $t \in [1, K]$ **do**
      $s^{\beta}_{t+1} \leftarrow \frac{\partial \Phi^{\beta}}{\partial s}(x, s_t, \theta)$ ▷ 2nd phase: *forward-time* computation
      $\Delta^{\text{EP}}_{\theta} \leftarrow \frac{1}{\beta}\left(\frac{\partial \Phi}{\partial \theta}(x, s^{\beta}_{t+1}, \theta) - \frac{\partial \Phi}{\partial \theta}(x, s^{\beta}_{t}, \theta)\right)$
      $\Delta\theta \leftarrow \Delta\theta + \Delta^{\text{EP}}_{\theta}$
    end for
    $\theta \leftarrow \theta + \alpha\Delta\theta$
  end for
end while

Algorithm 10 Backpropagation Through Time (BPTT)

Input: static input x, parameter θ, learning rate α.
Output: parameter θ.

while θ not converged **do**
  for each mini-batch x **do**
    $\Delta\theta \leftarrow 0$
    for $t \in [1, T]$ **do**
      $s_{t+1} \leftarrow \frac{\partial \Phi}{\partial s}(x, s_t, \theta)$ ▷ 1st phase: common to EP and BPTT
    end for
    for $t \in [1, K]$ **do**
      $\nabla_\theta^{\text{BPTT}} \leftarrow \frac{\partial \mathcal{L}}{\partial \theta_{T-t}}$ ▷ 2nd phase: *backward-time* computation
      $\Delta\theta \leftarrow \Delta\theta + \nabla_\theta^{\text{BPTT}}$
    end for
    $\theta \leftarrow \theta - \alpha\Delta\theta$
  end for
end while

2.4.1 Training Curves

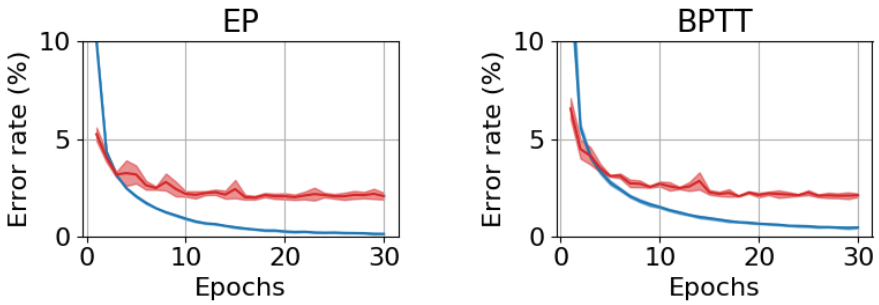

Figure 2.2: Train and test error achieved on MNIST by the fully connected layered architecture with one hidden layer (784-512-10) in the energy-based setting throughout learning, over five trials. Plain lines indicate mean, shaded zones delimiting mean plus/minus standard deviation.

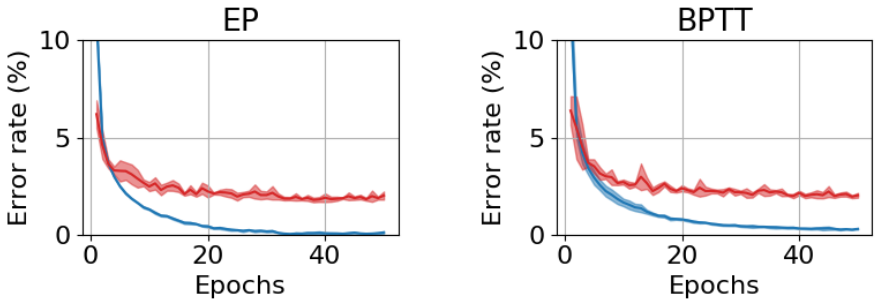

Figure 2.3: Train and test error achieved on MNIST by the fully connected layered architecture with two hidden layers (784-512-512-10) in the energy-based setting throughout learning, over five trials. Plain lines indicate mean, shaded zones delimiting mean plus/minus standard deviation.

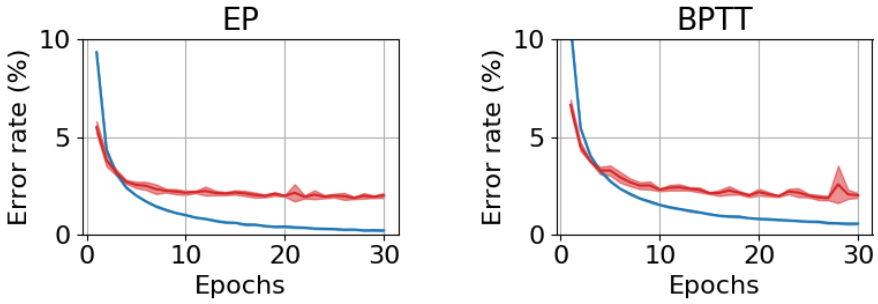

Figure 2.4: Train and test error achieved on MNIST by the fully connected layered architecture with one hidden layer (784-512-10) in the prototypical setting throughout learning, over five trials. Plain lines indicate mean, shaded zones delimiting mean plus/minus standard deviation.

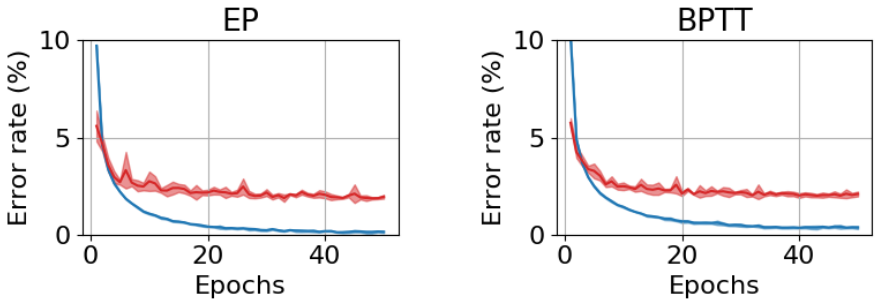

Figure 2.5: Train and test error achieved on MNIST by the fully connected layered architecture with two hidden layers (784-512-512-10) in the prototypical setting throughout learning, over five trials. Plain lines indicate mean, shaded zones delimiting mean plus/minus standard deviation.

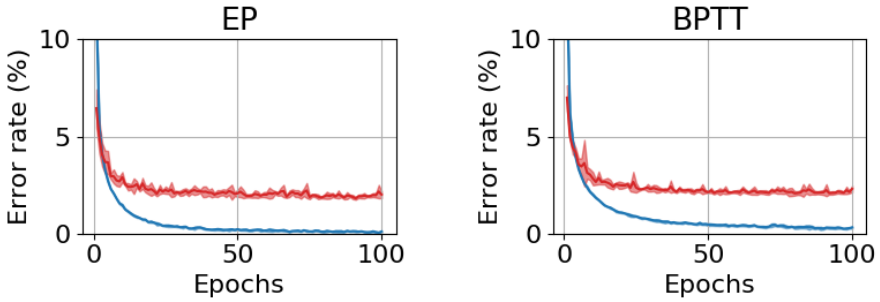

Figure 2.6: Train and test error achieved on MNIST by the fully connected layered architecture with three hidden layers (784-512-512-512-10) in the prototypical setting throughout learning, over five trials. Plain lines indicate mean, shaded zones delimiting mean plus/minus standard deviation.

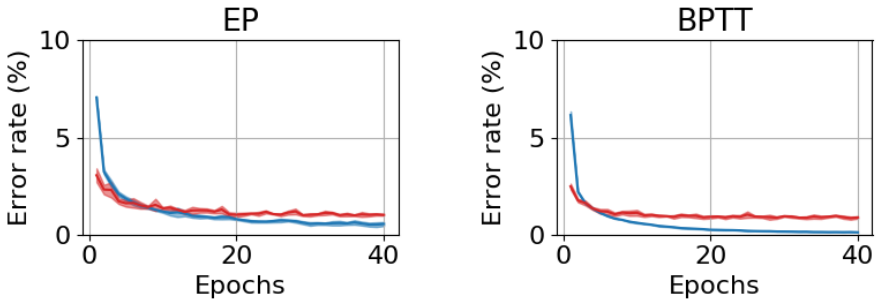

Figure 2.7: Train and test error achieved on MNIST by the convolutional architecture in the prototypical setting throughout learning, over five trials. Plain lines indicate mean, shaded zones delimiting mean plus/minus standard deviation.

Chapter 3

Appendix of part V

3.1 What 'Gradients' are the Gradients of RBP?

In this subsection we motivate the name of 'gradients' for the quantities $\nabla_s^{\mathrm{RBP}}(t)$ and $\nabla_\theta^{\mathrm{RBP}}(t)$ by proving that they are the gradients of $\mathcal{L}^*$ in the sense of Proposition 12 below. They are also the gradients of what we call the 'projected cost function' (Proposition 13), using the terminology of [148].

Proposition 12 (RBP Optimizes $\mathcal{L}^*$)**.** *The total gradient computed by the RBP algorithm is the gradient of the loss $\mathcal{L}^* = \ell(s_*, y)$, i.e.*

$$\sum_{t=1}^{\infty} \nabla_\theta^{\mathrm{RBP}}(t) = \frac{\partial \mathcal{L}^*}{\partial \theta}. \tag{3.1}$$

$\nabla_s^{\mathrm{RBP}}(t)$ and $\nabla_\theta^{\mathrm{RBP}}(t)$ can also be expressed as gradients of $\mathcal{L}_t = \ell(s_t, y)$, the cost after t time steps. In the terminology of [148], $\mathcal{L}_t$ was named the *projected cost*. For $t = 0$, $\mathcal{L}_0$ is simply the cost of the initial state s_0. For $t > 0$, $\mathcal{L}_t$ is the cost of the state projected a duration t in the future.

Proposition 13 (Gradients of RBP are Gradients of the Projected Cost)**.** *The 'RBP gradients' $\nabla_s^{\mathrm{RBP}}(t)$ and $\nabla_\theta^{\mathrm{RBP}}(t)$ can be expressed as gradients of the projected cost:*

$$\forall t \geq 0, \qquad \nabla_s^{\mathrm{RBP}}(t) = \left.\frac{\partial \mathcal{L}_t}{\partial s_0}\right|_{s_0 = s_*}, \qquad \nabla_\theta^{\mathrm{RBP}}(t) = \left.\frac{\partial \mathcal{L}_t}{\partial \theta_0}\right|_{s_0 = s_*} \tag{3.2}$$

where the initial state s_0 is the steady state s_*.

Proof of Proposition 12. First of all, by Definition 6 (Eq. 2.5-2.7) it is straightforward to see that

$$\forall t \geq 0, \qquad \nabla_s^{\mathrm{RBP}}(t) = \left(\frac{\partial F}{\partial s}\left(x, s_*, \theta\right)^\top\right)^t \cdot \frac{\partial \ell}{\partial s}\left(s_*, y\right), \tag{3.3}$$

$$\forall t \geq 1, \qquad \nabla_\theta^{\mathrm{RBP}}(t) = \frac{\partial F}{\partial \theta}\left(x, s_*, \theta\right)^\top \cdot \left(\frac{\partial F}{\partial s}\left(x, s_*, \theta\right)^\top\right)^{t-1} \cdot \frac{\partial \ell}{\partial s}\left(s_*, y\right). \tag{3.4}$$

Second, recall that the loss $\mathcal{L}^*$ is

$$\mathcal{L}^* = \ell\left(s_*, y\right), \tag{3.5}$$

where

$$s_* = F\left(x, s_*, \theta\right). \tag{3.6}$$

By the chain rule of differentiation, the gradient of $\mathcal{L}^*$ (Eq. 3.5) is

$$\frac{\partial \mathcal{L}^*}{\partial \theta} = \frac{\partial \ell}{\partial s}\left(s_*, y\right) \cdot \frac{\partial s_*}{\partial \theta}. \tag{3.7}$$

In order to compute $\frac{\partial s_*}{\partial \theta}$, we differentiate the steady state condition (Eq. 3.6) with respect to θ, which yields

$$\frac{\partial s_*}{\partial \theta} = \frac{\partial F}{\partial s}\left(x, s_*, \theta\right) \cdot \frac{\partial s_*}{\partial \theta} + \frac{\partial F}{\partial \theta}\left(x, s_*, \theta\right). \tag{3.8}$$

Rearranging the terms, and using the Taylor expansion $\left(\mathrm{Id} - A\right)^{-1} = \sum_{t=0}^{\infty} A^t$ with $A = \frac{\partial F}{\partial s}\left(x, s_*, \theta\right)$, we get

$$\frac{\partial s_*}{\partial \theta} = \left(\mathrm{Id} - \frac{\partial F}{\partial s}\left(x, s_*, \theta\right)\right)^{-1} \cdot \frac{\partial F}{\partial \theta}\left(x, s_*, \theta\right) \tag{3.9}$$

$$= \sum_{t=0}^{\infty} \left(\frac{\partial F}{\partial s}\left(x, s_*, \theta\right)\right)^t \cdot \frac{\partial F}{\partial \theta}\left(x, s_*, \theta\right). \tag{3.10}$$

Therefore

$$\frac{\partial \mathcal{L}^*}{\partial \theta} = \frac{\partial \ell}{\partial s}\left(s_*, y\right) \cdot \frac{\partial s_*}{\partial \theta} \tag{3.11}$$

$$= \sum_{t=0}^{\infty} \frac{\partial \ell}{\partial s}\left(s_*, y\right) \cdot \left(\frac{\partial F}{\partial s}\left(x, s_*, \theta\right)\right)^t \cdot \frac{\partial F}{\partial \theta}\left(x, s_*, \theta\right) \tag{3.12}$$

$$= \sum_{t=0}^{\infty} \nabla_\theta^{\mathrm{RBP}}(t). \tag{3.13}$$

□

Proof of Proposition 13. By the chain rule of differentiation we have

$$\frac{\partial \mathcal{L}_{t+1}}{\partial s_0} = \frac{\partial F}{\partial s}\left(x, s_0, \theta\right)^\top \cdot \frac{\partial \mathcal{L}_{t+1}}{\partial s_1}. \tag{3.14}$$

Evaluation this expression for $s_0 = s_*$ we get

$$\left.\frac{\partial \mathcal{L}_{t+1}}{\partial s_0}\right|_{s_0 = s_*} = \frac{\partial F}{\partial s}\left(x, s_*, \theta\right)^\top \cdot \left.\frac{\partial \mathcal{L}_{t+1}}{\partial s_1}\right|_{s_0 = s_*}. \tag{3.15}$$

Finally note that

$$\left.\frac{\partial \mathcal{L}_{t+1}}{\partial s_1}\right|_{s_0=s_*} = \left.\frac{\partial \mathcal{L}_{t+1}}{\partial s_1}\right|_{s_1=s_*} = \left.\frac{\partial \mathcal{L}_t}{\partial s_0}\right|_{s_0=s_*} \tag{3.16}$$

Therefore $\left.\frac{\partial \mathcal{L}_t}{\partial s_0}\right|_{s_0=s_*}$ and $\nabla_s^{\mathrm{RBP}}(t)$ satisfy the same recurrence relation, thus they are equal. Proving the equality of $\left.\frac{\partial \mathcal{L}_t}{\partial \theta_0}\right|_{s_0=s_*}$ and $\nabla_\theta^{\mathrm{RBP}}(t)$ is analogous. □

3.2 Experiments: demonstrating the GDD property

We provide the full table of hyperparameters used to demonstrate the GDD property on the different models defined - Table 3.1.

Table 3.1: Table of hyperparameters used to demonstrate Theorem 10.

	Figure	Angle Ψ (°)	Activation	T	K	β	ε	η
C-EP	4.1	0	tanh	800	80	0.01	0.08	$1.510^{-6} - 1.510^{-6}$
C-VF	4.1	45	tanh	800	80	0.01	0.08	$1.510^{-6} - 1.510^{-6}$
C-EP	4.1	0	tanh	800	80	0.01	0.08	$1.510^{-5} - 1.510^{-5}$
C-VF	4.1	45	tanh	800	80	0.01	0.08	$1.510^{-5} - 1.510^{-5}$
C-VF	3.5-3.6	0	tanh	800	80	0.005	0.08	$2.10^{-6} - 2.10^{-6}$
C-VF	3.5-3.6	0	tanh	800	80	0.005	0.08	$2.10^{-5} - 2.10^{-5}$
C-VF	3.7-3.8	0	tanh	150	10	0.01	–	$2.10^{-6} - 2.10^{-6}$
C-VF	3.7-3.8	0	tanh	150	10	0.01	–	$2.10^{-5} - 2.10^{-5}$
C-EP	3.1-3.2	0	tanh	800	80	0.05	0.08	$2.10^{-6} - 2.10^{-6}$
C-EP	3.1-3.2	0	tanh	800	80	0.05	0.08	$2.10^{-5} - 2.10^{-5}$
C-EP	3.3-3.4	0	tanh	150	10	0.01	–	$2.10^{-6} - 2.10^{-6}$
C-EP	3.3-3.4	0	tanh	150	10	0.01	–	$2.10^{-5} - 2.10^{-5}$

3.3 Illustrating the equivalence of the four algorithms on an analytically tractable model

Model. To illustrate the equivalence of the four algorithms (BPTT, RBP, EP and CEP), we study a simple model with scalar variable s and scalar parameter θ:

$$s_0 = 0, \qquad s_{t+1} = \frac{1}{2}\left(s_t + \theta\right), \qquad \mathcal{L}^* = \frac{1}{2}s_*^2, \tag{3.17}$$

where s_* is the steady state of the dynamics (it is easy to see that the solution is $s_* = \theta$). The dynamics rewrites $s_{t+1} = F\left(s_t, \theta\right)$ with the transition function $F\left(s, \theta\right) = \frac{1}{2}\left(s + \theta\right)$, and the loss rewrites $\mathcal{L}^* = \ell\left(s_*\right)$ with the cost function $\ell(s) = \frac{1}{2}s^2$. Furthermore, a primitive

function of the system * is $\Phi(s,\theta) = \frac{1}{4}(s+\theta)^2$. This model has no practical application ; it is only meant for pedagogical purpose.

Backpropagation Through Time (BPTT). With BPTT, an important point is that we approximate the steady state s_* by the state after T time steps s_T, and we approximate $\mathcal{L}^*$ (the loss at the steady state) by the loss after T time steps $\mathcal{L} = \ell(s_T)$.

In order to compute (i.e. 'backpropagate') the gradients of BPTT, Proposition 1 tells us that we need to compute $\frac{\partial \ell}{\partial s}(s_T) = s_T$, $\frac{\partial F}{\partial s}(s_t,\theta) = \frac{1}{2}$ and $\frac{\partial F}{\partial \theta}(s_t,\theta) = \frac{1}{2}$. We get

$$\forall t = 0,1,\dots,T-1, \qquad \nabla_s^{\mathrm{BPTT}}(t) = \frac{s_T}{2^t}, \qquad \nabla_\theta^{\mathrm{BPTT}}(t) = \frac{s_T}{2^{t+1}}. \tag{3.18}$$

Recurrent Backpropagation (RBP). Similarly, to compute the gradients of RBP, Definition 6 tells us that we need to compute $\frac{\partial \ell}{\partial s}(s_*) = s_*$, $\frac{\partial F}{\partial s}(s_*,\theta) = \frac{1}{2}$ and $\frac{\partial F}{\partial \theta}(s_*,\theta) = \frac{1}{2}$. We have

$$\forall t \geq 0, \qquad \nabla_s^{\mathrm{RBP}}(t) = \frac{s_*}{2^t}, \qquad \nabla_\theta^{\mathrm{RBP}}(t) = \frac{s_*}{2^{t+1}}. \tag{3.19}$$

The state after T time steps in BPTT converges to the steady state s_* as $T \to \infty$, therefore the gradients of BPTT converge to the gradients of RBP. Also notice that the steady state of the dynamics is $s_* = \theta$.

Equilibrium Propagation (EP). Following the equations governing the second phase of EP (Fig. 1.1), we have:

$$s_0^\beta = \theta, \qquad s_{t+1}^\beta = \left(\frac{1}{2} - \beta\right) s_t^\beta + \frac{1}{2}\theta. \tag{3.20}$$

This linear dynamical system can be solved analytically:

$$\forall t \geq 0, \qquad s_t^\beta = \frac{\theta}{1+2\beta}\left(1 + 2\beta\left(\frac{1}{2} - \beta\right)^t\right). \tag{3.21}$$

Notice that $s_t^\beta \to \theta$ as $\beta \to 0$; for small values of the hyperparameter β, the trajectory in the second phase is close to the steady state $s_* = \theta$.

Using Eq. 2.14, it follows that the normalized updates of EP are

$$\forall t \geq 0, \qquad \Delta_s^{\mathrm{EP}}(\beta,t) = -\frac{\theta}{2^t}(1-2\beta)^t, \qquad \Delta_\theta^{\mathrm{EP}}(\beta,t) = -\frac{\theta}{2^{t+1}}(1-2\beta)^t. \tag{3.22}$$

Notice again that the normalized updates of EP converge to the gradients of RBP as $\beta \to 0$.

*The primitive function Φ is determined up to a constant.

Continual Equilibrium Propagation (C-EP). The system of equations governing the system is:

$$\begin{cases} s_0^{\beta,\eta} = s_*, \\ \theta_0^{\beta,\eta} = \theta, \end{cases} \qquad \forall t \geq 0 : \begin{cases} s_{t+1}^{\beta,\eta} = \left(\frac{1}{2} - \beta\right) s_t^{\beta,\eta} + \frac{1}{2}\theta_t^{\beta,\eta}, \\ \theta_{t+1}^{\beta,\eta} = \theta_t^{\beta,\eta} + \frac{\eta}{2\beta}\left(s_{t+1}^{\beta,\eta} - s_t^{\beta,\eta}\right). \end{cases} \tag{3.23}$$

First, rearranging the terms in the second equation, we get

$$\frac{1}{\eta}\left(\theta_{t+1}^{\beta,\eta} - \theta_t^{\beta,\eta}\right) = \frac{1}{2\beta}\left(s_{t+1}^{\beta,\eta} - s_t^{\beta,\eta}\right). \tag{3.24}$$

It follows that

$$\Delta_\theta^{\text{C-EP}}(\beta, \eta, t) = \frac{1}{2}\Delta_s^{\text{C-EP}}(\beta, \eta, t). \tag{3.25}$$

Therefore, all we need to do is to compute $\Delta_s^{\text{C-EP}}(\beta, \eta, t)$. Second, by iterating the second equation over all indices from $t = 0$ to $t - 1$ we get

$$\theta_t^{\beta,\eta} = \theta + \frac{\eta}{2\beta}\left(s_t^{\beta,\eta} - s_*\right). \tag{3.26}$$

Using $s_* = \theta$ and plugging this into the first equation we get

$$s_{t+1}^{\beta,\eta} = \left(\frac{1}{2} - \beta + \frac{\eta}{4\beta}\right) s_t^{\beta,\eta} + \left(\frac{1}{2} - \frac{\eta}{4\beta}\right)\theta. \tag{3.27}$$

Solving this linear dynamical system, and using the initial condition $s_0^{\beta,\eta} = \theta$ we get

$$s_t^{\beta,\eta} = \frac{\theta}{1 - \frac{\eta}{2\beta} + 2\beta}\left[1 - \frac{\eta}{2\beta} + 2\beta\left(\frac{1}{2}\right)^t\left(1 - 2\beta + \frac{\eta}{2\beta}\right)^t\right] \tag{3.28}$$

Finally:

$$\Delta_s^{\text{C-EP}}(\beta, \eta, t) = -\frac{\theta}{2^t}\left(1 - 2\beta + \frac{\eta}{2\beta}\right)^t \tag{3.29}$$

3.4 Experimental Details

3.4.1 Training experiments (Table 4.1)

Simulation framework. Simulations have been carried out in Pytorch. The code has been attached to the supplementary materials upon submitting this work on OpenReview. We have also attached a readme.txt with a specification of all dependencies, packages, descriptions of the python files as well as the commands to reproduce all the results presented in this paper.

Data set. Training experiments were carried out on the MNIST data set. Training set and test set include 60000 and 10000 samples respectively.

Optimization. Optimization was performed using stochastic gradient descent with mini-batches of size 20. For each simulation, weights were Glorot-initialized. No regularization technique was used and we did not use the persistent trick of caching and reusing converged states for each data sample between epochs as in [160].

Activation function. For training, we used the activation function

$$\sigma(x) = \frac{1}{1 + \exp(-4(x - 1/2))}. \tag{3.30}$$

Although it is a shifted and rescaled sigmoid function, we shall refer to this activation function as 'sigmoid'.

Use of a randomized β. The option 'Random β' appearing in the detailed table of results (Table 3.3) refers to the following procedure. During training, instead of using the same β accross mini-batches, we only keep the same *absolute value* of β and sample its sign from a Bernoulli distribution of probability $\frac{1}{2}$ at each mini-batch iteration. This procedure was hinted at by [160] to improve test error, and is used in our context to improve the model convergence for Continual Equilibrium Propagation - appearing as C-EP and C-VF in Table 4.1 - training simulations.

Tuning the angle between forward and backward weights. In Table 4.1, we investigate C-VF initialized with different angles between the forward and backward weights - denoted as Ψ in Table 4.1. Denoting them respectively θ_{f} and θ_{b}, the *angle* κ between them is defined here as:

$$\kappa(\theta_{\mathrm{f}}, \theta_{\mathrm{b}}) = \cos^{-1}\left(\frac{\mathrm{Tr}(\theta_{\mathrm{f}} \cdot \theta_{\mathrm{b}}^{\top})}{\sqrt{\mathrm{Tr}(\theta_{\mathrm{f}} \cdot \theta_{\mathrm{f}}^{\top})}\sqrt{\mathrm{Tr}(\theta_{\mathrm{b}} \cdot \theta_{\mathrm{b}}^{\top})}}\right),$$

where Tr denotes the trace, i.e. $\mathrm{Tr}(A) = \sum_i A_{ii}$ for any squared matrix A. To tune arbitrarily well enough $\kappa(\theta_{\mathrm{f}}, \theta_{\mathrm{b}})$, the procedure is the following: starting from $\theta_b = \theta_f$, i.e. $\kappa(\theta_{\mathrm{f}}, \theta_{\mathrm{b}}) = 0$, we can gradually increase the angle between θ_{f} and θ_{b} by flipping the sign of an arbitrary proportion of components of θ_{b}. The more components have their sign flipped, the larger is the angle. More formally, we write θ_b in the form $\theta_b = M(p) \odot \theta_{\mathrm{f}}$ and we define:

$$\Psi(p) = \kappa(\theta_{\mathrm{f}}, M(p) \odot \theta_{\mathrm{f}}), \tag{3.31}$$

where $M(p)$ is a mask of binary random values {+1, -1} of the same dimension of θ_{f}: $M(p) = -1$ with probability p and $M(p) = +1$ with probability $1 - p$. Taking the cosine and the expectation of Eq. (3.31), we obtain:

$$\begin{aligned}\langle\cos(\Psi(p))\rangle &= p \times -\frac{\mathrm{Tr}(\theta_\mathrm{f}\cdot\theta_\mathrm{f}^\top)}{\mathrm{Tr}(\theta_\mathrm{f}\cdot\theta_\mathrm{f}^\top)} + (1-p)\times\frac{\mathrm{Tr}(\theta_\mathrm{f}\cdot\theta_\mathrm{f}^\top)}{\mathrm{Tr}(\theta_\mathrm{f}\cdot\theta_\mathrm{f}^\top)}\\ &= 1-2p\end{aligned}$$

Thus, the angle Ψ between θ_f and $\theta_f \odot M(p)$ can be tuned by the choice of p through:

$$p(\Psi) = \frac{1}{2}(1-\langle\cos(\Psi)\rangle) \tag{3.32}$$

Hyperparameter search for EP. We distinguish between two kinds of hyperparameters: the recurrent hyperparameters - i.e. T, K and β - and the learning rates. A first guess of the recurrent hyperparameters T and β is found by plotting the Δ^{EP} and ∇^{BPTT} processes associated to synapses and neurons to see qualitatively whether the theorem is approximately satisfied, and by conjointly computing the proportions of synapses whose Δ^{EP}_W processes have the same sign as its ∇^{BPTT}_W processes. K can also be found out of the plots as the number of steps which are required for the gradients to converge. Morever, plotting these processes reveal that gradients are vanishing when going away from the output layer, i.e. they lose up to 10^{-1} in magnitude when going from a layer to the previous (i.e. upstream) layer. We subsequently initialized the learning rates with increasing values going from the output layer to upstreams layers. The typical range of learning rates is $[10^{-3}, 10^{-1}]$, $[10, 1000]$ for T, $[2, 100]$ for K and $[0.01, 1]$ for β. Hyperparameters where adjusted until having a train error the closest to zero. Finally, in order to obtain minimal recurrent hyperparameters - i.e. smallest T and K possible - we progressively decreased T and K until the train error increases again.

Table 3.2: Table of hyperparameters used for training. "C" and "VF" respectively denote "continual" and "vector-field", "-#h" stands for the number of hidden layers. The sigmoid activation is defined by Eq. (3.30).

	Activation	T	K	β	Random β	Epochs	Learning rates
EP-1h	sigmoid	30	10	0.1	False	30	$0.08 - 0.04$
EP-2h	sigmoid	100	20	0.5	False	50	$0.2 - 0.05 - 0.005$
C-EP-1h	sigmoid	40	15	0.2	False	100	$0.0056 - 0.0028$
C-EP-1h	sigmoid	40	15	0.2	True	100	$0.0056 - 0.0028$
C-EP-2h	sigmoid	100	20	0.5	False	150	$0.01 - 0.0018 - 0.00018$
C-VF-1h	sigmoid	40	15	0.2	True	100	$0.0076 - 0.0038$
C-VF-2h	sigmoid	100	20	0.35	True	150	$0.009 - 0.0016 - 0.00016$

Full table of results. Since Table 4.1 does not show C-VF simulation results for all initial weight angles, we provide below the full table of results, including those which were used to

plot Fig. 4.1.

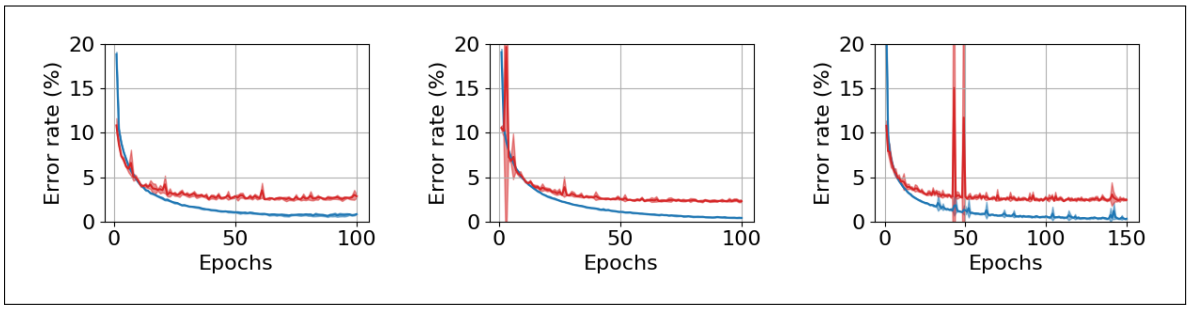

Figure 3.1: Train and test error achieved on MNIST with Continual Equilibrium Propagation (C-EP) on the Discrete-Time RNN model with symmetric weights. Plain lines indicate mean, shaded zones delimiting mean plus/minus standard deviation over 5 trials. **Left**: C-EP on the fully connected layered architecture with one hidden layer (784-512-10) without beta randomization. **Middle**: C-EP on the fully connected layered architecture with one hidden layer (784-512-10) with beta randomization. **Right**: C-EP on the fully connected layered architecture with two hidden layers (784-512-512-10) without beta randomization.

Table 3.3: Training results on MNIST with EP, C-EP and C-VF. "#h" stands for the number of hidden layers. We indicate over five trials the mean and standard deviation for the test error, the mean error in parenthesis for the train error. T (resp. K) is the number of iterations in the first (resp. second) phase.

	Initial $\Psi(\theta_\mathrm{f}, \theta_\mathrm{b})$ (°)	Error (%)		T	K	Random β	Epochs
		Test	Train				
EP-1h	–	2.00 ± 0.13	(0.20)	30	10	No	30
EP-2h	–	1.95 ± 0.10	(0.14)	100	20	No	50
C-EP-1h	–	2.85 ± 0.18	(0.83)	40	15	No	100
C-EP-1h	–	2.28 ± 0.16	(0.41)	40	15	Yes	100
C-EP-2h	–	2.44 ± 0.14	(0.31)	100	20	No	150
C-VF-1h	0	2.43 ± 0.08	(0.77)	40	15	Yes	100
	22.5	2.38 ± 0.15	(0.74)	40	15	Yes	100
	45	2.37 ± 0.06	(0.78)	40	15	Yes	100
	67.5	2.48 ± 0.15	(0.81)	40	15	Yes	100
	90	2.46 ± 0.18	(0.78)	40	15	Yes	100
	112.5	4.51 ± 3.96	(2.92)	40	15	Yes	100
	135	86.61 ± 4.27	(88.51)	40	15	Yes	100
	157.5	91.08 ± 0.01	(90.98)	40	15	Yes	100
	180	92.82 ± 3.47	(92.71)	40	15	Yes	100
C-VF-2h	0	2.97 ± 0.19	(1.58)	100	20	Yes	150
	22.5	3.54 ± 0.75	(2.70)	100	20	Yes	150
	45	3.78 ± 0.78	(2.86)	100	20	Yes	150
	67.5	4.59 ± 0.92	(4.68)	100	20	Yes	150
	90	5.05 ± 1.17	(4.81)	100	20	Yes	150
	112.5	20.33 ± 13.03	(20.30)	100	20	Yes	150
	135	59.04 ± 17.97	(60.53)	100	20	Yes	150
	157.5	77.90 ± 13.49	(78.04)	100	20	Yes	150
	180	74.17 ± 12.76	(74.05)	100	20	Yes	150

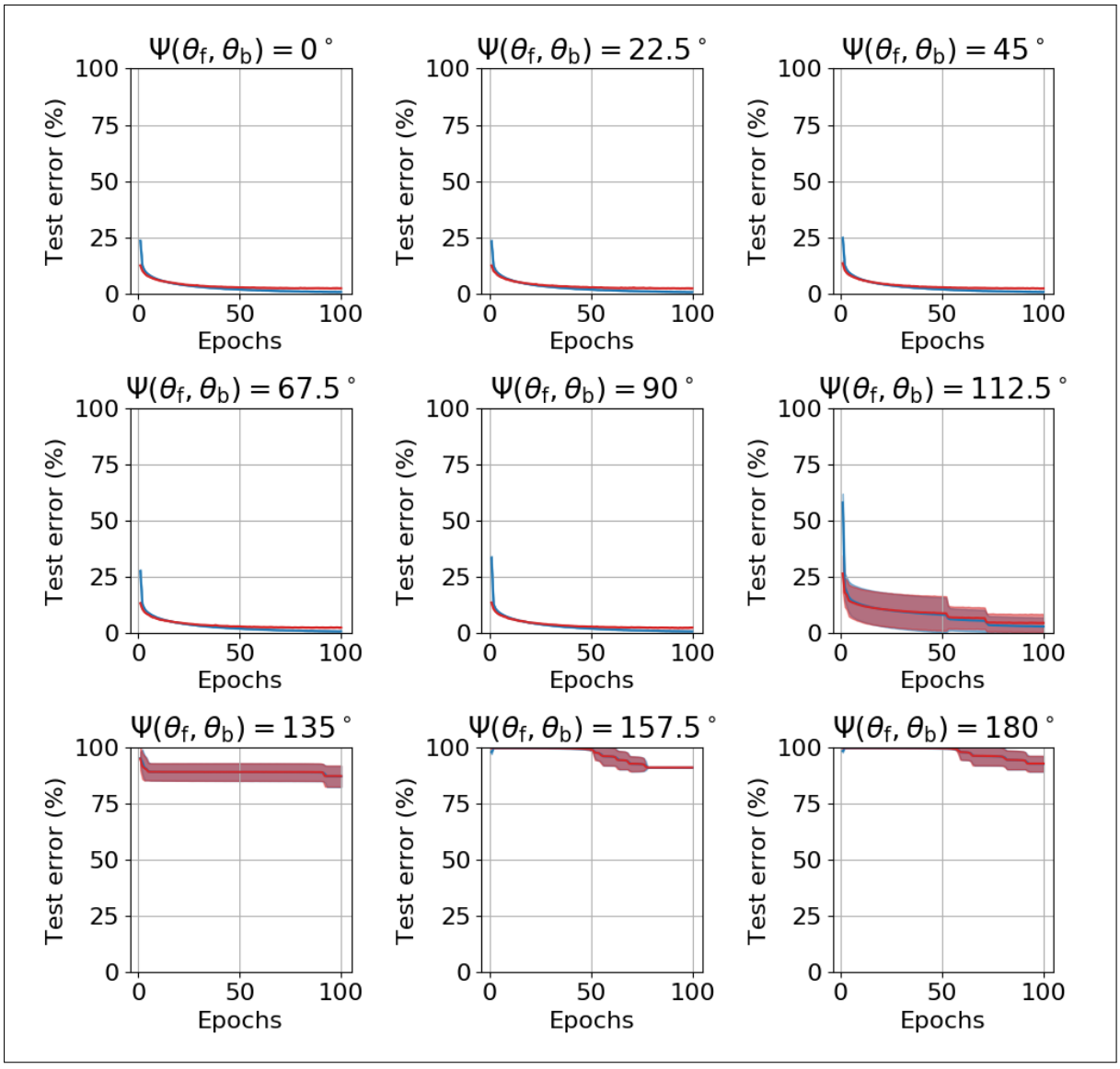

Figure 3.2: Train and test error achieved on MNIST by Continual Vector Field Equilibrium Propagation (C-VF) on the Discrete-Time RNN model with asymmetric weights with one hidden layer (784-512-10) for different initialization for the angle between forward and backward weights (Ψ). Plain lines indicate mean, shaded zones delimiting mean plus/minus standard deviation over 5 trials.

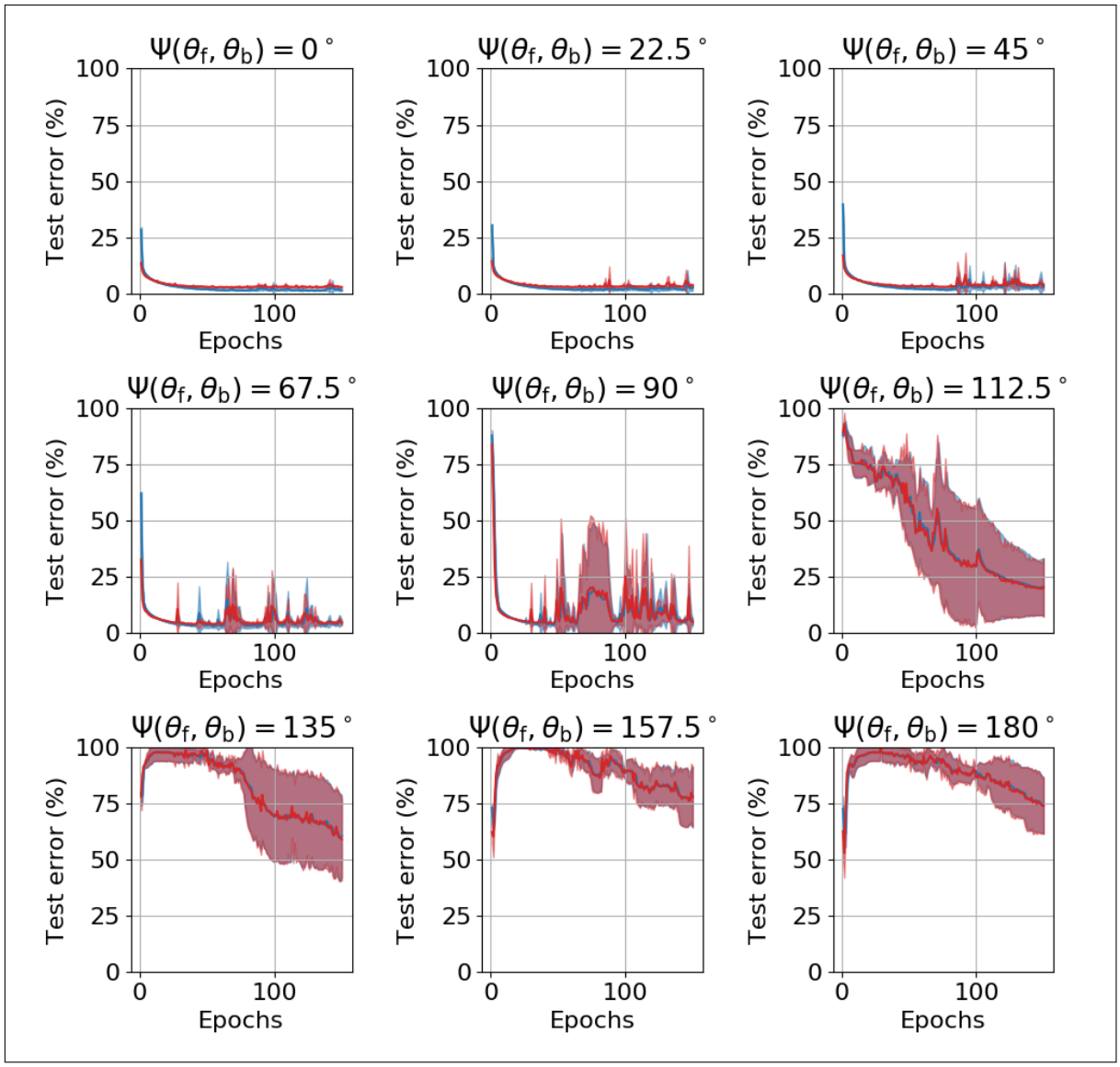

Figure 3.3: Train and test error achieved on MNIST by Continual Vector Field Equilibrium Propagation (C-VF) on the vanilla RNN model with asymmetric weights with two hidden layers (784-512-512-10) for different initialization for the angle between forward and backward weights (Ψ). Plain lines indicate mean, shaded zones delimiting mean plus/minus standard deviation over 5 trials.

Résumé court de la thèse

Le deep learning s'est imposé à l'ensemble de la société grâce à l'utilisation des GPUs (Graphical Processing Units). Aller au-delà de la capacité des GPUs pour l'entraînement des réseaux de neurones est la motivation principale de cette thèse. Une approche possible est le calcul neuromorphique consistant à repenser l'ordinateur à partir de zéro en imitant les caractéristiques du cerveau. En particulier, les memristors, qui peuvent stocker des valeurs de poids sous forme d'états de conductance, sont des candidats prometteurs pour les synapses artificielles. Une approche excitante pour réaliser des réseaux de neurones physiques utilisant memristors serait l'apprentissage sur puce : un tel dispositif pourrait réaliser à la fois l'inférence, le calcul de gradient et la mise à jour correspondante des conductances des memristors. Cependant, l'apprentissage sur puce est extrêmement difficile pour deux raisons. Tout d'abord, le calcul du gradient de la fonction objectif appelle à première vue à l'utilisation de l'algorithme de " backpropagation" , qui est intrinsèquement difficile à implémenter sur puce. Le deuxième défi de l'apprentissage sur puce est l'incrément de conductance à réaliser étant donnée une valeur de gradient : les memristors présentent de nombreuses imperfections qui entravent considérablement l'apprentissage sur puce. Dans cette thèse, nous proposons de démêler ces deux aspects de l'apprentissage sur puce. D'une part, nous étudions l'effet des imperfections des memristors sur l'apprentissage des machines Boltzmann restreintes et proposons des stratégies de programmation appropriées. D'autre part, nous nous appuyons sur l'algorithme de " Equilibrium Propagation" , un équivalent de la backpropagation dont la règle d'apprentissage, calculée par la physique du système lui-même, est spatialement locale et mathématiquement fondée.

Thesis Abstract

The deep learning approach to AI has taken upon the whole society thanks to the use of Graphical Computing Units (GPUs). Going beyond the capability of the GPUs for deep neural network training is the core motivation of this thesis. One possible approach is neuromorphic computing, which consists in rethinking the computer from scratch by mimicking brain features. In particular memristors, which can store weight values as conductance states, are promising artificial synapse candidates. An appealing approach to train memristor-based hardware neural networks would be on-chip learning: the chip could sustain inference, gradient computation and subsequent conductance update altogether. However, on-chip learning is extremely challenging for two reasons. First, the computation of the objective function gradient calls at first sight for backpropagation, which is hardware unfriendly. More hardware convenient approaches use learning heuristics that poorly scale to deeper architectures, probably because of their lack of theoretical guarantees.The second challenge of on-chip learning is the conductance update to be performed given a gradient value: memristors exhibit many imperfections which dramatically hamper on-chip learning. In this thesis, we propose to disentangle these two aspects of on-chip learning. On the one hand, we study the effect of memristive device imperfections on the training of Restricted Boltzmann Machines and propose appropriate programming strategies. On the other hand, we build upon Equilibrium Propagation, a hardware friendly counterpart of backpropagation whose learning rule, computed by the physics of the system itself, is spatially local and mathematically grounded.