Réseaux de neurones à l'échelle nano. Quels modèles d'apprentissage ?

Erwann Martin

université
PARIS-SACLAY

Réseaux de neurones à l'échelle nano. Quels modèles d'apprentissage ?

How can artificial neural networks composed of nanodevices learn ?

Thèse de doctorat de l'université Paris-Saclay

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

Thèse préparée dans l'unité de recherche UMPhy (Université Paris-Saclay, CNRS, Thales), sous la direction de Julie GROLLIER, directrice de recherche, le co-encadrement de Teodora PETRISOR, Dr., Ingénieure-chercheur (Thales Recherche et Technologies).

Thèse soutenue à Paris-Saclay, le 4 juillet 2022, par

Erwann MARTIN

Composition du jury

Gilles SASSATELLI Directeur de recherche, LIRMM	Président
Vincent GRIPON Directeur de recherche, IMT Atlantique	Rapporteur & examinateur
Timothée MASQUELIER Directeur de recherche, Cerco	Rapporteur & examinateur
Julie GROLLIER Directrice de recherche, UMPhy CNRS/Thales	Directrice de thèse

THESE DE DOCTORAT
NNT : 2022UPASP064

université PARIS-SACLAY | ÉCOLE DOCTORALE Physique en Île-de-France (PIF)

Titre : Réseaux de neurones à l'échelle nano. Quels modèles d'apprentissage ?

Mots clés : Apprentissage automatique, Memristor, Réseaux de neurones artificiel, Réseaux de neurones impulsionnel

Résumé : La performance de généralisation des réseaux de neurones profonds vient de leur capacité d'apprentissage qui requiert des ressources de calcul importantes et est extrêmement gourmand en énergie, dépassant de loin la consommation du cerveau, dont ces modèles sont pourtant inspirés.

Les architectures neuromorphiques se rapprochent du fonctionnement du cerveau en connectant au plus près des synapses et des neurones physiques. L'utilisation de nano-technologies émergentes pour ces composants est très prometteuse pour gagner en densité et en énergie.

L'entraînement des réseaux de neurones est, le plus souvent, effectué sur un circuit externe au réseau. Cette externalisation des calculs augmente considérablement le coût énergétique et la surface comparé au réseau seul. Avoir un apprentissage à l'état de l'art sans ce circuit additionnel nécessite de développer de nouveaux algorithmes.

Le but de la thèse a été l'élaboration de nouveaux algorithmes dédiés aux puces neuromorphiques, conçus pour un apprentissage intrinsèque, c'est-à-dire sans circuit additionnel, où les signaux des neurones calculent et appliquent localement le changement des poids synaptiques.

Nous avons développé *EqSpike*, un algorithme pour les neurones impulsionnels inspiré d'*Equilibrium Propagation*, avec un apprentissage intrinsèque qui permet d'obtenir des résultats à l'état de l'art sur des problèmes de classification d'images. Nous avons montré qu'*EqSpike* est compatible avec l'utilisation de nano-composants, en particulier de memristors comme synapses. Nous avons en particulier montré la robustesse de cet algorithme à des variances inter-composant réalistes. Nous avons de plus proposé deux variantes d' *EqSpike* avec un traitement des impulsions différent pour appliquer la loi d'apprentissage et répondant à différentes contraintes matérielles.

Les algorithmes développés dans la thèse permettraient une économie d'énergie de plusieurs ordres de grandeur comparé aux architectures *Von Neumann* et d'embarquer l'apprentissage, une fois déployés sur des puces neuromorphiques.

Title : How can artificial neural networks composed of nanodevices learn ?

Keywords : Machine learning, Memristor, Artificial Neural Network, Spiking Neural Network

Abstract : The generalization performance of deep neural networks comes from their ability to learn, which requires significant computational resources and is extremely energy-intensive, far exceeding the consumption of the brain, from which these models are nevertheless inspired.

Neuromorphic architectures approximate the functioning of the brain by connecting as closely as possible to synapses and physical neurons. The use of emerging nano-technologies for these components is very promising to gain density and energy.

The training of neural networks is, most often, performed on a circuit external to the network. This externalization of the computations considerably increases the energy cost and the surface area compared to the network alone. To have a state of the art learning without this additional circuit requires the development of new algorithms.

The goal of the thesis was the development of new algorithms dedicated to neuromorphic chips, designed for intrinsic learning, i.e. without additional circuitry, where the signals from the neurons compute and apply locally the change of the synaptic weights.

We have developed an algorithm for spiking neurons inspired by Equilibrium Propagation, with intrinsic learning that achieves state-of-the-art results on image classification problems. We have shown that EqSpike is compatible with the use of nano-components, in particular memristors as synapses. In particular, we have shown the robustness of this algorithm to realistic inter-component variances. We have also proposed two variants of *EqSpike* with different pulse processing to apply the learning law and responding to different hardware constraints.

The algorithms developed in the thesis would allow a power saving of several orders of magnitude compared to *Von Neumann* architectures and to embed the learning, once deployed on neuromorphic chips.

Remerciements

J'aimerais tout d'abord remercier Julie Grollier et Teodora Petrisor pour avoir encadré ma thèse. Leur gentillesse, leur expertise sur ce sujet pluri-disciplinaire et surtout leur patience. J'ai énormément appris auprès d'elles.

Merci aux rapporteurs, Vincent Gripon et Timothée Masquelier, pour leurs commentaires précieux et à tout les autres membres du jury, Gilles Sassatelli et Damien Querlioz, pour les discussions enrichissante lors de la soutenance.

Je remercie les relecteurs du manuscrit, Jean-Claude, Roman, Pierre-Louis, Kelly, Lola.

Merci aux collègues qui m'ont aidé pendant cette thèse, en particulier à l'équipe des thésards LRASC/LDO.

Merci à mes amis, mes parents et mon chat qui m'ont soutenu pendant la thèse et le Covid.

Résumé

La performance de généralisation des réseaux de neurones profonds vient de leur capacité d'apprentissage qui requiert des ressources de calcul importantes et est extrêmement gourmand en énergie, dépassant de loin la consommation du cerveau, dont ces modèles sont pourtant inspirés.

Les architectures neuromorphiques se rapprochent du fonctionnement du cerveau en connectant au plus près des synapses et des neurones physiques. L'utilisation de nano-technologies émergentes pour ces composants est très prometteuse pour gagner en densité et en énergie.

L'entraînement des réseaux de neurones est, le plus souvent, effectué sur un circuit externe au réseau. Cette externalisation des calculs augmente considérablement le coût énergétique et la surface comparé au réseau seul. Avoir un apprentissage à l'état de l'art sans ce circuit additionnel nécessite de développer de nouveaux algorithmes.

Le but de la thèse a été l'élaboration de nouveaux algorithmes dédiés aux puces neuromorphiques, conçus pour un apprentissage intrinsèque, c'est-à-dire sans circuit additionnel, où les signaux des neurones calculent et appliquent localement le changement des poids synaptiques.

Nous avons développé *EqSpike*, un algorithme pour les neurones impulsionnels inspiré d'*Equilibrium Propagation*, avec un apprentissage intrinsèque qui permet d'obtenir des résultats à l'état de l'art sur des problèmes de classification d'images. Nous avons montré qu'*EqSpike* est compatible avec l'utilisation de nano-composants, en particulier de memristors comme synapses. Nous avons en particulier montré la robustesse de cet algorithme à des variances inter-composant réalistes. Nous avons de plus proposé deux variantes d' *EqSpike* avec un traitement des impulsions différent pour appliquer la loi d'apprentissage et répondant à différentes contraintes matérielles.

Les algorithmes développés dans la thèse permettraient une économie d'énergie de plusieurs ordres de grandeur comparé aux architectures *Von Neumann* et d'embarquer l'apprentissage, une fois déployés sur des puces neuromorphiques.

Abstract

The generalization performance of deep neural networks comes from their ability to learn, which requires significant computational resources and is extremely energy-intensive, far exceeding the consumption of the brain, from which these models are nevertheless inspired.

Neuromorphic architectures approximate the functioning of the brain by connecting as closely as possible to synapses and physical neurons. The use of emerging nano-technologies for these components is very promising to gain density and energy.

The training of neural networks is, most often, performed on a circuit external to the network. This externalization of the computations considerably increases the energy cost and the surface area compared to the network alone. To have a state of the art learning without this additional circuit requires the development of new algorithms.

The goal of the thesis was the development of new algorithms dedicated to neuromorphic chips, designed for intrinsic learning, i.e. without additional circuitry, where the signals from the neurons compute and apply locally the change of the synaptic weights.

We have developed an algorithm for spiking neurons inspired by Equilibrium Propagation, with intrinsic learning that achieves state-of-the-art results on image classification problems. We have shown that EqSpike is compatible with the use of nano-components, in particular memristors as synapses. In particular, we have shown the robustness of this algorithm to realistic inter-component variances. We have also proposed two variants of *EqSpike* with different pulse processing to apply the learning law and responding to different hardware constraints.

The algorithms developed in the thesis would allow a power saving of several orders of magnitude compared to *Von Neumann* architectures and to embed the learning, once deployed on neuromorphic chips.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

L'accumulation massive de données des dix dernières années a permis la création de jeux de données atteignant des Téraoctets (To), permettant aux algorithmes statistiques de les exploiter, le tout couplé avec la puissance de calcul croissante permettant de traiter ces données de plus en plus vite, en particulier avec l'essor des cartes graphiques (GPU). L'apprentissage automatique, très gourmand en données et en puissance de calcul, a permis de dépasser les performances de l'état de l'art dans beaucoup de domaines en extrayant de la connaissance de ces jeux de données, pour atteindre des performances proche d'un expert. La première grande réussite est historiquement la reconnaissance d'images avant de s'étendre dans plusieurs domaines complexes, impossibles à traiter avec les algorithmes classiques, par exemple de la traduction [Auli et al., 2013], la description de scène [Zhou et al., 2014], le jeu de Go (AlphaGo) [Silver et al., 2016] ou de la conduite automatique [Kocić et al., 2019] (détails section 1.1).

Les réseaux de neurones artificiels sont un type d'algorithmes d'apprentissage automatique qui atteignent depuis plusieurs années des performances dépassant de loin celles des autres algorithmes d'apprentissage automatique pour certains types de problèmes, comme ceux cités précédemment. Les réseaux de neurones artificiels sont inspirés du cerveau qui possède des unités de calculs (neurones) reliées par des synapses. Le cerveau humain possède environ 100 milliards de neurones et 10^{15} synapses pour une consommation d'environ 20 watts. Le cerveau consomme plusieurs ordres de grandeur de moins que les architectures matérielles utilisées pour implémenter des réseaux de neurones artificiels qui peuvent consommer plusieurs centaines voir des dizaines de milliers de watts [Strubell et al., 2019].

Cependant, pour obtenir ces excellentes performances, un réseau de neurones a besoin d'être sur-paramétré avec des millions ou des milliards de paramètres en fonction de l'application [Zhou et al., 2020]. L'entraînement de ces paramètres nécessite en plus de présenter aux neurones des dizaines de fois le jeu données d'entraînement, composé de plusieurs millions d'exemples pour les plus gros. Ce processus itératif est extrêmement gourmand en puissance de calcul. Les performances des réseaux de neurones artificiels vont donc de pair avec une capacité de calcul des ordinateurs croissante qui suit la loi de Moore [Moore, 1998]. Malheureusement, cette capacité de calcul crée aussi une consommation d'énergie croissante. Pour calculer la sortie d'un réseau de neurones, il faut en grande partie effectuer des produits matriciels massivement parallélisables. Les supercalculateurs et le cloud permettent de rassembler des milliers de processeurs de calcul en parallèle, de cartes graphiques ou d'unités spécialisées pour le calcul matriciel (TPU) [Jouppi et al., 2017]. La majeure partie de la consommation d'énergie des réseaux de neurones artificiels est due au transfert de données (poids synaptique, signaux des neurones ou données d'entraînement) entre la mémoire et les processeurs de calculs [Sze et al., 2017], pour des architectures appelées *Von Neumann*. Ces architectures consommant bien plus d'énergie que le cerveau, il est raisonnable d'estimer qu'une réduction d'énergie de plusieurs ordres de grandeur est possible pour amener à une consommation (par données traitées) plus proche du cerveau [Sze et al., 2017].

Avec les enjeux environnementaux actuels, réduire l'énergie utilisée par les réseaux de neurones et donc les émissions de CO_2 associées est important. De plus, cette thèse est une thèse *CIFRE*, effectuée conjointement avec le *CNRS* et *Thales*. *Thales* produit des équipements et capteurs embarqués ayant des fortes contraintes en énergie, puissance de calcul ou temps d'exécution, pour exécuter des tâches telles que des analyses d'environnements complexes, réalisables aujourd'hui efficacement par des réseaux de neurones.

Cependant, la forte consommation des réseaux de neurones réduit la possibilité d'utilisation au plus proche de l'équipement et peut nécessiter un envoi de données sur un *cloud* pour être traitées. Cette solution est gourmande en énergie et induit des problèmes de sécurité. Réduire la consommation d'énergie des réseaux de neurones, et en particulier de leur entraînement, permettrait par conséquent leur introduction dans des applications industrielles nécessitant de s'adapter à des environnements changeants.

Les réseaux de neurones fonctionnent en deux phases : la phase dite d'*entraînement* (ou d'*apprentissage*) sur un jeu de données représentatif du problème et une phase dite d'*inférence* dans laquelle les paramètres appris sont appliqués à des nouvelles données. L'inférence peut être optimisée sur des accélérateurs dédiés pour avoir une faible consommation d'énergie, mais pour entraîner un réseau de neurones avec des performances à l'état de l'art, des circuits supplémentaires au réseau, qui sont coûteux en espace et en énergie, sont souvent nécessaires. En effet, la rétro propagation du gradient d'erreur (l'état de l'art, expliqué en section 1.2.1) nécessite un circuit externe au réseau pour calculer, propager et garder en mémoire les gradients, et un autre circuit pour les appliquer aux paramètres du réseau (les synapses) ce qui consomme de l'énergie et de l'espace supplémentaire.

Pour réduire l'énergie dépensée pendant l'entraînement d'un réseau de neurones, les principales approches sont :

— d'améliorer les algorithmes d'apprentissage et la topologie de réseaux pour réduire le nombre de paramètres à apprendre,
— de spécialiser l'architecture matérielle pour la rendre massivement parallèle et plus adaptée aux calculs matriciels très présents dans les réseaux de neurones,
— d'optimiser conjointement les architectures matérielles et les algorithmes d'apprentissage.

Dans le cerveau, les neurones communiquent grâce à des impulsions et ces impulsions modifient les synapses localement en fonction de différentes règles (voir section 1.3.1). Plusieurs modèles de neurones à impulsions ont été théorisés pour les réseaux de neurones artificiels (voir section 1.3). De plus, l'aspect événementiel des réseaux de neurones à impulsions est prometteur pour la réduction d'énergie comparé aux neurones formels qui envoient un signal tant que leur sortie est différente de 0 [Davidson and Furber, 2021]. En revanche, pour obtenir des performances à l'état de l'art, ils sont souvent entraînés avec une approximation de la rétro propagation du gradient d'erreur, qui nécessite des circuits externes au réseau.

Pour éviter la rétro propagation du gradient d'erreur, il est possible d'utiliser la dynamique du réseau pour calculer un gradient d'erreur. En effet, les réseaux de neurones impulsionnels peuvent calculer un gradient d'erreur localement pour se passer d'un circuit externe en utilisant la dynamique des neurones connectés à la synapse, en particulier leurs moments d'émission d'impulsions. Cependant, les performances ne sont pas à l'état de l'art, notamment pour les réseaux dit profonds. Afin d'améliorer les performances, il est possible de rajouter un signal de récompense/pénalisation, mais ce signal n'est pas local à la synapse ce qui rajoute de la complexité pour une implémentation physique, et donc de la consommation d'énergie et de l'espace utilisé.

Des algorithmes pour des réseaux de neurones formels utilisent la dynamique du réseau pour de l'apprentissage supervisé en calculant implicitement le gradient de la rétro propagation d'erreur en minimisant l'énergie du système (voir section 1.5.4). Il est prometteur de les adapter pour des réseaux impulsionnels afin de profiter de la dynamique interne des neurones à impulsions et de leur faible coût énergétique tout en ayant un apprentissage local avec un gradient d'erreur qui est une approximation de la rétro propagation du gradient. Dans cette thèse, nous nous baserons sur la règle d'apprentissage de l'algorithme *Equilibrium propagation* [Scellier and Bengio, 2016] pour calculer le gradient d'erreur localement grâce à la dynamique du réseau.

Adapter le matériel aux algorithmes d'apprentissage permet de minimiser l'énergie globale dépensée. Actuellement, la consommation d'énergie des réseaux de neurones est en grande partie due au transfert de données sur l'architecture des GPU/CPU dite *Von Neumann* qui sépare la mémoire des unités de calculs. Pour aller plus loin dans l'économie d'énergie, il est possible de rapprocher la mémoire des unités de calcul pour avoir une architecture plus distribuée que les *GPU*s ou *TPU*, dite *Beyond Von Neumann*. Cette approche est expliquée dans la section 1.4.

Les approches précédentes restent loin de l'architecture physique du cerveau où neurones et synapses sont fortement interconnectés et ont à la fois une fonctionnalité de mémoire et d'unité de calcul, ce qui semble clé pour l'efficacité énergétique. Reproduire cette fonctionnalité avec des nanocomposants nouveaux,

compatibles avec les technologies de fabrication CMOS (*Complementary Metal Oxide Semiconductor*, en anglais), permettrait d'avancer vers une réduction d'énergie plus significative. Cette approche est expliquée dans la section 1.4.1. De plus, certains de ces composants, en tant que synapses, pourraient être modifiés localement grâce aux impulsions des neurones connectés.

Parmi les technologies émergentes prometteuses pour les réseaux de neurones, tels que les memristors, les oscillateurs spintroniques ou les composants optroniques, nous avons identifié les memristors. Ces nano-composants ont l'avantage d'être non volatiles et de pouvoir émuler le comportement d'une synapse avec une faible consommation d'énergie et d'espace dans un réseau de neurones. Cette thèse s'inscrit dans la recherche et le codéveloppement d'algorithmes d'apprentissage optimisés pour des puces neuromorphiques avec des memristors comme synapses. Le but est de développer des algorithmes qui exploitent le comportement des réseaux à impulsions pour calculer localement la loi d'apprentissage et modifier la synapse constituée de memristors de façon intrinsèque grâce aux propriétés des memristors et aux impulsions des neurones. La thèse est un codéveloppement entre la physique et l'algorithmique. Des modèles de comportement des memristors ont été pris en compte dans les développements algorithmiques, et une recherche sur la robustesse aux variations inter-composant a été effectuée.

Les travaux effectués durant cette thèse sont :
— Développement d'un algorithme, *EqSpike*, capable d'un apprentissage local et intrinsèque qui peut s'adapter à l'architecture matérielle (une publication dans le journal iScience [Martin et al., 2021])
— Adaptation de *EqSpike* pour un circuit avec des memristor comme synapses (un brevet déposé [*FR2101311 + PCT/EP2022/053026*] et article en préparation)
— Implémentation d'une variation de *EqSpike* utilisant l'accumulation des impulsions des neurones pour calculer la règle d'apprentissage

1.1 Apprentissage automatique

L'apprentissage automatique (*Machine Learning - ML, en anglais*) est un sous domaine de l'Intelligence Artificielle qui consiste à développer des algorithmes statistiques dont les paramètres θ sont souvent modifiés itérativement à partir d'un jeu de données d'entraînement, dans le but de résoudre une tâche spécifique. Les données d'entraînement doivent être représentatives de la tâche à résoudre, et les exemples ne sont généralement pas exhaustifs. Dans le cas de l'*apprentissage supervisé*, les données d'entrée $x \in X$ (espace vectoriel) ont également un ensemble $y \in Y$ d'*étiquettes* associées à chaque entrée, représentant le résultat désiré. Dans un cas d'*apprentissage non supervisé*, seules les données d'entrée x sont disponibles. Par exemple, dans une tâche de reconnaissance d'image, l'ensemble d'entraînement, X, contient des exemples de tous les objets à reconnaître, où x peut représenter des images de chien ou de chat, y l'animal présent sur la photo et la tâche est de reconnaître quel animal est présent sur l'image.

Pour ce faire, il s'agit de définir un modèle, $f_\theta : X \to Y$, qui est une fonction mathématique généralement sur-paramétrée avec des paramètres θ, et d'y associer un algorithme d'apprentissage, qui va chercher à optimiser ces paramètres pour résoudre la tâche donnée, avec θ^* les paramètres optimaux.

Dans le cas d'un apprentissage supervisé, l'algorithme cherche à minimiser une fonction dite de coût, $L(\hat{y}, y)$, qui prend comme paramètre la sortie du modèle, $\hat{y} = f_\theta(x)$, et l'étiquette associée à l'exemple, y. Par exemple, une fonction de coût courante est l'erreur quadratique moyenne, définie par :

$$L(\hat{y}, y) = \frac{1}{M} \sum_{k=0}^{M} (\hat{y_k} - y_k)^2 \tag{1.1}$$

qu'il s'agit de minimiser, avec M la taille du vecteur de sorties du modèle.

Les capacités de *généraliser* un modèle à partir d'exemples rendent le *ML* plus performant que des approches non statistiques pour des tâches où les exemples du jeu de données ne peuvent pas être exhaustifs et où la *généralisation* permet d'être performant sur des exemples qui n'ont pas déjà été vus par le modèle. Par exemple, dans la Figure 1.1 centrale, l'ensemble de points correspond aux données d'entrée représentées par leurs coordonnées, et le but est d'apprendre la régression de la fonction qui peut générer ces points. La courbe en pointillée est la courbe générée par le modèle une fois appris.

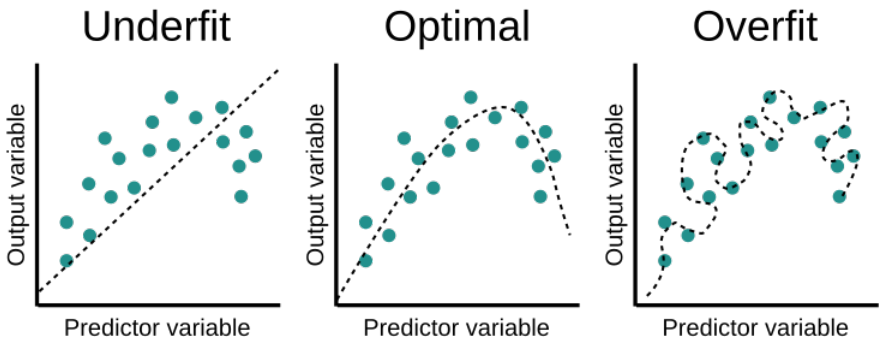


Figure 1.1 – Illustration d'un sous apprentissage à gauche, d'un apprentissage optimal au centre et d'un sur-apprentissage à droite. Extrait de https://www.fastaireference.com/overfitting.

Si le modèle arrive à associer l'étiquette et les exemples du jeu de données, mais n'arrive pas à généraliser sur de nouvelles données, le modèle *sur-apprend*. Le sur-apprentissage peut arriver si le modèle possède trop de paramètres qui vont apprendre parfaitement le jeu de données comme s'il était exhaustif et non bruité au détriment de la généralisation, comme sur la Figure 1.1 à droite.

Au contraire, si le modèle est trop simple et n'a pas assez de paramètres, alors le modèle ne pourra pas généraliser à de nouveaux exemples, et on dit que le modèle *sous-apprend*. Ce cas est illustré dans la Figure 1.1 à gauche, où une régression linéaire ne suffit pas pour représenter ce jeu de données.

Pour réduire l'erreur globale du modèle, un algorithme itératif souvent utilisé est la descente de gradient [Robbins and Monro, 1951]. Il consiste à calculer le gradient de l'ensemble des données par rapport aux paramètres du modèle, modèle qui est représenté par la fonction $f_\theta(x)$, puis de soustraire ce gradient des paramètres, pondéré par un taux d'apprentissage η, comme suit :

$$\theta_{t+1} = \theta_t - \eta \frac{\partial f_{\theta_t}(x)}{\partial \theta_t} \tag{1.2}$$

avec t le nombre d'itérations.

Un des algorithmes de descente de gradient le plus utilisé est la descente de gradient stochastique (*SGD - Stochastic gradient descent*, en anglais) où les données sont présentées au réseau, une à une, de façon aléatoire. Plusieurs variantes de *SGD* existent pour essayer d'obtenir une convergence du modèle plus rapide sur certains types de données. Par exemple, il est possible de rajouter une inertie au gradient [Rumelhart et al., 1986], de calculer le gradient moyen sur un sous-ensemble de X [Polyak and Juditsky, 1992] ou d'utiliser diverses régularisations [Hinton et al., 2012b] pour s'adapter au problème donné ou aux caractéristiques du modèle. Parmi les méthodes les plus répandues, nous citerons *AdaGrad* [Duchi et al., 2011], *ADAM* [Kingma and Ba, 2017] ou *RMSProp* [Hinton et al., 2012a] que nous détaillons ci-après.

AdaGrad [Duchi et al., 2011] (pour *Adaptive Gradient*, en anglais) est une variante de *SGD* qui adapte le taux d'apprentissage comme suit : le taux d'apprentissage est augmenté pour les paramètres avec une valeur de mis à jour faible et réduit pour les paramètres qui ont une valeur de mis à jour élevée. Cela permet une convergence du modèle plus rapide sur des problèmes avec peu de données. En revanche, le taux d'apprentissage global décroît à chaque itération, ce qui peut ralentir la convergence du modèle, au point de rendre infime chaque modification, ce qui va empêcher l'évolution des paramètres du modèle.

Similairement, *RMSProp* [Hinton et al., 2012a] (pour *Root Mean Square Propagation*, en anglais) adapte son taux d'apprentissage grâce à un paramètre v en utilisant la moyenne des carrés des gradients pour normaliser les gradients :

$$v(\theta, t) = \gamma v(\theta, t-1) + (1-\gamma)(\frac{\partial f_{\theta_t}(x)}{\partial \theta_t})^2$$

avec γ un facteur d'oubli. La mise à jour des paramètres devient :

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v(\theta, t)}} \frac{\partial f_{\theta_t}(x))}{\partial \theta_t}$$

ADAM [Kingma and Ba, 2017] est un mélange entre *AdaGrad* et *RMSProp*. Il utilise des hyper-paramètres (c'est-à-dire non appris) β_1, β_2,et utilise la moyenne des gradients m et la moyenne des carrés des gradients, v :

$$m(\theta_t, t) = \beta_1 m(\theta_t, t-1) + (1-\beta_1)(\frac{\partial f_{\theta_t}(x)}{\partial \theta_t})$$
$$v(\theta_t, t) = \beta_2 v(\theta_t, t-1) + (1-\beta_2)(\frac{\partial f_{\theta_t}(x)}{\partial \theta_t})^2$$

puis *ADAM* calcule deux moments pour adapter le taux d'apprentissage, $\hat{m_\theta}$ et $\hat{v_\theta}$, tel que :

$$\hat{m_\theta} = \frac{m(\theta, t)}{(\beta_1)^t}$$
$$\hat{v_\theta} = \frac{v(\theta, t)}{(\beta_2)^t}$$

La mise à jour des paramètres au temps t+1 devient :

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m_\theta}}{\sqrt{\hat{v}(\theta, t)} + \epsilon}$$

ADAM est un algorithme qui fonctionne bien pour la plupart des problèmes rencontrés dans le domaine du *ML*. Les deux variables m et v fonctionnent comme une inertie, normalisée, sur l'évolution des paramètres, ce qui permet de modifier les paramètres en fonction de leur valeur de mise à jour comme avec *AdaGrad* et de garder en mémoire les modifications des paramètres pour avoir une modification des paramètres plus lisse comme dans *RMSProp*.

Nous avons présenté les algorithmes de descente de gradient les plus utilisés pour entraîner des modèles de ML. Une liste complète peut être trouvée dans [Ruder, 2016].

Cette thèse se concentre sur les réseaux de neurones artificiels comme modèles à apprendre. Nous allons maintenant décrire le fonctionnement de ces réseaux de neurones artificiels.

1.2 Fonctionnement des réseaux de neurones artificiels

Le cerveau est composé de cellules appelées *neurones* connectées entre elles par des *synapses* et communiquant grâce à des signaux bioélectriques. Même si le cerveau est composé d'autres types de cellule comme les cellules *gliales*, les neurones (avec leurs synapses associées) sont théoriquement les cellules responsables de la capacité de traitement de l'information du cerveau. Les réseaux de neurones artificiels sont des modèles de machine learning inspirés de cette architecture composée de neurones et de synapses. Historiquement, le premier réseau de neurones artificiel est le *Perceptron* [Rosenblatt, 1958]. La valeur des entrées et des sorties du neurone est un nombre réel qui représente de façon abstraite les fréquences d'impulsions des neurones biologiques.

Un réseau de neurones artificiels, qui sera désormais appelé "réseau de neurones" ou simplement *ANN* (*Artificial Neural Network*, en anglais) par la suite, est représentable par un graphe orienté composé de différents nœuds de calcul appelés *neurones* reliés par des arêtes, représentant les *synapses*. Ces dernières transmettent et pondèrent l'information entre les neurones, leurs valeurs étant appelées *poids synaptiques* notés w (W sera l'ensemble des poids du réseau). Utiliser un *ANN* pour résoudre une tâche par apprentissage revient donc à trouver l'ensemble de poids synaptiques optimal W^* selon une fonction de coût prédéfinie.

Un neurone a pour entrée la somme des sorties des neurones qui lui sont connectés pondérées par les synapses et pour sortie $\hat{y_j}$, un réel qui représente la sortie d'une fonction σ, appelée *fonction d'activation*, tel que :

$$\hat{y_j} = \sigma(\sum_i w_{ij} x_i + b_j), \tag{1.3}$$

avec b_j un biais associé au neurone j et x_i la i-ème entrée du neurone. Le biais propre à chaque neurone est un paramètre qui peut être optimisé comme les w_{ij}. Il sert à ajuster l'entrée du neurone. La fonction d'activation est une fonction monotone non linéaire de préférence dérivable pour faciliter le calcul des gradients. Parmi les fonctions les plus utilisées, il y a la fonction *ReLU* (*Rectified Linear Unit*) :

$$f(x) = \begin{cases} x, & \text{si } x > 0 \\ 0, & \text{sinon} \end{cases}. \tag{1.4}$$

et la sigmoïde :

$$f(x) = \frac{1}{1 + e^x} \tag{1.5}$$

ou la fonction marche d'escalier :

$$f(x) = \begin{cases} 1, & \text{si } x > 0 \\ 0, & \text{sinon} \end{cases} \tag{1.6}$$

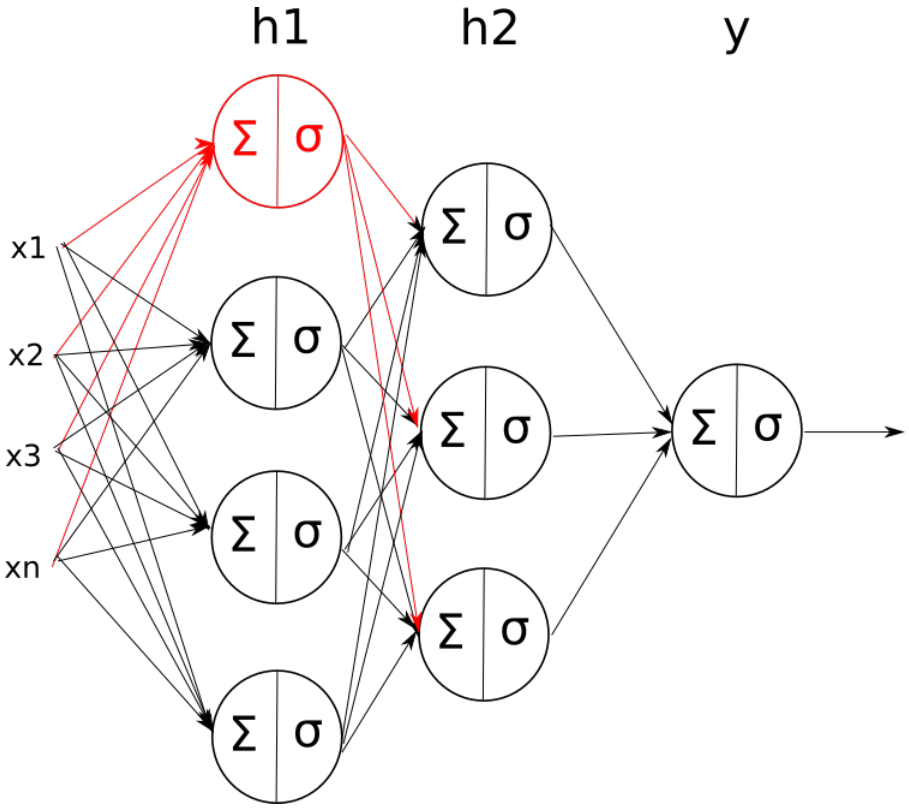


Figure 1.2 – Représentation d'un réseau de neurones avec 4 neurones d'entrée, une couche cachée de 4 neurones, une couche cachée de 3 neurones et 1 neurone de sortie, les flèches représentant les synapses. Les flèches rouges représentent les entrées et sorties d'un neurone particulier.

Un neurone avec la fonction d'activation marche d'escalier permet de séparer par un hyperplan les entrées X où les sorties du neurone correspondent aux coordonnées de l'hyperplan séparateur. L'objectif du réseau est d'apprendre une séparation optimale des données. La Figure 1.3 montre un problème avec des entrées à 2 dimensions séparables linéairement en deux classes (chaque couleur correspond à une classe). La courbe rouge est la droite séparatrice du neurone après entraînement.

Dans le cadre de l'apprentissage supervisé, un réseau de neurones est un modèle qui permet d'approximer des fonctions associant des données d'entrée X à des sorties souhaitées Y (par exemple, associer une classe à une image pour un problème de classification d'image). Pour apprendre ce problème, il faut modifier les paramètres w_{ij} de façon itérative. Les données sont présentées une à une au réseau. Après chaque itération, une erreur est calculée par exemple comme étant la distance entre l'étiquette (valeur désirée) de la donnée

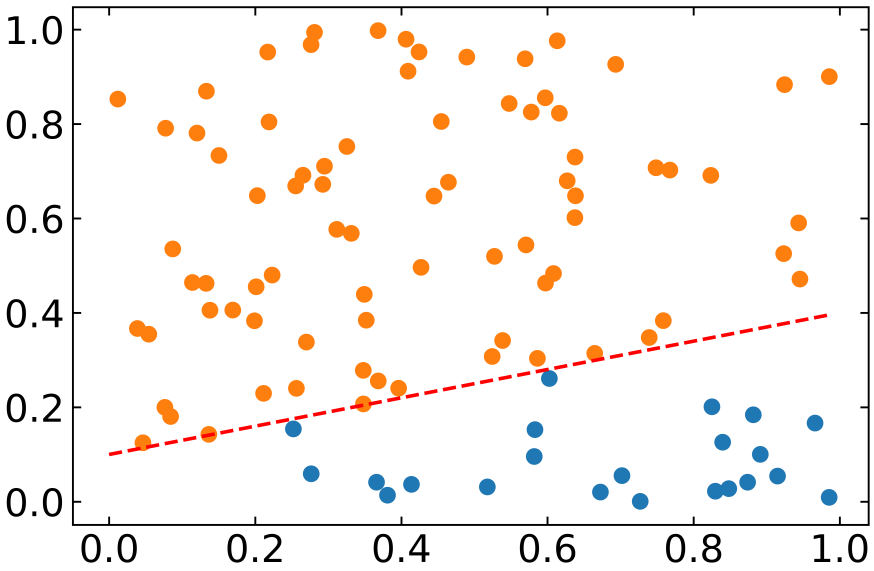


Figure 1.3 – Séparation linéaire entre la classe orange et la classe bleue par la droite en pointillés apprise avec un *Perceptron*

présentée et la sortie du neurone. La modification des poids dépend du gradient de l'erreur par rapport à l'entrée, pondéré par un taux d'apprentissage η :

$$w_{ij} \leftarrow w_{ij} - \eta(y_j - \hat{y_j})x_i \tag{1.7}$$

Avec un jeu de données séparable linéairement, les synapses d'un neurone sont garanties de converger vers un jeu de poids optimal (c'est-à-dire une erreur nulle).

Cependant, la plupart des problèmes ne sont pas linéairement séparables. Par exemple, la fonction logique *XOR* (f(A,B) = (A ET non B) OU (non A ET B)) ne peut pas être approximée par un neurone (illustré Figure 1.4). Pour résoudre ce problème, il faut alors plusieurs couches imbriquées, c'est-à-dire que la sortie d'une couche soit l'entrée de la suivante. Un *ANN* est alors construit comme une représentation hiérarchique en *couches de neurones*, telle qu'illustrée sur la Figure 1.2. Les neurones d'une même couche ne communiquent pas entre eux et reçoivent en entrée les sorties des neurones de la couche précédente (sauf pour la première couche dite couche d'entrée). Les neurones d'entrée peuvent être assimilées aux neurones sensoriels qui reçoivent les entrées de l'environnement comme dans le cortex visuel ou auditif. La couche finale est la couche qui prend la décision/ classe un objet. Les couches qui ne sont ni la couche d'entrée, ni la couche finale, sont appelées "couches cachées". Le résultat est un *Perceptron* multicouches (*Multi Layer Perceptron - MLP*, en anglais) qui permet d'avoir une frontière de séparation plus complexe qui est la composée de l'ensemble des couches, tel que :

$$f_\theta(x) = \hat{y} = f_{\theta_k}(f_{\theta_{k-1}}(...(f_{\theta_0}(x)) \tag{1.8}$$

avec k le nombre de couches et θ_{k-1} les paramètres associé à la couche k. Si le réseau de neurone a plus de 3 couches, il est alors appelé un réseau de neurone profond (*Deep Neural Network - DNN*, en anglais) [Bengio, 2009, LeCun et al., 2015].

Un *ANN* avec une couche cachée est un approximateur universel, il peut donc approximer n'importe quelle fonction arbitrairement complexe à condition qu'il possède suffisamment de neurones et qu'il possède les bons poids synaptiques [Hornik et al., 1989]. En revanche, le nombre de neurones nécessaire croit de façon exponentielle en fonction de la précision final souhaitée, ce qui est souvent inapplicable pour des problèmes réels.

Ajouter des couches cachées au réseau permet d'augmenter le niveau d'abstraction de la représentation. Le nombre de neurones théorique pour approximer un problème croit exponentiellement avec un réseau à une couche en fonction de la précision souhaitée. Augmenter le nombre de couches permet souvent de réduire le nombre de paramètres à apprendre grâce à l'exploitation des niveaux d'abstraction, pour la même précision finale.

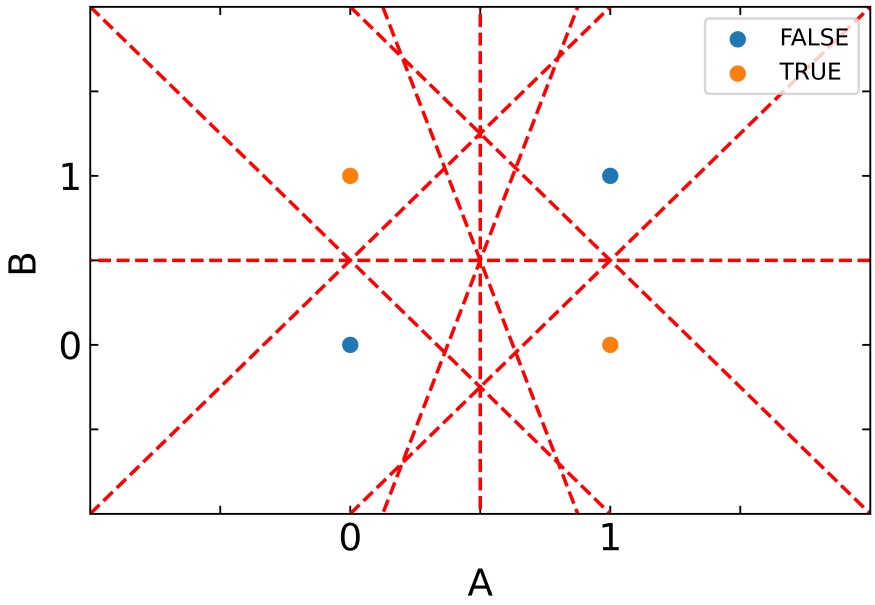


Figure 1.4 – Les points représentent les possibilités de la fonction *XOR*, les lignes en rouges correspondent à des droites séparatrices. Aucune n'arrive à séparer le problème.

1.2.1 Entraînement des *ANNs* grâce à la rétro-propagation du gradient d'erreur

Les poids des réseaux de neurones sont le plus souvent appris grâce à des algorithmes de descente de gradient, comme présenté dans la section 1.1. Pour trouver le gradient de chaque couche synaptique par rapport à la fonction de coût, l'algorithme le plus répandu est la rétro-propagation du gradient d'erreur (*backward propagation of errors - BP*) [Rumelhart et al., 1986]. Comme son nom l'indique, le gradient de l'erreur, $\frac{\partial L(\hat{y},y)}{\partial W}$, est propagé dans le réseau, de la couche de sortie vers la couche d'entrée, afin de pouvoir calculer l'actualisation des poids W de chaque couche. Ainsi le calcul est récursif, le gradient d'erreur exact de la couche $n-1$ est calculé avec le gradient d'erreur de la couche n. Par exemple, pour le réseau présenté dans la Figure 1.2, on obtient alors, pour la dérivée de l'erreur par rapport aux poids pour les synapses entre les neurones de la couche h2 et la sortie $\hat{y}$:

$$\frac{\partial L(\hat{y},y)}{\partial w_{h2,\hat{y}}} = \frac{\partial L(\hat{y},y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_{h2,\hat{y}}}$$

pour les synapses entre les neurones de la couche $h1$ et la couche $h2$:

$$\frac{\partial L(\hat{y},y)}{\partial w_{h1,h2}} = \frac{\partial L(\hat{y},y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h2} \frac{\partial h2}{\partial w_{h1,h2}}$$

Et ainsi pour les synapses entre les neurones de la couche d'entrée et la couche h1 :

$$\frac{\partial L(\hat{y},y)}{\partial w_{x1,h1}} = \frac{\partial L(\hat{y},y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h2} \frac{\partial h2}{\partial h1} \frac{\partial h1}{\partial w_{x1,h1}}$$

Grâce à la rétro propagation, il est possible de propager le gradient à travers un réseau arbitrairement profond, si toutes les fonctions d'activation des neurones sont dérivables.

1.2.2 Adapter le réseau aux types de données avec les réseaux convolutifs et récurrents

Chaque neurone d'un MLP effectue une somme pondérée de toutes ses entrées, ce qui ne permet pas de prendre en compte certaines structures où les paramètres d'entrée sont corrélés entre eux. Par exemple,

un pixel d'une image est corrélé avec les autres pixels proches dans l'espace. La structure locale des données est donc importante à prendre en compte. De même pour les données qui évoluent au cours du temps, il est souvent important de prendre en compte les données précédentes temporellement en adaptant la structure du réseau.

Pour permettre une meilleure prise en compte de ces structures de données, comparé aux neurones "classiques", il est possible d'utiliser des modèles de neurones qui se comportent différemment. Les neurones les plus courants qui diffèrent des neurones "classiques" sont les neurones récurrents pour prendre en compte la structure des données temporelles et les neurones convolutifs pour prendre en compte la structure spatiale des données.

Ces derniers sont inspirés du cortex visuel des animaux. Chaque neurone effectue une convolution sur des signaux à plusieurs dimensions pour extraire des informations en exploitant la structure spatiale des données où le noyau de convolution w (ou matrice de convolution) de taille NxN, correspond au poids des synapses :

$$f_\theta(x) = \sum_{i=1}^{N}\sum_{j=1}^{N} x_{ij} \cdot w_{ij} \tag{1.9}$$

Une fonction d'activation est appliquée après chaque convolution pour avoir la non-linéarité nécessaire. Une illustration de convolution est montrée sur la Figure 1.5.

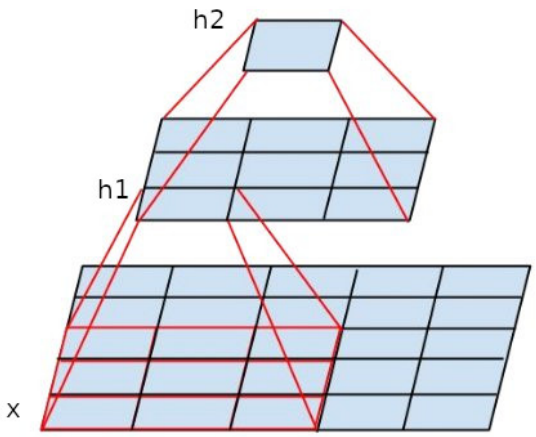


Figure 1.5 – Deux convolutions successives avec un noyau de taille 3x3 puis 5x5 où $h1_{11} = \sum_{i=1}^{3}\sum_{j=1}^{3} x_{ij} \cdot w1_{ij}$ et $h2 = \sum_{i=1}^{5}\sum_{j=1}^{5} h1_{ij} \cdot w2_{ij}$. Figure adapté de [Szegedy et al., 2015]

Une couche convolutive est généralement suivie d'une couche qui réduit la taille de la représentation obtenue, dite de *pooling* (en anglais), grâce à une opération sur les pixels d'une zone. Par exemple le *max pooling* qui sélectionne les coefficients résultants avec la valeur maximale par zone. Cette couche de *pooling* permet de réduire la taille des données tout en gardant les paramètres importants. Un réseau convolutif (*Convolution Neural Network -CNN*, en anglais) se termine généralement par plusieurs couches entièrement connectées. Les couches convolutives extraient les paramètres importants avec une corrélation spatiale et les dernières couches exploitent ces paramètres abstraits. Un exemple de *CNN* est donné dans la Figure 1.6 représentant le réseau VGG-16 [Simonyan and Zisserman, 2014]. Ce réseau est composé de 13 couches convolutives puis de 3 couches entièrement connectées. Les réseaux convolutifs sont les réseaux qui obtiennent les meilleurs résultats sur la classification d'image, comme sur un des benchmarks les plus complexes appelé *ImageNet*[1] [Russakovsky et al., 2015].

Un réseau récurrent est un réseau dans lequel un neurone, ou un bloc de neurones, reçoit, pour une partie de ses entrées, sa propre sortie (la connexion est récurrente), comme représenté sur la Figure 1.7. Les RNN permettent de garder une mémoire des informations précédentes traitées par les neurones. Le problème de ces réseaux est que la sortie des neurones, ainsi que les gradients, peut exploser (tendre vers l'infini) ou disparaître (tendre vers 0), supprimant le mécanisme de mémoire. Certains types de neurones

1. https://image-net.org/

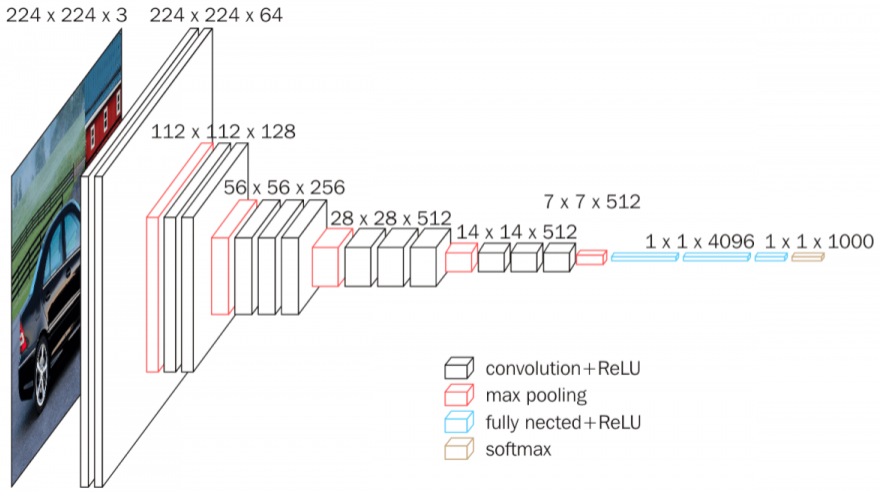


Figure 1.6 – Représentation du réseau convolutif VGG [Simonyan and Zisserman, 2014] à 16 couches. Les couches en noir sont les couches convolutives, le pooling est en rouge et les couches entièrement connectées en bleu. Figure extraite de https://neurohive.io/en/popular-networks/vgg16/.

récurrents, comme les *LSTM (Long Short-Term Memory)* [Hochreiter and Schmidhuber, 1997] ou les *GRU (Gate Recurrent Unit)* [Cho et al., 2014], diminuent ce problème. Ils évitent les problèmes de disparition de l'information temporelle grâce à une cellule qui enregistre les informations pertinentes à long terme et une cellule qui enregistre les informations pertinentes à court terme.

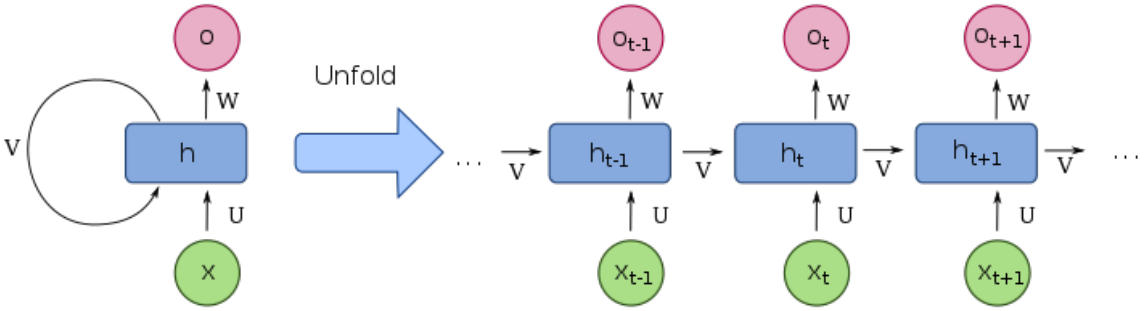


Figure 1.7 – RNN - Un neurone récurent, à droite le réseau est déplié pour montrer le calcul au cours du temps. (fdeloche,CC BY-SA 4.0, via Wikimedia Commons).

Pour les réseaux récurrents, la variante de la rétro-propagation du gradient d'erreur est *Backpropagation through time (BPTT)* [Werbos, 1990]. Le calcul du gradient est propagé pour chaque itération récursive le même nombre de fois que lors de l'exécution de l'inférence, ce qui revient à déplier le réseau de neurones dans le temps, comme montré sur la Figure 1.7. Chaque état du réseau est conservé en mémoire et le calcul en chaîne du gradient (*BP*) est effectué à la fin de l'inférence. Malheureusement, *BPTT* peut avoir une importante empreinte mémoire pour les réseaux de neurones avec plusieurs milliards de paramètres et/ou plusieurs milliers d'itérations.

1.3 Réseaux de neurones à impulsions

Les neurones qui ont été présentés ci-dessus sont une approximation des neurones biologiques où ils communiquent grâce à une valeur représentant leurs fréquences d'impulsions [Maass et al., 1991]. Il est cependant prouvé que les neurones communiquent également grâce à la temporalité des impulsions [Thorpe and Imbert, 1989]. Pour représenter ce comportement, des modèles de neurones à impulsion ont été proposés

[Gerstner et al., 2014].

Un réseau de neurone à impulsions (*Spiking Neural Network* - *SNN*, en anglais), est un *ANN*, mais où les neurones, appelé neurones à impulsions, communiquent avec de bref signaux temporels plutôt que des valeurs réelles. Les neurones à impulsions s'échangent de l'information binaire grâce à des signaux de très courte durée (les impulsions) émis à un temps t, dont l'amplitude est pondérée par le poids synaptique. Une impulsion transporte donc une information binaire et une information temporelle contrairement aux neurones classiques où l'information est encodée avec un nombre réel. La Figure 1.8 montre un neurone à impulsions qui a pour entrée trois trains d'impulsions et pour sortie un train d'impulsion. Le neurone intègre les impulsions des neurones connectés et émet ses propres impulsions.

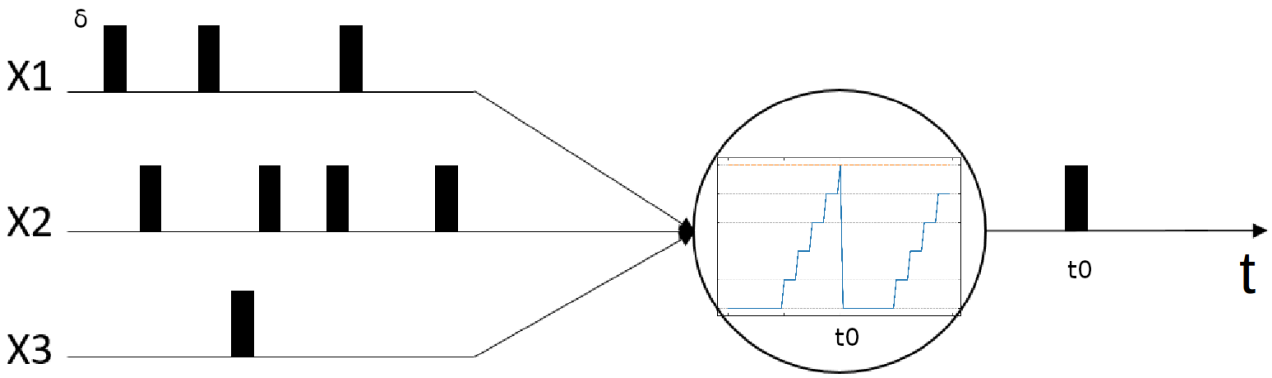


Figure 1.8 – Neurone à impulsion - il a pour entrée trois trains d'impulsions et pour sortie un train d'impulsion.

Les *SNNs* sont différents de la plupart des *ANNs* car ce sont des réseaux dynamiques. La valeur de sortie des neurones, et donc du réseau, évolue par rapport au temps (sauf les *RNNs* qui partagent aussi cette propriété). La temporalité des impulsions, en plus de l'impulsion en elle-même, encode de l'information. L'information supplémentaire encodée grâce au temps permet d'encoder plus d'information avec le même nombre de neurones que les neurones sans impulsion [Maass, 1997].

La dynamique globale du réseau dépend de la dynamique interne des neurones. Il existe plusieurs modèles de neurone à impulsions avec des caractéristiques influençant les temps des émissions d'impulsions et leur nombre.

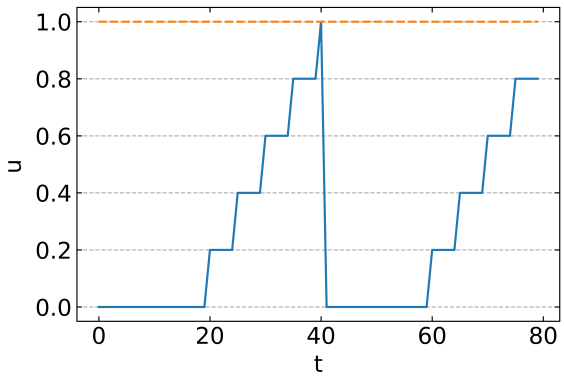
Un modèle basique de neurone à impulsions est le neurone **Integrate-and-Fire (IF)** [Gerstner et al., 2014]. Ce neurone possède une mémoire interne u, appelé *potentiel de membrane*, qui intègre en permanence l'entrée du neurone. Si le potentiel de membrane dépasse un certain seuil, alors ce neurone émet une impulsion et son potentiel de membrane retourne à une valeur noté u_{rest}. Quand ce neurone émet une impulsion, il rentre dans une période réfractaire $T_{refract}$ pendent laquelle il ne peut plus émettre d'impulsions. L'équation de sa dynamique au cours du temps est :

$$\frac{du}{dt} = RI(t), \tag{1.10}$$

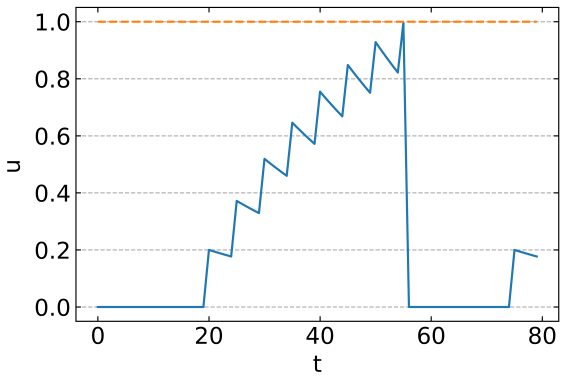
avec R une résistance interne propre au neurone et I l'entrée du neurone à un temps t :

$$I(t) = \sum_{j=1}^{N_{neuron}} w_{ij}\delta(t - t_j), \tag{1.11}$$

où δ est une fonction Dirac, N_{neuron} le nombre de neurones d'entrée et t_j le temps où chaque neurone d'entrée émet une impulsion. Une illustration du comportement d'un neurone **IF** est représentée sur la Figure 1.9a où le neurone reçoit une impulsion d'amplitude 0.2 tous les 5 pas de temps à partir du temps t_0, avec le seuil à 1 (ligne orange). Quand le seuil est dépassé, le neurone émet une impulsion et u devient égal à 0 pendent 20 pas de temps ($u_{rest} = 0$). Le modèle IF n'est pas biologiquement plausible, car un neurone biologique ne conserve pas son potentiel de membrane constant, si $u \neq u_{rest}$, sans signal d'entrée.



(a) Potentiel de membrane d'un neurone **IF**. Le neurone émet une impulsion à $t = 40$.



(b) Potentiel de membrane d'un neurone **LIF** avec $\tau_m = 0.03$. Le neurone émet une impulsion à $t = 55$.

Figure 1.9 – Potentiel de membrane d'un neurone **IF** (a) et d'un neurone **LIF** (b) au cours du temps, avec le seuil à 1 (ligne orange). Quand le seuil est dépassé, le neurone émet une impulsion et u est égal à 0 pendant 20 pas de temps (réfraction). Les neurones reçoivent une impulsion de 0.2 tout les 5 pas de temps à partir du temps t.

Cette considération a conduit à autre modèle fréquemment utilisé dans la littérature neuromorphique et d'apprentissage artificiel, appelé **Leaky-Integrate-and-Fire (LIF)** [Gerstner et al., 2014]. Ce modèle se comporte comme un neurone **IF**, mais le potentiel de membrane décroît au cours du temps selon un facteur τ_m, comme illustré sur la Figure 1.9b avec $\tau_m = 0.03$. L'équation de la dynamique d'un neurone **LIF** au cours du temps est :

$$\tau_m \frac{du}{dt} = -(u - u_{rest}) + RI(t), \tag{1.12}$$

Une limitation potentielle des neurones **LIF** ou **IF** est que la mémoire des précédentes impulsions est perdue pendant la réinitialisation du potentiel de membrane lors de l'émission d'une impulsion, supprimant ainsi de l'information dans la dynamique du réseau. Un autre défaut est le manque d'adaptation des neurones à une entrée constante, par exemple ne réduisant pas sa sensibilité à un signal trop fort constant qui peu être considéré comme peu utile comparé à d'autres informations d'entrée.

D'autres types de neurones permettent de régler ces problèmes comme les modèles LIF adaptatifs (**AdEx**) [Gerstner and Brette, 2009] qui ont des paramètres internes permettant une adaptation à une entrée constante ou les **Escape Noise Model** [Gerstner, 2008] qui sont des modèles plus génériques permettant une émission d'impulsion stochastique avec un seuil dynamique. Nous ne rentrons pas dans les détails de ces modèles, car ils sont plus complexes à mettre en œuvre matériellement que le modèle *LIF*. Pour cette raison, les travaux de cette thèse considèrent les modèles plus simples des **LIF** et **IF**, qui sont en grande majorité utilisés dans les implémentations neuromorphiques.

1.3.1 Entraînement des *SNNs*

Pour effectuer un apprentissage supervisé d'un *SNN*, il faut dans un premier temps pouvoir transformer la sortie du réseau en une valeur exploitable par une fonction de coût. Pour encoder l'erreur d'un SNN, il y a deux méthodes principales.

La première méthode est de traiter le train d'impulsions comme une valeur correspondant à la fréquence de celui-ci. Cela nécessite de mesurer la fréquence des neurones de sortie pour pouvoir exploiter cette valeur avec la fonction de coût, comme vu précédemment. Une variante consiste à compter le nombre d'impulsions de chaque neurone de sortie pendant un temps donné, ce qui revient à mesurer la fréquence sans normalisation temporelle.

La seconde méthode est d'utiliser le moment d'émission des impulsions de la couche de sortie comme valeur pour la fonction de cout. Le plus souvent le premier neurone de sortie à émettre est la classe de sortie

sélectionné (*winner take all*, en anglais), la valeur de sortie de ce neurone sera alors considéré comme à 1 et la valeur des autres neurones comme à 0 pour la fonction de coût. Il est aussi possible d'attendre un certain temps T et de définir la valeur de sortie comme $T - t_i$ avec t_i le temps de la première émission d'impulsion du neurone i. La méthode qui utilise le temps est préférable à la méthode avec les fréquences dans les cas où le moment d'émission des impulsions est plus important que la fréquence des neurones. De plus, cela limite le nombre d'impulsions du réseau, source de consommation d'énergie. Encoder l'information avec des fréquences consomme de l'énergie à chaque impulsion. Dans une optique de réduction énergétique, il peut être avantageux d'encoder l'information dans une latence des impulsions.

Pour utiliser la méthode de la descente de gradient pour apprendre les poids synaptiques, il faut pouvoir calculer le gradient par rapport à l'erreur, qui dépend de la méthode choisie au-dessus, pour chaque poids synaptique du réseau.

Les impulsions étant des événements discrets, donc non dérivables, un *SNN* ne peut pas être entraîné directement avec *BP*. Pour pouvoir appliquer *BP* aux *SNNs* il est nécessaire d'utiliser des astuces pour rendre dérivable l'activation des neurones. [O'Connor and Welling, 2016] calcule la fréquence du train d'impulsion d'un neurone en faisant l'hypothèse qu'elle est lié linéairement à l'entrée du neurone (pour des valeurs positive), ce qui revient à une approximation de la fonction *ReLU*. Cela permet d'utiliser les fréquences des neurones pour effectuer *BP* sur les fréquences plutôt que sur les impulsions. [Lee et al., 2016] utilisent la variation du potentiel de membrane des neurones comme un signal dérivable, en considérant comme du bruit le changement brusque lors de l'émission d'une impulsion, pour permettre de rétropropager le gradient à travers le neurone et ainsi effectuer *BP*. [Kheradpisheh and Masquelier, 2020] encodent la valeur de sortie des neurones en fonction du temps d'émission de l'impulsion t du neurone *IF*, la valeur est $T - t$ avec T une constante temporel. Chaque neurone ne peut émettre qu'une unique impulsion. Le gradient est ensuite trouvé en rétro propageant l'erreur sur le moment des émissions des impulsions. Cette méthode nécessite une fenêtre temporelle bien définie pour encoder l'entrée des neurones et l'erreur.

En utilisant *BP* sur les fréquences ou sur le potentiel de membrane pour entraîner les *SNNs*, on ne tire pas partie des impulsions et de leur temporalité relative aux impulsions émises par les neurones proches, comme c'est le cas dans le cerveau. Il est cependant possible d'utiliser cette temporalité pour apprendre les poids localement et ainsi éviter le problème de transfert du gradient d'erreur à travers le réseau.

La loi d'*Hebb* (ou apprentissage hebbien) [Hebb, 1949] est un modèle biologique qui permet de renforcer les connexions des synapses si le neurone présynaptique émet des impulsions juste avant que le neurone postsynaptique émette une impulsion. Cette loi est locale et ne prend en compte que l'information des neurones connectés à la synapse comme montré dans la Figure 1.10. Une interprétation possible est que si un neurone présynaptique (neurone j avant la synapse) émet une impulsion avant que le neurone postsynaptique (neurone i après la synapse) émette une impulsion, alors il y a corrélation qui est interprété comme de la causalité et le poids synaptique est augmenté. Le mécanisme inverse existe et est appelé anti-Hebbien.

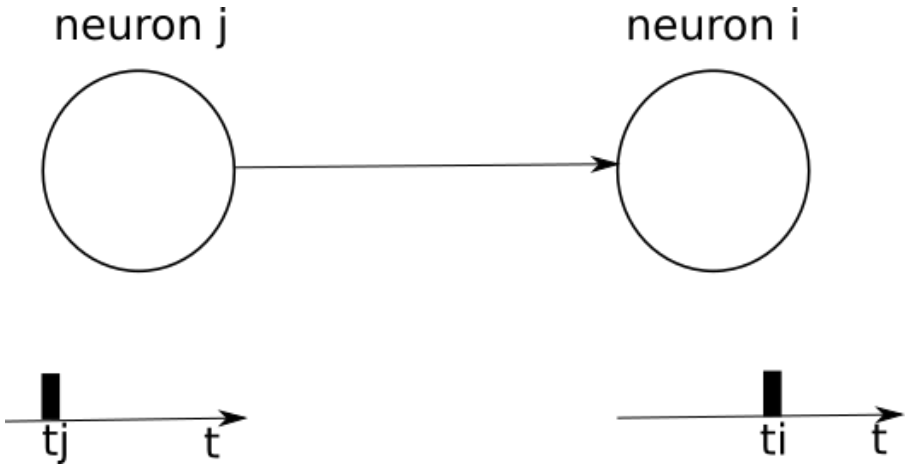


Figure 1.10 – Illustration de deux neurones à impulsions connectés par une synapse. Le neurone j est appelé neurone présynaptique et le neurone i neurone postsynaptique.

Une classe d'algorithmes plus complexes, inspiré de la loi *Hebbienne*, permet d'ajuster les poids en fonction de la différence temporelle des impulsions, la *STDP* [Bi and Poo, 2001] (*Spike-Timing-Dependent*

Plasticity, en anglais). Les différences temporelles entre les impulsions présynaptique j et postsynaptique i des neurones modifient le poids synaptique suivant une fonction f_w :

$$\Delta w_{ij} = f_w(t_i - t_j) \tag{1.13}$$

avec t_i le moment d'impulsion du neurone i et t_j le moment d'impulsion du neurone j.

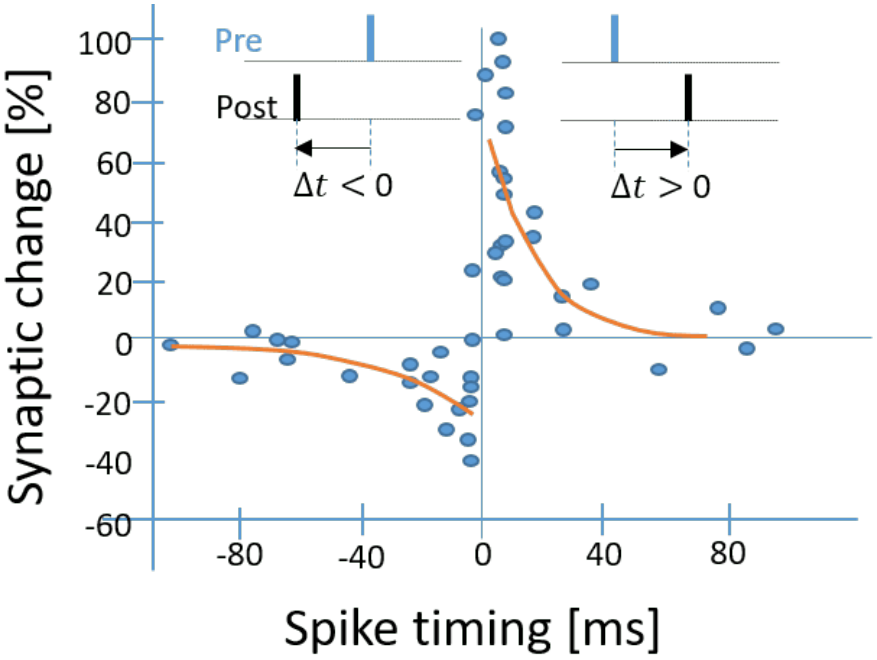


Figure 1.11 – Illustration de la règle d'apprentissage *STDP*, reproduite de [Bi and Poo, 2001].

Il est courant que la fonction f_w soit une inverse comme illustré sur la Figure 1.11 reproduite de [Bi and Poo, 2001] où chaque point est une mesure réelle de l'évolution du poids synaptique des neurones biologiques en fonction de la différence temporelle entre deux impulsions et la courbe orange est la représentation de la fonction f_w.

Une règle généralisé de la *STDP* pour un ensemble d'impulsion est :

$$\Delta w_{ij} = \sum_{n=1}^{N_i} \sum_{f=1}^{N_j} f_w(t_i^n - t_j^f) \tag{1.14}$$

avec t_i^n et t_j^f la n-ème et f-ème impulsion des neurones i et j, respectivement et N leur nombre de d'impulsion.

La *STDP* est un type d'algorithme qui cherche à modéliser un comportement observé en biologie, pour sa version de base, pour de l'apprentissage non supervisé et pour des réseaux peu profonds. Utiliser la *STDP* pour entraîner des réseaux de neurones de manière non-supervisée (93.5% [Querlioz et al., 2013] avec *STDP* sur *MNIST* [LeCun et al., 2010]) donne en général de moins bons résultats que *BP* (98.8% en supervisé sur MNIST [Lee et al., 2016]), ou que des *autoencoder* (pour l'apprentissage non supervisé) [Falez et al., 2019]. Un *autoencoder* est un algorithme pour des réseaux multicouches avec autant de neurones de sortie que de neurones d'entrée. En général les couches cachées ont moins de neurones que la couche d'entrée et l'algorithme minimise la différence entre l'entrée et la sortie (fonctionne pour les *ANNs* et les *SNNs*). Plusieurs variations de la *STDP* existent pour de l'apprentissage supervisé, ces algorithmes sont présentés dans la section 1.5.

Les *SNNs* ne sont pas optimisés intrinsèquement pour des architectures matérielles largement parallèles comme les *GPUs* ou les *TPUs*, car la prise en compte du temps dans l'algorithme oblige un calcul séquentiel, par nature non parallélisable. En revanche, les *SNNs* peuvent consommer moins d'énergie que les ANNs sur du matériel adapté [Davidson and Furber, 2021]. Il est donc nécessaire d'avoir un matériel optimisé pour les *SNNs* pour profiter au maximum de leurs caractéristiques.

1.4 Implémentations physiques optimisées pour les réseaux de neurones

Pour profiter du pouvoir de généralisation des *ANNs/SNNs*, il faut une forte puissance de calcul pour implémenter des réseaux à plusieurs millions/milliards de paramètres, ce qui entraîne une consommation massive d'énergie. Par exemple, un *GPU* consomme plusieurs centaines de Watts, comparé à une vingtaine pour le cerveau. De plus, le nombre de données à fournir pour l'apprentissage, combiné au nombre d'itérations nécessaire au réseau sur chaque donnée d'entrée, rend extrêmement coûteux en temps son apprentissage.

La surconsommation d'énergie et de temps des architectures matérielles actuellement les plus répandues, comme les *CPUs* et les *GPUs*, pour le calcul de l'inférence et surtout de l'apprentissage des *ANNs/SNNs*, s'explique par l'architecture *Von Neumann*. L'architecture *Von Neumann* sépare la mémoire de l'unité de calcul, ce qui demande des aller-retours incessants de données entre la mémoire et l'unité de calcul. De plus, la structure des *ANNs/SNNs* les rend particulièrement efficaces sur une implémentation matérielle massivement parallèle. En effet, chaque neurone d'une même couche peut être en général calculée indépendamment des autres, car les entrées d'un neurone dépend des sorties des neurones de la couche précédente. En revanche, les réseaux bio-inspirés peuvent atteindre des milliards de synapses, ce qui est plusieurs ordres de grandeur supérieurs au nombre de processus traitables en parallèle par les *GPUs*.

L'architecture *Von Neumann* est représentée schématiquement sur la Figure 1.12. Les données en mémoire sont stockées ailleurs que sur l'unité de calcul, qui est aussi séparée de l'unité de contrôle et chaque donnée traitée doit être déplacée pour effectuer un calcul. Le coût en temps et en énergie du transfert de la mémoire à l'unité de calcul est 200 fois supérieur au coût d'une opération *MAC* (*Multiply–ACcumulate*, en anglais) d'un CPU [Sze et al., 2017]. Pour réduire l'énergie et accélérer le calcul du réseau, il faut utiliser des architectures qui sortent de la structure *Von Neumann*.

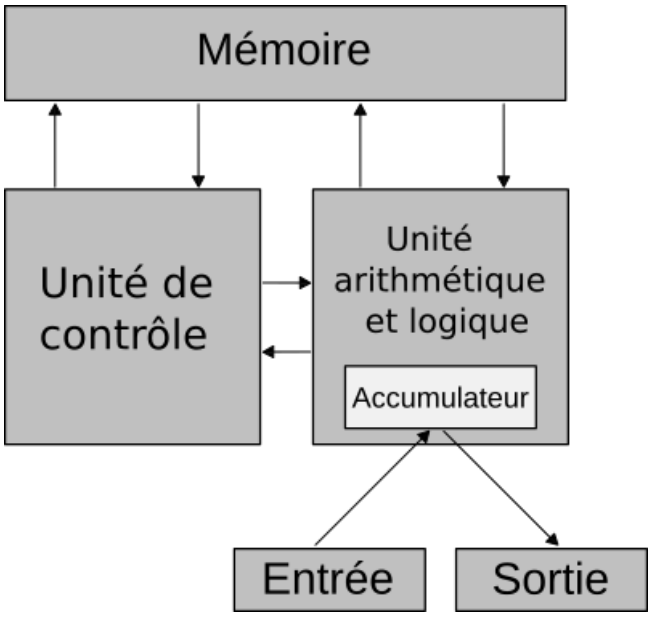


Figure 1.12 – Architecture Von Neumann. La mémoire est séparée de l'unité de calcul et de l'unité de contrôle. Échanger des données entre la mémoire et l'unité de calcul consomme beaucoup d'énergie comparé à l'énergie dépensée pour le calcul. (Schega, CC BY-SA 3.0, via Wikimedia Commons)

Une architecture massivement parallèle nécessite des transferts d'informations entre chaque neurone pour implémenter un *ANN*, informations qui sont stockées dans une mémoire éloignée des neurones sur une architecture *Von Neumann*. Il est préférable d'avoir une unité de calcul émulant le moins de neurones possibles afin de paralléliser au maximum l'implémentation physique et une mémoire (les poids synaptiques) la plus proche possible des unités de calculs. Ce type d'architecture est appelée *Beyond Von Neuman*. Idéalement, il faudrait un bloc de calcul par neurone, que la mémoire soit contenue dans la synapse qui connecte les unités de calcul et que la modification de l'information se fasse au moment de traverser la synapse.

Avec une architecture neuromorphique (optimisée pour représenter la structure neurone/synapse du cerveau) CMOS, le gain possible est de un à deux ordres de grandeur comme présenté sur la Figure 1.13 reproduit de [Zhang et al., 2020b] qui compare l'efficacité énergétique (milliard d'opérations, *GOP*, par seconde par watt) pour l'apprentissage atteignable pour des *ANNs* sur *CPU*, *GPU*, architecture CMOS *Beyond Von Neumann* (SRAM) et architecture neuromorphique avec des mémoires non volatiles (*Non Volatile Memory - NVM*, en anglais).

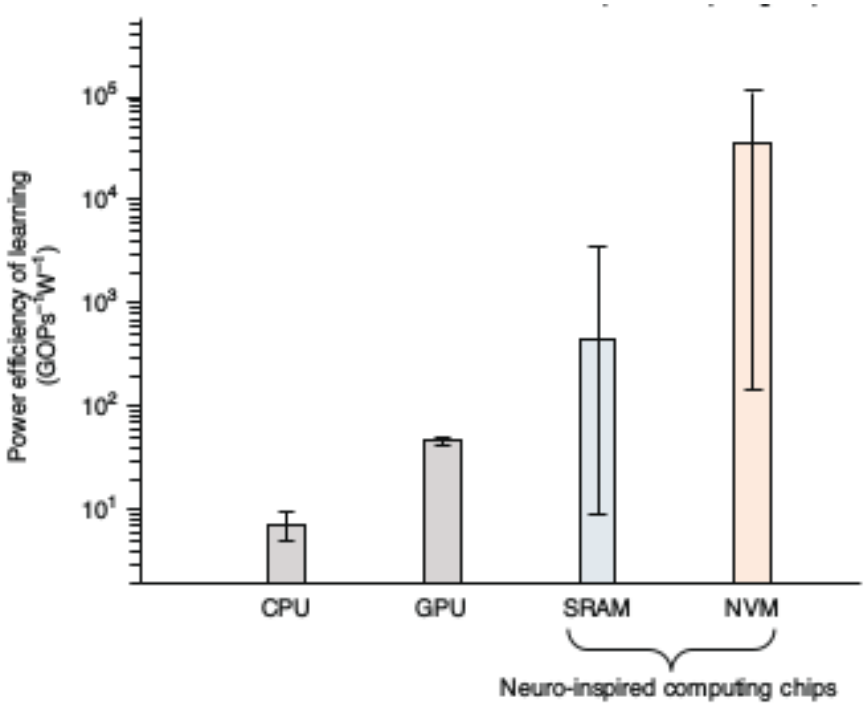


Figure 1.13 – Efficacité de puissance (milliard d'opérations, *GOP*) par seconde par watt, en fonction de l'architecture utilisée. Reproduit de [Zhang et al., 2020b]

Plusieurs implémentations CMOS *Beyond Von Neumann* ont été proposées. Elles peuvent être séparées en trois méthodes : les méthodes analogiques, les méthodes avec des signaux mixtes et les méthodes entièrement numériques. Les deux premières méthodes exploitent la physique des semi-conducteurs pour implémenter la dynamique du réseau directement sur la puce, ce qui réduit l'énergie dépensée pour une sensibilité au bruit et aux erreurs de calcul plus importante dû à la nature analogique des synapses. La dernière méthode utilise des portes logiques et des fonctions abstraites pour simuler le comportement des neurones/synapses, réduisant la sensibilité au bruit et aux erreurs de calcul, mais augmentant souvent la surface en silicium requise [Frenkel et al., 2021].

Certaines puces utilisent un mélange de signaux analogiques et numériques. Par exemple, *BrainScaleS* [Schemmel et al., 2010] est constitué de puces *HICANN (High Input Count Analog Neural Network)* interconnectées possédant jusqu'à 40M de synapses et 180k neurones à impulsions *AdExp*. L'interconnexion est effectuée grâce à des *FPGAs*. Malheureusement, la modification des poids synaptiques de la puce doit se faire par un logiciel externe à la puce.

En exploitant le transistor MOS en régime *sub-threshold,* il est possible d'émuler le comportement d'un neurone biologique. *Neurogrid* [Benjamin et al., 2014] est une puce à 16 cœurs de calcul de 64K neurones utilisant ce principe. Les neurones sont connectés à l'intérieur des cœurs par des synapses, mais les poids synaptiques doivent être modifiés par un ordinateur externe à la puce.

La puce *ROLLS* [Qiao et al., 2015] utilise aussi le transistor MOS en régime *sub-threshold,* avec un processus de fabrication CMOS de 180 nm pour avoir un réseau de 128 K synapses analogiques et 256 circuits de neurones. *ROLLS* permet un apprentissage sur puce grâce à la moitié des synapses qui possèdent une plasticité long terme et l'autre moitié une plasticité court terme [Brader et al., 2007]. *DYNAPs* [Moradi et al., 2018] est une augmentation de taille où plusieurs puces *ROLLS* sont regroupées sur un réseau matriciel avec une communication optimisée. Malheureusement, le besoin de communication entre les puces augmente l'énergie dépensée, ce qui réduit le gain énergétique obtenu au global.

Du matériel entièrement numérique simule le comportement des neurones en rapprochant la mémoire des cœurs de calcul pour être plus efficace que des architectures *Von Neumann*. *SpiNNaker* [Galluppi et al., 2014] est une architecture *Von Neumann distribuée* pour supercalculateur développé par l'université de Manchester. Il est composé de 57k nœuds, chacun composé de 19 processeurs *ARM*, et il est capable de simuler un milliard de neurones.

Des puces numériques entièrement crées pour des réseaux de neurones ont été également développées. La puce *TrueNorth* [Merolla et al., 2014], développée par IBM, possède 1M neurones à impulsions LIF simplifiés et 256M synapses. Les synapses peuvent seulement avoir un poids binaires et sont encodées grâce à une mémoire volatile qui oblige à recharger les poids à chaque arrêt de la puce.

Loihi [Davies et al., 2018] développée par Intel possède 128 cœurs capables de simuler 1024 neurones LIF avec les synapses, avec une précision des poids synaptique jusqu'à 9 bits et capable d'apprendre selon des règles programmables. De plus, des *ANNs* plus gros ont pu être simulés en connectant plusieurs puces *Loihi* entre elles (*Pohoiki Beach* avec 64 puces connectées, *Pohoiki Springs* avec 768 puces connectées), mais la communication entre les puces pose des problèmes d'adressage de neurone et rajoute un surcoût sur la consommation d'énergie du système.

ODIN [Frenkel et al., 2019a] est une puce compacte avec 256 neurones et 64k synapses de 4 bits capable d'un apprentissage sur puce. *ODIN* est capable d'exprimer des types de neurones à impulsions plus complexes comme les *AdExp* contrairement *Loihi* et *TrueNorth* qui utilisent des modèles LIF. *MorphIC* [Frenkel et al., 2019b] est une augmentation de la taille de la puce avec quatre cœurs *ODIN*, 2k neurones *LIF* et 2M synapses binaires.

Même si ces puces sont capables de simuler plusieurs millions de synapses pour les plus performantes, les modèles de réseaux de neurones les plus performants peuvent posséder des milliards de paramètres (par exemple, GPT-3 possède 175 milliards de paramètres [Brown et al., 2020]). Ces implémentations sont limitées par leurs densités, en effet chaque neurone et synapse nécessitent plusieurs transistors et le nombre de synapses est limité par la mémoire *SRAM* possible dans une surface donnée. Augmenter le nombre de synapses ou de neurones nécessite ainsi plusieurs puces connectées entre elles, ce qui réduit le gain énergétique des puces à cause du transfert d'information supplémentaire entre les puces.

Pour aller plus loin dans la densité et l'énergie économisée, on peut utiliser des nouveaux composants plus compacts et, pour les synapses, capables d'enregistrer des informations de manière non volatile en rapprochant encore plus la synapse du neurone. Pour cela, des composants non volatiles émergents peuvent être utilisées. Les neurones à base de *NVMs* sont appelés *neuristors* mais ne seront pas étudiés dans cette thèse, car nous nous sommes concentrés sur les synapses, bien plus nombreuses que les neurones dans la plupart des cas.

1.4.1 Utilisation de nanocomposants émergents pour augmenter l'intégration de la mémoire avec le calcul

[Strukov et al., 2008] propose une implémentation physique avec des *NVM* comme poids pour les synapses qui seront appelés des *memristors*. Par la suite, nous appellerons *memristors* tout type de *NVMs* utilisé comme des synapses. Les memristors sont des composants électroniques passifs qui se comportent comme des résistances programmables et non volatiles. Leur conductance est modifiée grâce à des impulsions de tension appliquées à leurs bornes (2 à 3 bornes en fonction du composant). Ces composants permettent d'obtenir des synapses efficaces [Jo et al., 2010] en utilisant leur conductance comme poids synaptique. De plus, les memristors sont compatibles avec la fabrication des composants CMOS, facilitant leur intégration dans des circuits.

Les memristors peuvent être utilisés comme synapses pour relier physiquement deux neurones grâce à une architecture en réseau matriciel (*crossbar array*). Si les neurones communiquent par des signaux de tensions, alors la résistance du composant va modifier l'intensité du signal en le pondérant. La Figure 1.14 illustre trois neurones d'entrée et trois neurones de sortie reliés par des synapses de type memristif en réseau matriciel. Les courants aux sorties du *crossbar*, I_j, sont le produit entre les tensions d'entrées, V_i et la

conductance des memristors, G_{ij}, selon la loi d'Ohm et les lois de Kirchhoff :

$$I_j = \sum_i G_{ij} V_i \tag{1.15}$$

Grâce à cette implémentation, avec G_{ij} représentant les poids synaptiques et V_i l'entrée du neurone, le circuit va effectuer une somme pondérée des entrées du neurone par les poids synaptiques. Cette implémentation permet donc de réaliser avec un seul composant la fonction mémoire et la fonction de calcul de la synapse. En effet, la conductance du memristor est programmable (mémoire) et le signal électrique qui le traverse va être modifié par la résistance du composant (fonction de calcul).

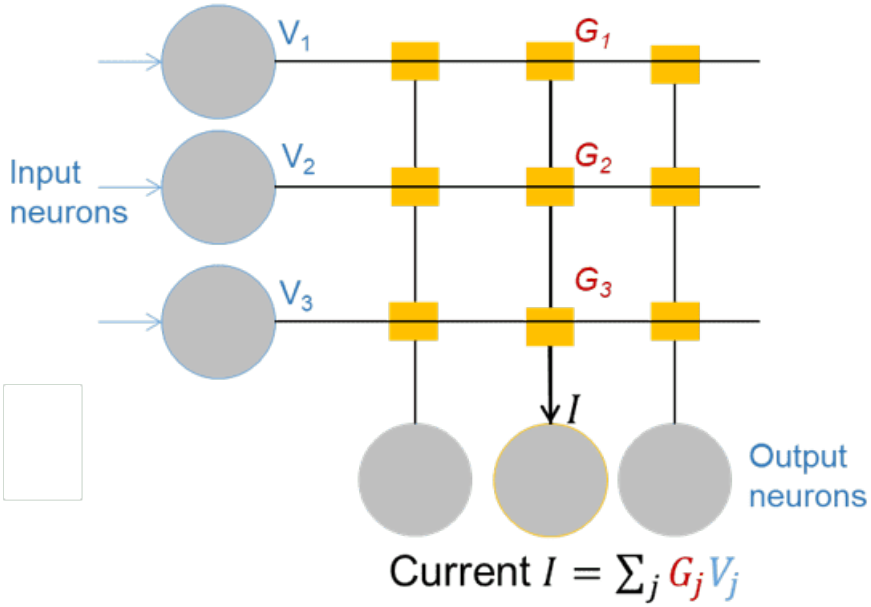


Figure 1.14 – Illustration d'une couche de neurones connectés par un *Cross-bar array* de NVM. Les rectangles représentent les conductances G des synapses et les cercles représentent les neurones LIF qui émettent une tension V et intègrent un courant I.

Les memristors sont séparés en quatre grandes catégories :

- Les composants pour lesquels la phase d'un matériau change en se cristallisant ou en devenant amorphe ce qui change la résistance du composant (Phase Change Memory - PCM) [Raoux et al., 2008],
- Les memristors où des filaments conducteurs nanoscopiques sont créés et manipulés pour changer la résistance du composant comme les *Conductive Bridge Random-Access Memory (CBRAM), en anglais* [Chen et al., 2016] ou les *Resistive Random-Access Memory (RRAM), en anglais* [Jo et al., 2010],
- Les composants pour lesquels la modification de la résistance est due à des effets magnétiques, comme les *Magnetic Random-Access Memory (MRAM), en anglais* [Song et al., 2018],
- Et les composants ferroélectriques (*FeFET, FeRAM*) [Jerry et al., 2017, Chanthbouala et al., 2012].

Le principe des *PCMs* est de faire cristalliser ou de rendre amorphe un matériau semi-conducteur ce qui change sa résistance, comme schématisé sur la Figure 1.15a. Les *PCMs* ont une forte asymétrie sur l'augmentation et la diminution des conductances. En particulier l'augmentation de la conductance nécessite de faire fondre le matériau, ce qui consomme plus d'énergie. En revanche, ce type de composant à une meilleure plage de résistance entre les états haut et bas que d'autres technologies comme les *RRAMs* (Tableau 1.1). Les *PCMs* ont été utilisés dans des systèmes neuromorphiques avec de bonnes performances de classification [Burr et al., 2015] mais souffrent de leur forte asymétrie non linéaire ainsi que d'une dérive de la résistance au cours du temps qui rend difficile un apprentissage sur puce sans des circuits de contrôle coûteux en surface et en énergie.

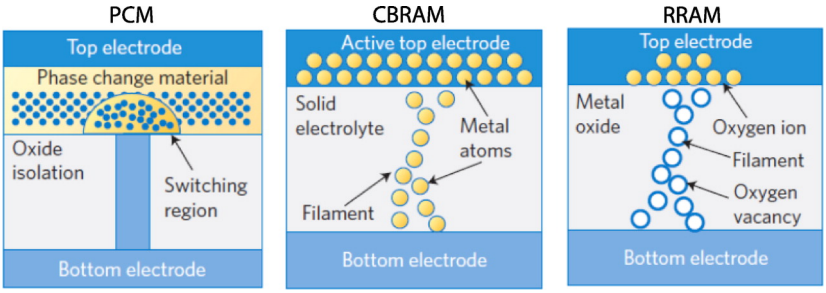


Figure 1.15 – Comportement interne pour les *PCM*, CBRAM et RRAM (a) La *PCM*) dépend de la grande différence de résistivité électrique entre les phases amorphes (faible conductivité) et cristallines (forte conductivité) des matériaux dits à changement de phase. (b) La *CBRAM* est basée sur la formation électrochimique de filaments métalliques conducteurs à travers un électrolyte ou un oxyde solide isolant. (c) Les filaments conducteurs dans une *RRAM* filamentaire sont des chaînes de défauts à travers un oxyde en couche mince autrement isolant. Figure reproduite de [Burr et al., 2017]

Les *RRAMs* sont composées d'électrodes métalliques séparées par un oxyde isolant. Lorsqu'une tension suffisamment élevée traverse le composant, le champ électrique généré crée ou détruit un filament conducteur dans le matériau isolant (usuellement du TiO_x ou du HfO_x), comme présenté sur la Figure 1.15c. Les *CBRAM* [Chen et al., 2016] ont un comportement similaire, mais le filament est formé par le mouvement d'ions métalliques comme du cuivre. La différence est donc la nature de l'isolant et des matériaux utilisés pour créer le filament, comme illustré sur la Figure 1.15b.

Le développement des *RRAMs* est récent et la technologie est moins mature que d'autres technologies comme les *PCM*, ce qui induit une plus grande variance entre les composants, avec une endurance relativement faible, de l'ordre de 10^{5-8} écritures (Tableau 1.1). Ceci est problématique pour des réseaux de neurones où les poids synaptiques peuvent être modifiés plusieurs millions (voir milliards) de fois.

Les *MRAMs* sont composées de deux aimants séparés par une couche isolante. L'aimantation d'une couche est fixée et l'aimantation de l'autre couche peut être modifiée grâce au couple de transfert de spin exercé par un courant spin polarisé à travers une jonction tunnel. Contrairement aux autres composants memristifs, il est compliqué de les utiliser en réseau matriciel utilisant les lois de Kirchoff car leurs variations de résistance sont faibles, ce qui induit des problèmes de *sneak path*.

Une implémentation physique de réseaux matriciels avec des *MRAMs* n'a été effectuée que récemment pour des réseaux binaires [Jung et al., 2022] grâce à une somme des résistances plutôt que des courants dans le réseau matriciel.

Le nombre d'états analogiques (Figure 1.16b) correspond au nombre d'états représentables par les composants. Les *ANNs* sur des ordinateurs ont souvent des poids synaptique encodés sur 32 bits soit 2^{32} nombres représentables, ce qui permet de changer précisément les poids synaptiques lors de l'apprentissage. Il est possible de binariser les poids une fois l'apprentissage fini [Hirtzlin et al., 2019] afin d'être moins dépendant de la précision des poids pendant l'inférence.

Le ratio ON/OFF (Figure 1.16c) est la différence entre l'état le plus haut de conductance et le plus bas, un ratio important pour pouvoir mieux séparer les deux états lors du calcul et ainsi éviter les erreurs. Le temps d'écriture et la tension à appliquer pour écrire le memristor doivent être les plus petits possibles pour réduire au maximum l'énergie dépensée par le circuit. Le Tableau 1.1 (adapté de [Ielmini and Ambrogio, 2020] et [Milo et al., 2020]) présente, de façon chiffrée, les différentes technologies pour les comparer.

Les memristors sont souvent fortement non-linéaires (Figure 1.16d) lors de l'écriture de la conductance : la variation de conductance obtenue suite à l'application d'une impulsion dépend de l'état de conductance initiale. Ceci qui nécessite de connaître la valeur réelle de la conductance ainsi que le coefficient de non-linéarité pour ne pas créer des erreurs d'écritures importantes.

De plus, les memristor sont asymétriques (Figure 1.16e) : le nombre d'impulsions pour accroître la conductance et la diminuer de la même valeur n'est pas le même. Couplé avec une forte non-linéarité, cela peut rendre compliqué un apprentissage sur puce sans des circuits de contrôle pour s'assurer d'avoir bien

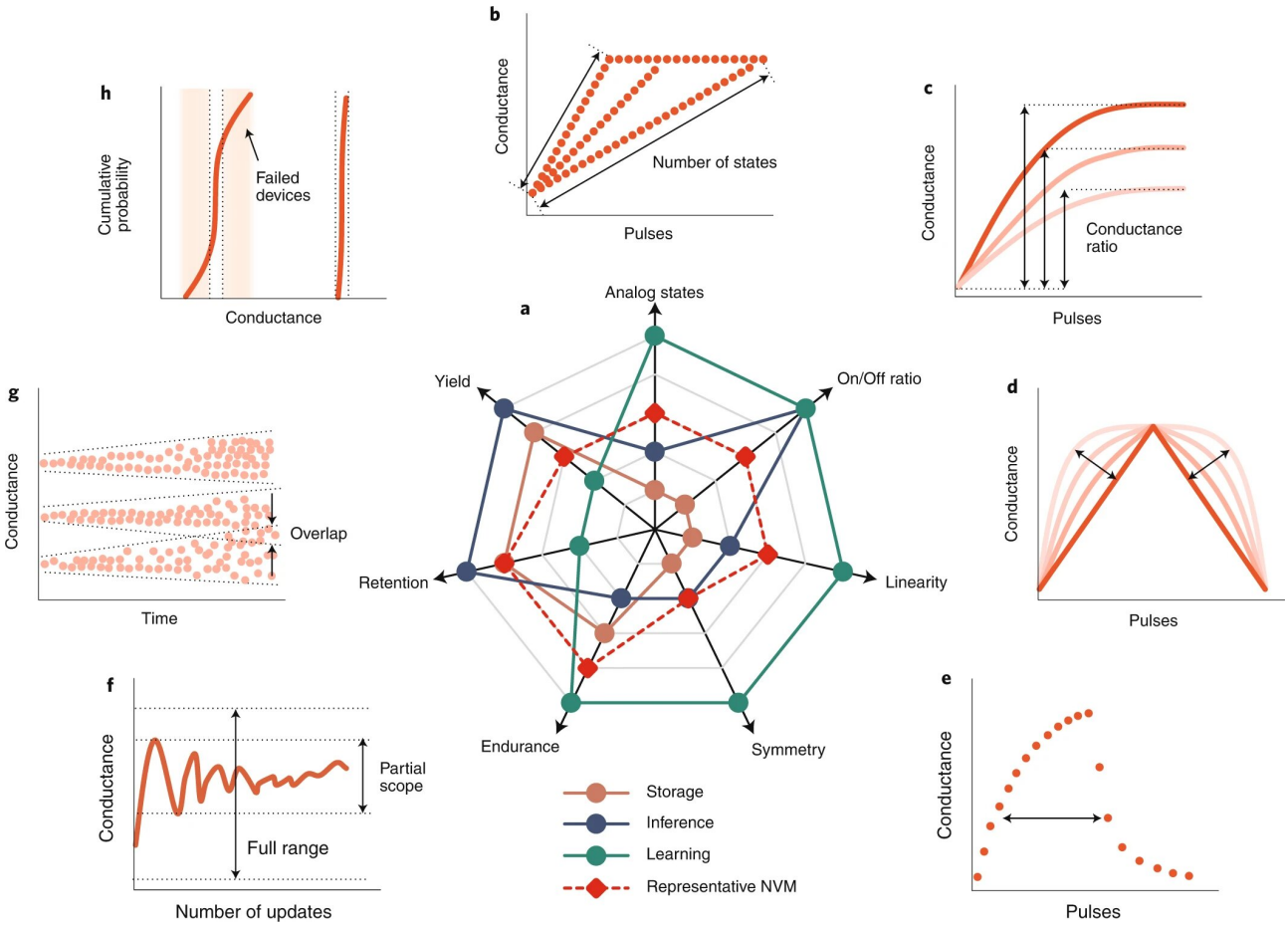


Figure 1.16 – Résilience aux défauts des *NVM* pour différentes fonctionnalités intéressante pour des puces neuromorphique. (a), Classement des exigences qualitatives des composants pour trois fonctionnalités potentielles. Une valeur plus grande sur un axe donné indique une exigence plus élevée pour la métrique correspondante. La ligne rouge pointillée représente les données expérimentales des NVMs qui ont été précédemment rapportées dans des travaux représentatifs. (b-h), Illustrations schématiques des exigences des composants memristifs pour le calcul : états analogiques (b), rapport état haut/bas (c), linéarité (d), symétrie (e), endurance (f), rétention (g) et rendement (h). Les courbes pointillées et pleines en b-e indiquent le réglage de la conductance d'un composant NVM analogique. Les mises à jour de la conductance d'un composant NVM au cours du processus d'apprentissage sont généralement dans une gamme partielle plutôt que dans la plage complète de la fenêtre de conductance (f). Après que les composants NVMs aient été réglés sur différents niveaux de conductance, la conductance des composants peut fluctuer dans le temps et deux niveaux peuvent se chevaucher (g). Les composants NVMs qui ne peuvent pas être réglés sur le niveau de conductance cible sont considérés comme des composants défaillants (h). Extrait de [Zhang et al., 2020b].

écrit le memristor. Ces circuits rajoutent un surcoût en énergie et en espace.

L'endurance (Figure 1.16f) correspond au nombre de fois qu'un composant peut être modifié sans altérer sa gamme de conductance. Les réseaux de neurones sont modifiés à chaque itération durant l'entraînement et il est donc important d'avoir une endurance élevée, en particulier pour un apprentissage sur puce.

Même si les memristors sont des composants non volatiles, leur conductance change légèrement au cours de temps. La Rétention (Figure 1.16g) représente cette dérive. Généralement l'apprentissage se fait suffisamment rapidement, comparé au temps nécessaire pour que les valeurs de conductance change, pour que ce phénomène soit impactant. En revanche, une fois le réseau appris, cela est problématique si la valeur des poids change et modifie les résultats de l'inférence.

Pour finir, ce type de composant doit être le plus petit possible pour accroître la densité sur une puce.

Les paramètres souhaités pour un bon composant NVM neuromorphique sont donc un faible coût énergétique pour l'écriture et la lecture, une écriture rapide, une densité d'intégration élevée, ainsi qu'une bonne rétention et une plage de valeur élevée. Une bonne endurance et une faible non-linéarité sont importantes si un apprentissage sur puce est effectué.

Le Tableau 1.1 (adapté de [Ielmini and Ambrogio, 2020] et [Milo et al., 2020]) présente, de façon chiffrée, les différentes technologies pour les comparer.

Les *MRAMs* ont un coût énergétique très faible pour l'écriture avec une très bonne endurance, mais souffrent résistance et d'une plage de valeur faible, souvent binaire. La faible résistance crée des problèmes de *sneak path*. Les *MRAMs* peuvent avoir plus d'état possible, mais le ratio ON/OFF est très faible. Cela rend la lecture et la différenciation des états compliqué [Sharad et al., 2012, Lequeux et al., 2016, Zhang et al., 2021].

Les *FeFET* possèdent un très faible coût d'écriture aussi, mais une endurance très faible comparé à celles d'autres composants, surtout pour des réseaux de neurones où les synapses peuvent être réécrites plusieurs centaines de milliers de fois.

Les *PCMs* et les *RRAMs* sont des composants très intéressants pour des applications neuromorphiques en raison de leur coût d'écriture modéré avec une bonne endurance ainsi qu'une bonne densité d'intégration et un nombre de valeurs encodables de plusieurs bits [Xi et al., 2021]. Pour la suite de la thèse, nous simulerons des *RRAMs* pour les raisons citées ainsi que pour leur linéarité meilleurs que celle des *PCMs*.

Technologie	RRAM	PCM	MRAM	FeFET
ON/OFF Ratio	$10 - 10^2$	$10^2 - 10^4$	1.5-2	5-50
Tension d'écriture	<3V	<3V	<1.5V	<5V
Temps d'écriture	<10ns	500ns	<10ns	10ns
Temps de lecture	<10ns	<10ns	<10ns	10ns
Énergie d'écriture (J/bit)	0.1-1pJ	10pJ	100fJ	<100fJ
Linéarité	basse	basse	-	basse
Densité d'intégration	haute	haute	basse	haute
Endurance	$10^5 - 10^8$	$10^6 - 10^9$	10^{15}	10^5

Table 1.1 – Comparaison de différents types de memristors en fonction des paramètres importants pour les synapses. Adapté de [Ielmini and Ambrogio, 2020] et [Milo et al., 2020].

1.4.2 Entraînement des réseaux de neurones composés de memristors pour synapses

Pour qu'un *ANN* soit performant sur une tâche précise, il doit posséder des poids synaptiques adéquats. Trois solutions existent pour obtenir les bons poids synaptiques sur une architecture neuromorphique avec des *NVMs* comme synapses.

La première solution est de faire de l'apprentissage par un ordinateur hors puce et de transférer les poids [Yao et al., 2020, Wan et al., 2020, Jung et al., 2022]. Cette méthode demande d'entraîner le réseau une seule fois sur un système consommateur en énergie. Cette option rend compliqué une mise à jour des poids synaptiques.

Une autre solution consiste à faire de l'apprentissage par un ordinateur hors puce et de transférer les poids pour disposer de l'avantage des poids pré-entraînés puis d'entraîner à nouveau le réseau sur puce pour devenir robuste aux variations internes des composants [Truong et al., 2014, Liu et al., 2019].

Une dernière solution est d'avoir un apprentissage sur puce. L'inférence est effectuée sur le réseau de neurones physique, mais la rétro propagation du gradient d'erreur est souvent effectuée grâce à un circuit externe au réseau de neurones. L'apprentissage sur puce permet d'exploiter un circuit dédié à cela, qui peut être beaucoup moins consommateur d'énergie que des *TPUs/GPUs* pour l'entraînement.

Avec un apprentissage hors puce, les poids synaptiques peuvent être initialisés sur la puce de façon précise grâce à des circuits de contrôle pour limiter les problèmes de variations inter-composants. En revanche, cette opération est longue en temps et doit être effectuée pour chaque puce. Avec un apprentissage sur puce, il est possible d'entraîner les puces avec les défauts des memristors, ce qui permet d'y être plus robuste dès l'entraînement [Querlioz et al., 2013, Lim et al., 2019]. Pour entraîner des *ANNs* sur puce, l'algorithme le plus utilisé est *BP*, mais le problème de la transposée des poids pour la rétro propagation et le besoin de mémoriser puis d'appliquer les gradients rend compliquée ou très coûteuse en surface et énergie l'implémentation physique de *BP*, comme présenté ci-dessous.

[Burr et al., 2015, Kataeva et al., 2015, Lim et al., 2019, Nandakumar S. et al., 2017] proposent un entraînement sur puce puis calculent le gradient d'erreur grâce à *BP* sur un processeur externe au circuit du réseau. Ils proposent une implémentation physique capable d'utiliser le réseau physique pour rétro propager le gradient à travers les couches, puis modifier les composants avec une impulsion, plutôt que l'utilisation d'un circuit externe. Pour simplifier le calcul de l'impulsion qui modifie le poids synaptique, il est possible d'envoyer une impulsion qui dépend seulement du signe du gradient, au prix d'une perte de précision finale [Prezioso et al., 2015]. [Li et al., 2018] proposent un circuit sur la puce qui permet de modifier les memristors de façon linéaire et symétrique pour s'adapter directement à la variance inter-composants.

[Hirtzlin et al., 2019] réalisent des *ANNs* binaires. Les synapses utilisées pour l'inférence sont constituées de deux *RRAMS*, la valeur binaire des poids dépend de la comparaison entre leur conductance ($G_1 > G_2$). Pour avoir une bonne performance, les poids binaires doivent avoir une valeur interne non binaire pendent l'entraînement, mais seulement la valeur binaire est utilisée à l'inférence. Les poids binaires permettent d'être robuste à certaines imperfections des synapses, comme le ratio haut/bas de la conductance, la dérive des poids ou la précision des poids synaptique, tout en permettant un apprentissage sur puce grâce au circuit numérique.

Les méthodes précédemment citées proposent des implémentations de *BP* pour des *ANNs*. Comme vu précédemment, les *SNN* peuvent consommer moins d'énergie que les *ANNs* [Davidson and Furber, 2021]. En revanche, calculer le gradient d'erreur avec *BP* sur les *SNNs* demande d'enregistrer une partie des états du réseau au cours du temps pour les dérivées. En revanche, les *SNNs* peuvent exploiter la dynamique des neurones de façon locale pour entraîner les synapses. Les algorithmes de type *STDP* sont des algorithmes locaux qui ne dépendent que de deux facteurs : les temps d'impulsion des neurones présynaptique et postsynaptique. Il s'agit en général d'algorithmes qui apprennent de façon non supervisée, avec des performances de classification bien en dessous des *ANNs* entraînés de façon supervisée. En revanche, il est possible de modifier l'algorithme de la *STDP* en rajoutant un troisième facteur pour encoder l'erreur globale du réseau de neurone, ce qui augmente ses performances (plusieurs de ses algorithmes seront présentés dans la section suivante). Pour un apprentissage sur puce des *SNNs* trois catégories d'algorithme ressortent : les méthodes basées sur les *Restricted Boltzmann Machine (RBM)*, les méthodes basées sur la *STDP* avec

deux facteurs et les méthodes qui propagent un gradient d'erreur grâce à un troisième facteur.

Les *RBMs* sont des réseaux de neurones avec un graphe biparti où une partie représente les couches d'entrée/sortie et où l'autre partie représente la couche cachée. L'apprentissage des *RBMs* est un algorithme non supervisé qui cherche à réduire l'*énergie du système*, qui correspond non pas à l'énergie physique, mais à une fonction du réseau. Malheureusement, les performances sur des problèmes supervisés ne sont pas aussi bonnes avec les *RBMs* qu'avec *BP* car l'information de l'étiquette n'est pas exploitée. De plus, la topologie du réseau est limitée à seulement deux couches, ce qui empêche d'exploiter les niveaux d'abstraction de couches supplémentaires. Dans une *RBM* à impulsions, les neurones émettent des impulsions de façon stochastique. [Suri et al., 2015, Chen et al., 2016, Parmar and Suri, 2018] utilisent des *RRAMs* pour entraîner des réseaux avec la *contrastive divergence (CD)* [Hinton, 2002]. La *contrastive divergence (CD)* est un algorithme non supervisé qui permet d'estimer la distribution de probabilité à partir des impulsions des neurones de sortie du modèle grâce à des chaines de *Markov* puis d'optimiser la log-vraisemblance du modèle par rapport à cette distribution de probabilité. [Ernoult et al., 2019a] propose d'utiliser des *PCMs* pour synapses. Les poids sont modifiés avec une impulsion d'amplitude constante, positive ou négative, en fonction du signe du gradient de l'erreur par rapport au poids, calculé par un ordinateur.

[Payvand et al., 2020] ont développé une puce pour des *SNNs*, composée de neurones faits avec des transistors et des synapses en *RRAM*. La puce possède un processeur dédié pour calculer le gradient d'erreur du réseau, puis le propage à travers la puce à chaque couche. Ils proposent une règle d'apprentissage à trois facteurs, les deux premiers facteurs sont les dynamiques des membranes des neurones connectés et sont donc local à la synapse, et le troisième facteur est l'erreur de la couche. Pour calculer l'erreur de la couche, un classifieur local est associé à chaque couche avec des poids aléatoires fixes et l'erreur est propagée à travers ce classifieur. Les poids synaptiques sont ensuite modifiés par une impulsion émise par un circuit externe au réseau basée sur la valeur de modification des synapses.

1.4.3 Utilisation des mécanismes des propriétés de changement de conductance des memristors pour implémenter de façon intrinsèque la *STDP*

Les méthodes d'apprentissages montrées précédemment demandent soit un ordinateur en plus du circuit du neurone, soit un contrôleur externe au réseau pour calculer et appliquer la loi d'apprentissage voulue, ce qui est coûteux en énergie et en espace. Pour éviter ces circuits supplémentaires, il faudrait concevoir un apprentissage où la règle d'apprentissage est calculée localement et où la modification de la conductance de la synapse, implémentée avec des *NVMs*, se ferait grâce aux impulsions des neurones. Nous appelons cette forme d'apprentissage *l'apprentissage intrinsèque*.

[Linares-Barranco and Serrano-Gotarredona, 2009, Querlioz et al., 2013, Prezioso et al., 2018, Milo et al., 2016, Falez et al., 2019] utilisent des *RRAMs* implémentant intrinsèquement la *STDP* grâce à des impulsions bipolaires émises par les neurones. La superposition des impulsions, avec une forme adéquate, permet d'écrire la valeur du memristor directement par le comportement des neurones et sans le besoin de circuit dédié externe aux neurones et aux synapses. [Ishii et al., 2019] utilise le même principe avec des *PCMs* et des *RBMs*, en revanche [Eryilmaz et al., 2013] l'utilise pour appliquer une loi de *Hebb* et non la *STDP*.

Une *RRAM* ou un *PCM* nécessite de recevoir une tension qui dépasse un seuil propre au composant pour que sa conductance soit modifié. Si deux impulsions sont juste en dessous du seuil, alors leur superposition permet de modifier le memristor comme montré sur la Figure 1.17 où deux impulsions sont superposées, ce qui permet de modifier la conductance du memristor.

L'apprentissage intrinsèque avec la *STDP* permet de s'affranchir des circuits externes pour l'apprentissage et permet d'être moins sensible à la variation inter-composant des memristors [Querlioz et al., 2013]. De plus, comme préciser à la section 1.3.1, utiliser la *STDP* pour entraîner des réseaux de neurone donne en général de moins bons résultat que *BP* (pour l'apprentissage supervisé) qui obtient 98.8% [Lee et al., 2016] contre 93.5% [Querlioz et al., 2013] avec *STDP* sur *MNIST* [LeCun et al., 2010]), ou que des *autoencoder* (pour l'apprentissage non supervisé) [Falez et al., 2019].

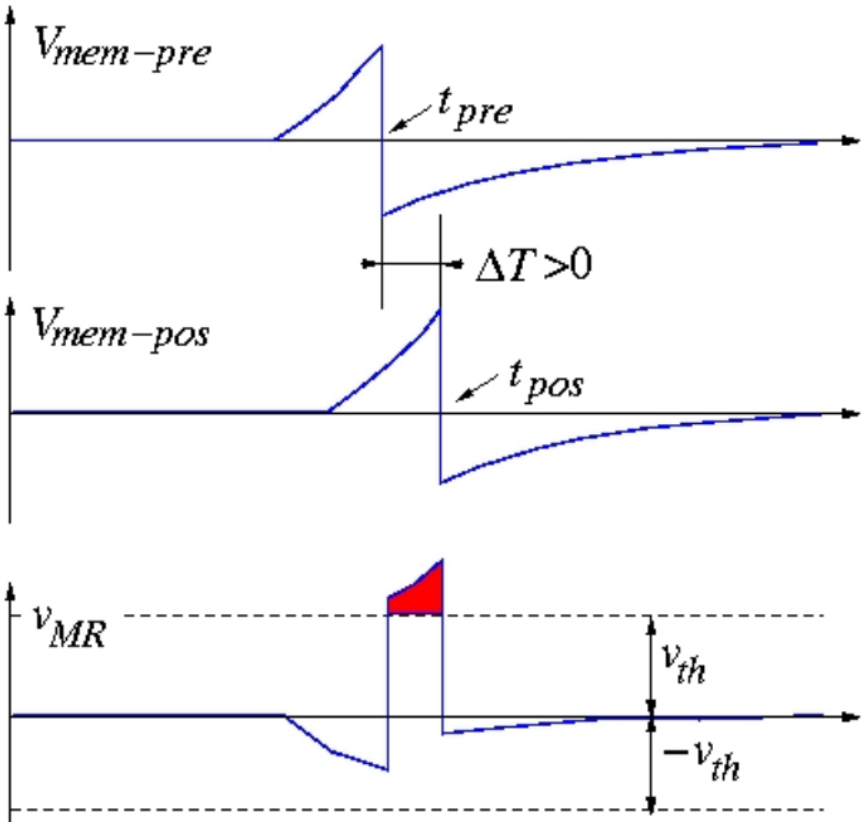


Figure 1.17 – *STDP* avec des memristors : représentation des tensions du neurone présynaptique, V_{mem_pre}, du neurone postsynaptique, V_{mem_post} et de la tension totale aux bornes du memristor, V_{MR}. Une impulsion pour chaque neurone est émise avec une différence temporelle Δ_t. La superposition des impulsions dépasse le seuil du memristor V_{th} ce qui modifie sa conductance d'une valeur qui dépend de la tension en rouge. Image extraite de [Linares-Barranco and Serrano-Gotarredona, 2009]

1.5 Vers un algorithme avec une règle d'apprentissage plus proche des synapses pour une implémentation matérielle efficace

Parmi les puces avec apprentissage existantes, une partie se base sur la rétro propagation du gradient. Cependant, cette méthode nécessite des circuits externes pour être réalisée. En effet, *BP* souffre de plusieurs problèmes comme le problème du calcul de la transposée des poids lors de la phase de rétro propagation ou du stockage des gradients avant leur application. Pour résoudre ces problèmes, il est possible de rendre le calcul du gradient plus local à la synapse. De nouveaux algorithmes existent réglant certains problèmes de *BP* pour permettre un apprentissage sur puce plus adapté aux contraintes du matériel physique pour une implémentation plus dense et plus économe en énergie. Dans cette section, nous allons étudier les différents algorithmes qui évitent certains problèmes de *BP* évoqués pour les *ANNs* et les *SNNs*.

Il y a 3 approches pour contourner le besoin de calcul explicite des gradients de *BP* :

— En permettant à chaque couche de neurone d'estimer son erreur grâce à un classifieur local à la couche. Cette stratégie évite de plus d'avoir à attendre la fin de l'inférence pour modifier les poids des premières couches. Ce problème est appelé le verrouillage des poids. Par contre, rajouter un classifieur par couche prend de la place et consomme de l'énergie.

— En utilisant la différence temporelle entre les impulsions comme les algorithmes inspirés de la *STDP*. Ces algorithmes peuvent être intrinsèques, mais pour avoir de meilleures performances en apprentissage supervisé, un troisième facteur venant de la couche de sortie est en général nécessaire.

— Certains modèles d'*ANNs* se basent sur la minimisation de leur énergie interne, comme les réseaux de Hopfield [Hopfield, 1982]. Le gradient d'erreur est encodé par l'évolution de leur dynamique interne qui est corrélée à leur énergie.

Nous allons maintenant détailler ces différentes classes d'algorithmes.

1.5.1 Évitement du problème de la transposée des poids

Avec *BP*, le gradient de l'erreur d'une couche i est $W_i^T f'_{i+1}(h_i)$, avec $f_{i+1}(h_i)$ la fonction d'activation de la couche $i+1$ et h_i la sortie de la couche i. La Figure 1.18a montre le chemin de la rétro-propagation en rouge avec la matrice de retour W_i^T. Cela impose de faire la transposée des poids pour propager le gradient, ce qui nécessite pour une implémentation physique soit de mesurer les poids, soit que le *crossbar array* fonctionne dans les deux sens comme proposé par [Burr et al., 2015]. Pour contourner ce problème, *Feedback alignement (FA)* [Lillicrap et al., 2016] propose d'utiliser des poids fixes et aléatoires pour rétro-propager l'erreur plutôt que les poids synaptiques du réseau. *FA* propose de remplacer la transposée des poids W_i^T par une matrice B_i constante initialisée aléatoirement. La Figure 1.18b montre en bleu le chemin de l'inférence pour quatre couches de neurones, et en rouge le chemin de la propagation de l'erreur avec une matrice B_i pour chaque couche.

Un problème de cette technique est la perte de performance comparée à la rétro-propagation du gradient d'erreur lors de la mise à l'échelle des réseaux, en particulier pour les *CNNs*. Par exemple, *FA* obtient seulement 64.4% de précision sur le jeu de données de classification d'images *CIFAR-10*[2] [Krizhevsky et al., 2009], comparé à 89% pour *BP* [Moskovitz et al., 2018]. Pour contourner ce problème, il a été proposé d'utiliser des poids de retour initialisés avec le même signe que les poids d'inférence [Moskovitz et al., 2018] permettant d'atteindre 86.9% de précision sur *CIFAR-10*[2].

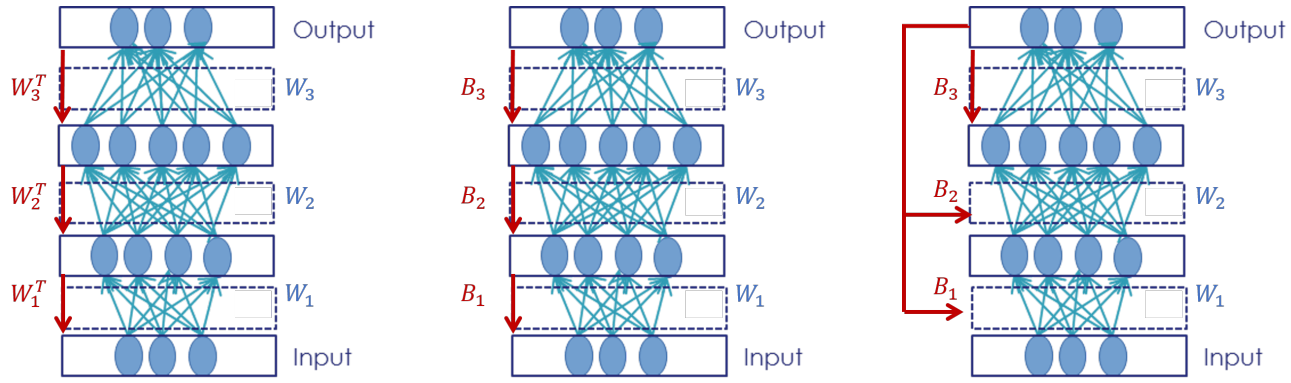
Une autre méthode, appelée *Direct Feedback alignement (DFA)* [Nøkland, 2016], propose d'éviter la propagation de l'erreur à travers les couches, mais plutôt de calculer l'erreur d'une couche par rapport à l'erreur global e du réseau et d'une matrice B_i tel que $\Delta W = (B_i \cdot e) \cdot f'_{i+1}(h_i)$. Le chemin de l'erreur est montré en rouge sur la Figure 1.18c, le calcul de l'erreur ne se fait plus à travers les couches une par une, mais se propage sur un chemin parallèle aux synapses.

Les performances sont en dessous de l'état de l'art pour les réseaux convolutifs 71.4% pour *CIFAR-10*[2], par exemple. L'apprentissage avec *FA* ou *DFA* permet de réduire le nombre de mesures des poids pour calculer le gradient et évite d'effectuer la transposée des poids. La matrice de retour est fixée à l'initialisation

2. https://www.cs.toronto.edu/~kriz/cifar.html

du réseau, ce qui permet une implémentation similaire entre les synapses et la matrice de retour pour le produit matriciel avec l'erreur.

Ces méthodes permettent d'avoir un chemin parallèle pour propager l'erreur ainsi que d'éviter la transposée des poids, peu biologiquement plausible, au prix d'une baisse de performances de classification sur les CNNs. De plus, les nouvelles matrices, pour une implémentation physique, doivent être stockées en mémoire ou implémentées avec des *crossbar array* et initialisées pseudo aléatoirement pour avoir les meilleures performances possible, ce qui nécessite de mesurer les poids du réseau et un circuit pour ajuster ces matrices.



(a) Réseau de neurones à quatre couches, l'inférence est représentée en bleu et la propagation de l'erreur en rouge pour *BP*.

(b) Réseau de neurones à quatre couches, l'inférence est représentée en bleu et la propagation de l'erreur en rouge pour *FA*.

(c) Réseau de neurones à quatre couches, l'inférence est représentée en bleu et la propagation de l'erreur en rouge pour *DFA*.

Figure 1.18 – Chemin pour le calcul du gradient d'erreur pour *BP*, *FA* et *DFA* sur un réseau de neurones à quatre couches. *BP* propage l'erreur grâce à la transposée des poids synaptiques, *FA* grâce à une matrice aléatoire B_i et *DFA* grâce au produit matriciel entre l'erreur e et une matrice B_i. En bleu le chemin de l'inférence et en rouge le chemin de la propagation de l'erreur, sur un réseau avec une couche cachée.

1.5.2 Apprentissage grâce à une fonction de coût locale à chaque couche de neurones

Une méthode permettant d'obtenir un gradient d'erreur calculé localement, consiste à mettre en œuvre une fonction de coût au niveau de chaque couche de neurones plutôt qu'uniquement sur la couche de sortie. [Weston et al., 2008, Zhang et al., 2016] proposent d'utiliser des fonctions de coût locales pour augmenter les performances de classification du réseau.

Quand le réseau calcule l'erreur, il est impossible de l'utiliser en même temps pour classifier une donnée. Ce problème est appelé verrouillage des poids. [Jaderberg et al., 2017, Nøkland and Eidnes, 2019] proposent des méthodes d'apprentissage local pour résoudre ce problème de blocage du réseau complet lors de la rétro-propagation du gradient d'erreur. Pour gagner en efficacité et/ou permettre de modifier les couches cachées tout en laissant l'information de classification se propager en avant dans le réseau, ils proposent un apprentissage local, permettant de débloquer les couches indépendamment les unes des autres.

[Jaderberg et al., 2017] proposent une implémentation d'un bloc qui calcule un *gradient synthétique* pour entraîner les couches cachées. Un *gradient synthétique*, $\hat{\delta_A}$, est une estimation approximée du gradient d'erreur de la couche, δ_A. La Figure 1.19 représente deux couches de neurones, f_A et f_B ainsi que le bloc M_B qui calcule le gradient synthétique. Cet entraînement nécessite plusieurs itérations de *BP* pour obtenir le vrai gradient, puis le bloc M_B est entraîné à minimiser les distances entre le vrai gradient et le gradient estimé par le bloc M_B. Le bloc du gradient synthétique est entraîné à prédire le gradient de la couche f_A par rapport à la sortie de f_A, l'état de f_B noté S_B et l'étiquette de l'entrée notée c, soit $\|\delta_A - \hat{\delta_A}\|$. En pratique, le bloc M_B actualise ses propres paramètres pour prédire le gradient à chaque itération grâce au gradient synthétique de la couche suivante et en minimisant la distance avec le gradient estimé. Sur la Figure 1.19, le chemin de l'erreur est en vert et le gradient synthétique en bleu. Cette méthode permet d'atteindre des

performances proche de celle de *BP* avec 81% de reconnaissance sur le jeu de données *CIFAR-10*[2] avec un *CNN* à trois couches contre 82.1% avec *BP* sur le même réseau. En revanche, même si cette méthode évite *BP* après l'entraînement du bloc M_B, *BP* est utilisé pour cet entraînement.

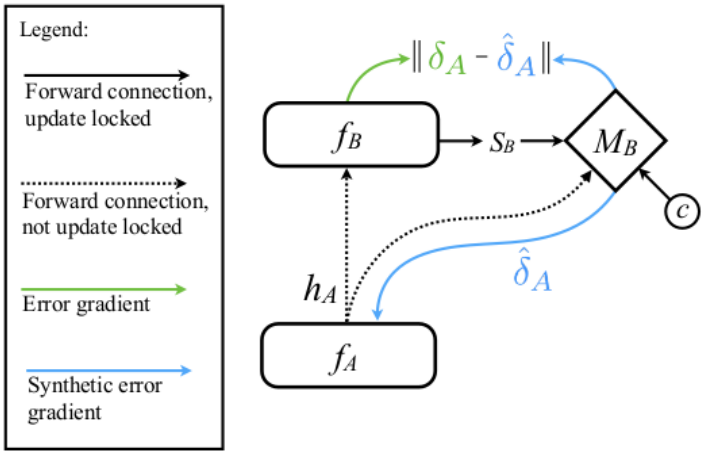


Figure 1.19 – Illustration de la méthode du gradient synthétique. Le bloc M_B est entraîné à estimer le gradient d'erreur de la couche f_A, δ_A par rapport à la sortie de f_A, l'état de B S_B et l'étiquette de l'entrée c. h_A est la sortie de la couche de neurone f_A. Figure extraite de [Jaderberg et al., 2017].

Plutôt que d'entraîner un bloc à estimer le gradient d'erreur de la couche, [Mostafa et al., 2018] rajoute à chaque couche un classifieur linéaire local à la couche. Le gradient d'erreur appliqué au poids synaptique d'une couche i est calculé par rapport à l'erreur du classifieur local de la couche i projeté par une matrice K_i (similaire à *DFA* localement), évitant ainsi une rétro-propagation du gradient d'erreur à travers l'ensemble du réseau. Cette méthode ne permet pas d'obtenir des résultats similaires à une rétropropagation du gradient global sur l'ensemble du réseau sur les réseaux convolutifs avec un taux de reconnaissance d'environ 79% contre 84.4% pour une rétropropagation du gradient global sur l'ensemble du réseau. [Nøkland and Eidnes, 2019] proposent d'étendre cette méthode pour de meilleurs résultats avec les *CNNs* en classification d'image, en rajoutant un calcul de similarité d'image local en plus du classifieur linéaire. En effet, la précision sur *CIFAR-10* pour un réseau *CNN* à 11 couches passe de 94.7%avec leur méthode contre 94.44% avec une rétropropagation du gradient global sur l'ensemble du réseau. En revanche, selon les auteurs, cette méthode devrait donner de moins bons résultats sur des jeux de données plus gros comme *ImageNet*. La Figure 1.20 représente schématiquement une couche convolutive (*Conv*) avec ses fonctions locales pour un réseau de classification d'image (*Linear*). Les flèches noires représentent le sens de l'inférence. Les flèches rouges correspondent à la propagation locale de l'erreur des fonctions de coût local. Pour une couche avec des convolutions, la sortie de la couche est connectée à une couche convolutive qui reconstruit l'image d'entrée et une couche linéaire qui classifie l'image. Une fonction de similarité (*Sim Matching loss*) est calculée entre l'image d'entrée et la sortie. Une fonction *cross entropie* calcule l'erreur entre la sortie du classifieur linéaire et la classe estimée de l'image. Le gradient appliqué à la couche convolutive est la combinaison linéaire de deux fonctions de coût.

Pour une implémentation physique, cette couche supplémentaire nécessiterait de l'espace supplémentaire pour implémenter les neurones et le calcul de la fonction de coût. Le gradient doit être gardé en mémoire, mais comme le calcul est local, une quantité de mémoire moins importante serait utilisée (gradient d'une couche vs tout le réseau).

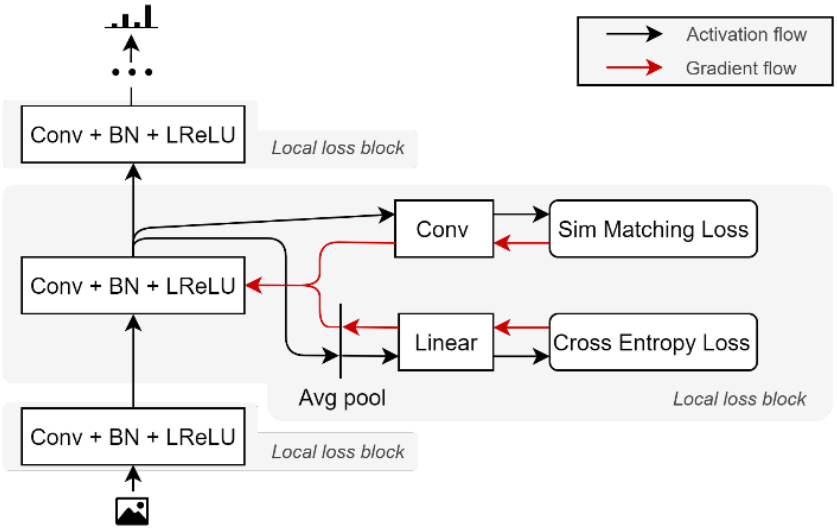


Figure 1.20 – Illustration de l'apprentissage par une couche de classification locale pour des couches convolutives. Les flèches noires représentent le sens de l'inférence. Les flèches rouges correspondent à la propagation locale de l'erreur des fonctions de coût locales. La sortie de la couche est connectée à une couche convolutive qui reconstruit l'image d'entrée et une couche linéaire qui classifie l'image. La valeur du gradient d'erreur de la couche est la combinaison linéaire de l'erreur de classification et de la similarité entre l'image reconstruite et l'image d'entrée. Figure extraite de [Nøkland and Eidnes, 2019].

1.5.3 Apprentissage avec des algorithmes inspiré de la *STDP*

Comme évoqué dans la section 1.5, l'implémentation de *BP* sur des *SNNs* n'est pas triviale. Nous avons vu que les fonctions utilisant la plasticité synaptique avec des informations locales des impulsions des neurones, comme la *STDP*, permettent l'entraînement des *SNNs*. Néanmoins, les performances de la *STDP* avec deux facteurs (les impulsions des neurones connectés) sont inférieurs à *BP* pour des problèmes supervisés. Grâce à l'ajout d'un troisième facteur correspondant à un signal de récompense (ou un signal d'erreur), il est possible de profiter de l'aspect local de l'algorithme de la *STDP* tout en ayant des performances proches de l'état de l'art.

Event driven Backpropagation [Neftci et al., 2017] utilise des neurones *IF* pour dériver la fonction d'activation des neurones et s'inspire de *FA* [Lillicrap et al., 2016] pour calculer une erreur pondérée par une matrice aléatoire. Pour un neurone *IF* avec un temps de réfraction non nul, la fréquence du neurone est compris entre 0 et un $\frac{1}{T_{refract}}$ et augmente proportionnellement au signal d'entrée. Il est donc possible d'approximer la fonction d'activation du neurone par une fonction hard sigmoïde, dont la dérivée vaut 1 entre 0 et $\frac{1}{T_{refract}}$. La synapse est modifiée par l'erreur obtenue par *FA* si le neurone présynaptique a une dérivée non nulle de sa fonction d'activation, donc s'il a émis un nombre d'impulsions compris entre deux valeurs préalablement choisies. Cet algorithme à trois facteurs est local en temps et en espace tout en obtenant des résultats 1% inférieur à ceux de *BP* sur le jeu de classification *MNIST*. De plus, chaque synapse ne nécessite qu'une addition et deux comparaisons pour chaque modification, rendant cet algorithme peu gourmand en opération pour une implémentation sur du matériel physique.

DECOLLE [Kaiser et al., 2020] est également un algorithme basé sur les fréquences des neurones qui utilise une erreur locale, calculée grâce à une matrice de poids, associée à chaque couche, suivie d'une fonction de coût, inspiré de [Mostafa et al., 2018]. Le gradient du poids synaptique est la multiplication du gradient d'erreur local, du substitut du gradient du neurone [Zenke and Ganguli, 2018], ainsi que la trace de la membrane (son évolution au cours du temps) du neurone présynaptique.

La *trace d'éligibilité* (*eligibility trace*) [Gerstner et al., 2018] est une théorie d'inspiration biologique qui permet d'associer un drapeau à une synapse qui permet de lui signaler qu'il doit modifier la valeur de la synapse dans le cas où il recevrait un signal externe, comme un gradient d'erreur. La *trace d'éligibilité* est activée sur une synapse quand les neurones post et présynaptiques émettent une impulsion dans un intervalle

proche. Ainsi, des méthodes d'apprentissage à trois facteurs peuvent être composées d'une *trace d'éligibilité* combinée avec un signal de récompense qui dépend de l'algorithme utilisé, comme suit.

[Detorakis et al., 2018] proposent un *framework* pour une implémentation utilisant une *trace d'éligibilité* pour implémenter la *STDP* optimisée pour le calcul sur du matériel numérique. La différence temporelle des impulsions utilisé par la *STDP* est calculée grâce à une unique valeur temporelle enregistrée par synapse, plutôt que d'enregistrer le moment d'émission de toutes les impulsions. La règle d'apprentissage du *framework* est à trois facteurs où le troisième facteur est une fonction générique à définir qui dépend de l'état du neurone postsynaptique. Par exemple, cette fonction peut-être équivalent à la règle d'apprentissage à trois facteurs comme celle proposée par [Neftci et al., 2017]. Les performances de classification sont légèrement moins bonnes que l'état de l'art sur le jeu de reconnaissance manuscrit *MNIST* avec 4% d'erreur contre moins de 2% pour *BP* pour une architecture similaire.

L'algorithme *e-prop* [Bellec et al., 2020] utilise la *trace d'éligibilité* pour entraîner localement des réseaux de neurones récurrents à impulsions. La valeur de la *trace d'éligibilité* est multipliée avec une erreur locale calculée à chaque pas de temps. L'erreur locale est l'erreur d'un classifieur linéaire, propre à chaque couche de neurones. Cet algorithme permet d'obtenir des performances similaires à *BPTT*, l'état de l'art sur ce type de réseau, sur des *RNNs* tout en étant bio-plausible.

Malgré une localité de l'information, tous ces algorithmes sont à trois facteurs. Ce troisième facteur, qui représente une récompense/erreur, nécessite un circuit supplémentaire pour une implémentation matérielle. De plus, ce facteur est souvent calculé grâce à un classifieur local, ce qui entraîne les mêmes problèmes qu'évoqué dans la section précédente.

1.5.4 Apprentissage avec des modèles basés sur l'énergie

Les modèles basés sur l'énergie sont des modèles de réseaux de neurones dynamiques qui minimisent l'énergie du réseau pour modifier les poids. Ces modèles nécessitent une topologie du réseau qui permet une dynamique grâce à de la récurrence des connexions. Les premiers modèles basés sur l'énergie sont les réseaux de *Hopfield*. Les réseaux de *Hopfield* [Hopfield, 1982], sont des réseaux dynamiques où les connexions entre les neurones ne sont pas en couches successives, mais arbitraires. De plus, les connexions synaptiques sont bidirectionnelles et symétriques tel que que $w_{ij} = w_{ji}$. La Figure 1.21 représente un réseau de cinq neurones entièrement connectés.

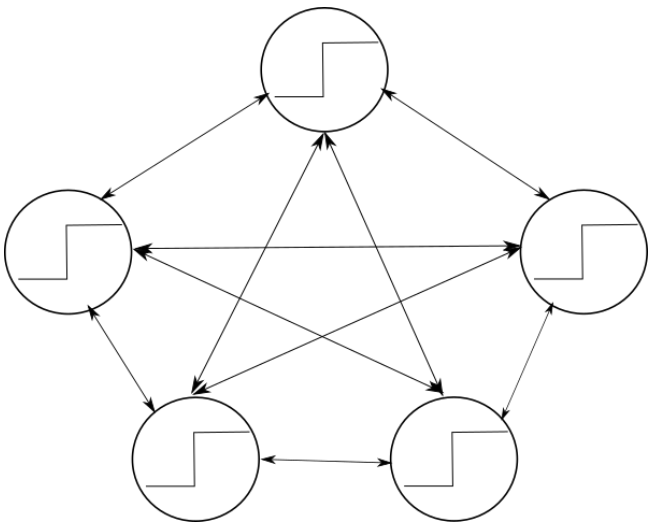


Figure 1.21 – Réseau de *Hopfield* ([Hopfield, 1982]) à 5 neurones entièrement connecté avec une fonction d'activation *Heaviside*.

L'état des neurones évolue au cours du temps. La dynamique de réseau minimise l'énergie E égale à :

$$E = -\frac{1}{2}\sum\sum_{i \neq j} \sigma_i \sigma_j w_{ij}, \tag{1.16}$$

où $\sigma_{i/j}$ est la sortie des neurones i/j. L'évolution de E au cours du temps correspond à :

$$\Delta E = -\Delta\sigma_i \sum_{j \neq i} w_{ij}\sigma_j \tag{1.17}$$

L'apprentissage de ce réseau se fait en minimisant l'énergie du réseau. L'idée est de faire en sorte que chaque classe soit un minimum énergétique (différent) du réseau. Ainsi, pour une entrée x, le réseau convergera vers le minimum énergétique propre à la classe de x. La loi d'apprentissage de ces réseaux est inspirée de la loi d'*Hebb* [Hebb, 1949] où, comme expliqué dans la section 1.5 les modifications du poids synaptique dépendent des activités des neurones connectés en fonction des corrélations des temps d'émissions des impulsions de chacun.

$$w_{ij} = \frac{1}{n} \sum_{k=1}^{p} x_i^k x_j^k \tag{1.18}$$

ou n est le nombre de classes, x_i^k et x_i^k sont respectivement les k-ieme bits d'entrée des neurones i et j, et p la dimension de la donnée d'entrée. Cette loi d'apprentissage est biologiquement possible, mais demande des entrées binaires.

Les réseaux de *Hopfield* ont été généralisés sur des problèmes continus [Movellan, 1991], avec une méthode d'apprentissage appeler *Contrastive Hebbian Learning (CHL)*. L'apprentissage doit minimiser l'énergie totale du système, F, défini par :

$$F = E + S \tag{1.19}$$

avec S une fonction de pénalité qui fait tendre les sorties des neurones vers une valeur centrale de leur fonction d'activation. L'apprentissage est effectué itérativement en deux phases :

- une phase, dite libre, où les entrées du réseau sont imposées par les valeurs de la donnée d'entrée, mais les sorties évoluent selon la dynamique du réseau, comme sur l'équation 1.17,
- et une phase, dite verrouillée, où les entrées du réseau sont imposées et où les sorties du réseau sont imposées aussi à une valeur souhaité.

La différence des valeurs de sortie des neurones σ_i et σ_j connectés à une synapse, entre la phase libre (-) et la phase verrouillée (+), encode le changement de poids tel que :

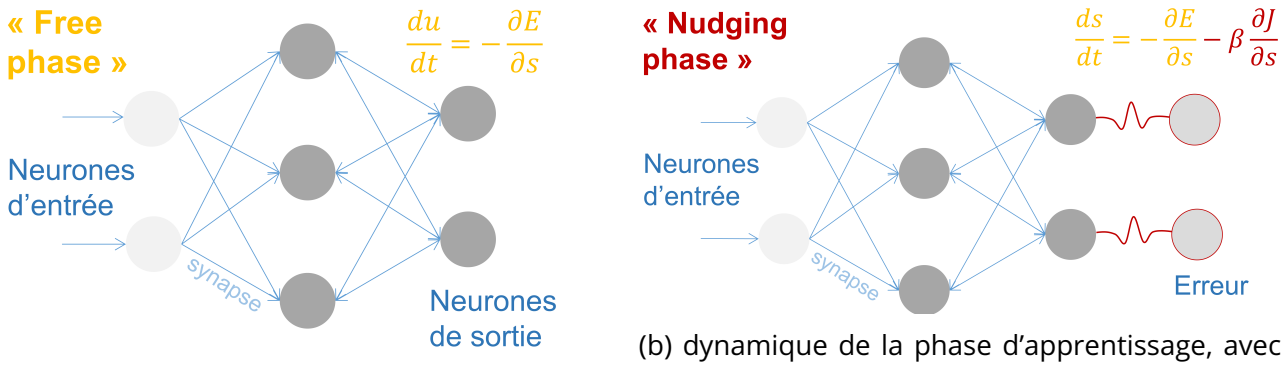
$$\frac{\partial(F(+) - F(-))}{\partial w_{ij}} \propto \Delta w_{ij} \propto \sigma(s_i)(+)\sigma_i(+) - \sigma_i(-)\sigma_i(-) \tag{1.20}$$

Avec la *Contrastive Hebbian Learning*, le gradient des poids est calculé en utilisant la structure du réseau contrairement à *BP* qui a besoin d'un circuit spécial où des calculs différent de ceux effectués par le réseau sont réalisés pour calculer le gradient des poids. En effet, pour entraîner *BP*, il y a une phase d'inférence et une phase de rétro propagation du gradient, or cette deuxième phase ne peut pas être effectuée avec le réseau comme à la phase d'inférence. En revanche, la fonction de coût n'est pas définie explicitement par le réseau, car les sorties ne sont pas verrouillées en fonction d'une erreur, mais de l'étiquette de la donnée d'entrée. De plus, si le minimum énergétique local des deux phases est différent, il est possible que la règle d'apprentissage ne minimise pas l'énergie du modèle (donc n'apprenne pas).

Equilibrium propagation (*Eq-prop*) [Scellier and Bengio, 2016] proposent une méthode d'apprentissage supervisé pour les réseaux de type *Hopfield* qui s'inspire de la *CHL* mais permet de définir une fonction de coût spécifique. De plus, comme démontré sur le plan théorique et numérique, la règle d'apprentissage constitue une approximation des mises à jour de la rétro-propagation du gradient d'erreur dans le temps (BPTT), l'état de l'art pour des réseaux neuronaux récurrents [Ernoult et al., 2019b]. Cela permet à *Equilibrium propagation* d'atteindre une précision inférieure à 1% de la *BPTT* avec des architectures convolutives sur l'ensemble de données *CIFAR-10* [Laborieux et al., 2021].

Equilibrium propagation utilise une formulation basée sur l'état des neurones dynamiques. Pour un réseau de neurones décrit par une fonction d'énergie de type Hopfield :

$$E(s) = \frac{1}{2}\sum_i s_i^2 - \frac{1}{2}\sum_{i \neq j} \sigma(s_i)\sigma(s_j) - \sum_i b_i \sigma(s_i) \tag{1.21}$$



(a) dynamique de la phase d'inférence.

(b) dynamique de la phase d'apprentissage, avec l'erreur en rouge.

Figure 1.22 – Équation de la dynamique des deux phases de *Equilibrium propagation* pour un réseau de neurones avec une couche cachée. Le réseau va converger vers un minimum énergétique en suivant l'équation $\frac{ds}{dt}$ soumise à une entrée continue.

, où s sont les états des neurones et σ leur fonction d'activation. L'énergie totale du système, F, dépend d'une fonction de coût J, telle que :

$$F = E + \beta J \tag{1.22}$$

avec β un réel positif. Comme pour la *CHL*, l'algorithme fonctionne en deux phases pour minimiser F : une phase d'inférence libre et une phase de perturbation où les sorties sont *faiblement verrouillées*.

Dans la phase d'inférence, les entrées du réseau sont statiques pendant toute la phase. La dynamique du réseau est donnée par :

$$\frac{ds}{dt} = -\frac{\partial E}{\partial s} \tag{1.23}$$

Le réseau converge vers un point d'équilibre ($\frac{ds}{dt} = 0$) qui correspond au minimum énergétique du réseau. Pour entraîner le réseau, cet état d'équilibre est ensuite perturbé par une force proportionnelle à la dérivée de la fonction de coût J.

Pendant cette phase de perturbation, l'erreur de prédiction au niveau de la couche de sortie est convertie en un signal agissant sur les neurones de sortie afin de les tirer vers un état souhaité. Contrairement à *CHL*, la valeur de sortie des neurones de sortie n'est pas verrouillée, mais seulement perturbée. Grâce à la bidirectionnalité des synapses, le signal d'erreur induit par cette perturbation se propage au reste du réseau au fil du temps, avec la dynamique du réseau telle que :

$$\frac{ds}{dt} = -\frac{\partial F}{\partial s} = -\frac{\partial E}{\partial s} - \beta\frac{\partial J}{\partial s} \tag{1.24}$$

Cette phase dure jusqu'à ce qu'un second équilibre soit atteint. Pour l'apprentissage, les valeurs synaptiques sont mises à jour selon la règle d'apprentissage similaire à *CHL* :

$$\frac{\partial J}{\partial w_{ij}} \propto \Delta W_{ij} \propto \frac{1}{\beta}(\sigma(s_i)_n\sigma(s_j)_n - \sigma(s_i)_f\sigma(s_j)_f) \tag{1.25}$$

où le produit $\sigma(s_i)\sigma(s_j)$ est mesuré à l'équilibre, à la fin de la phase de perturbation $()_n$ et de la phase d'inférence $()_f$. Cela correspond à une descente de gradient stochastique où $J = \frac{1}{2}\|y^0 - y\|^2$ avec y^0 la valeur de sortie des neurones à la fin de la phase d'inférence et y la valeur à la fin de la phase de perturbation. Théoriquement, le temps d'attente pour atteindre l'équilibre du réseau pour chaque phase peut être long et l'équilibre n'est pas toujours garanti. Pour pallier à ce problème, le temps est généralement fixé sans calculer l'équilibre, et l'hypothèse est faite que le réseau est suffisamment stable une fois ce temps atteint et les fonctions d'activation des neurones sont souvent bornées.

Cette règle d'apprentissage peut être étendue aux cas où les poids sont continuellement mis à jour pendant la deuxième phase [Ernoult et al., 2020] :

$$\Delta w_{ij} \propto \frac{1}{\beta}(\sigma_{t+1}(s_i)\sigma_{t+1}(s_j) - \sigma_t(s_i)\sigma_t(s_j)) \tag{1.26}$$

La somme des modifications pour chaque temps t sera proportionnel à l'Équation 1.25. Cette équation peut se réécrire sous la forme de :

$$\frac{1}{\beta}(\sigma_{t+1}(s_i)\sigma_{t+1}(s_j) - \sigma_t(s_i)\sigma_t(s_j)) \propto \frac{d}{dt}(\sigma(s_i)\sigma(s_j))$$
$$= (\frac{d\sigma(s_i)}{dt}\sigma(s_j) + (\sigma(s_i)\frac{d\sigma(s_j)}{dt})$$

La dynamique du réseau, avec *Eq-prop*, calcule donc directement localement en temps et en espace le gradient de l'erreur, encodée dans la dérivée de la sortie des neurones $\frac{d\sigma(s)}{dt}$ et multipliée par la sortie des neurones connectés $\sigma(s)$.

L'implémentation d'*Eq-prop* suppose des poids symétriques pour la bidirectionnalité. [Scellier et al., 2018] généralise l'algorithme à des champs de vecteurs pour ainsi éviter d'imposer la symétrie des poids.

Même si *Eq-prop* a été développé pour des *ANNs*, les articles [Bengio et al., 2017, Scellier and Bengio, 2016] font un lien entre la *STDP* et *Eq-prop* avec des *SNN*. En effet, quand les neurones communiquent par fréquence, la modification du poids par la *STDP* correspond à la moyenne temporelle des impulsions du neurone postsynaptique dans la fenêtre de la *STDP*. L'application de la loi d'apprentissage en continu d'*Eq-prop* : $\Delta w_{ij} = (\frac{d\sigma(s_i)}{dt}\sigma(s_j) + (\sigma(s_i)\frac{d\sigma(s_j)}{dt})$ est équivalent en moyenne à la *STDP* fréquentielle, l'idée étant que : s'il y a un changement de fréquence d'impulsion du neurone postsynaptique avec une fréquence constante du neurone présynaptique, cela entraînera une augmentation du nombre d'impulsions après l'impulsion présynaptique comparé à avant l'impulsion présynaptique. La Figure 1.23 illustre ce comportement. En effet, pour une fréquence d'impulsion constante du neurone présynaptique, ρ_{pre}, et pour une accélération de la fréquence du neurone postsynaptique, ρ_{post}, le nombre d'impulsions postsynaptiques est de deux avant l'impulsion présynaptique et de 4 après (fenêtre *STDP* en orange).

Le principe des deux phases reste le même, mais la loi d'apprentissage devient :

$$\frac{dW_{ij}}{dt} \propto \sigma(u_i)\frac{d\sigma(u_j)}{dt} + \sigma(u_j)\frac{d\sigma(u_i)}{dt} \tag{1.27}$$

avec $u_{i/j}$ le potentiel de membrane des neurones i/j.

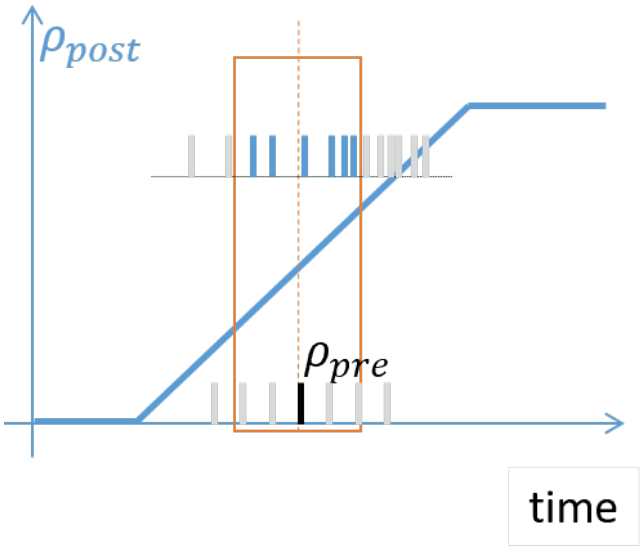


Figure 1.23 – pour une fréquence d'impulsion constante ρ_{pre} et pour une accélération de la fréquence de ρ_{post}, le nombre d'impulsions postsynaptique est de deux avant l'impulsion présynaptique te de 4 après (fenêtre *STDP* en orange) ce qui proportionnel à la loi de *Eq-prop* : $\rho_{pre}\frac{d\rho_{post}}{dt}$. Illustration reproduite de [Bengio et al., 2017].

[O'Connor et al., 2019] proposent une implémentation de *Eq-prop* avec des neurones à impulsions sans *STDP*. Pour pouvoir utiliser les impulsions dans la règle d'apprentissage, ils utilisent des fonctions

d'encodage pour définir le comportement du neurone pour émettre des impulsions et de décodage sur les trains d'impulsion ainsi que l'utilisation d'un recuit (*annealing*, en anglais) pour permettre une convergence vers un point stable du réseau. Malgré la garantie de convergence apporté, le besoin du recuit qui modifie les paramètres des fonctions d'encodages et de décodages rajoute une complexité pour une implémentation physique qui consomme de l'espace et de l'énergie.

[Mesnard et al., 2016] proposent une implémentation pour *SNN*, aussi sans *STPD*, où les neurones sont des neurones *LIF*. En mesurant la fréquence d'impulsion des neurones pour la convertir en une valeur, il est possible de calculer la règle d'apprentissage de *Eq-prop* comme proposé pour des *ANNs*. Cette approche ne tire pas parti des impulsions individuelles comme peut le faire la *STDP* et nécessite de mesurer et d'enregistrer les fréquences des neurones.

Des implémentations de *Eq-prop* sur du matériel analogique ont été proposées dans les articles [Kendall et al., 2020, Zoppo et al., 2020] en obtenant des résultats en simulation sur *MNIST* similaires à l'état de l'art. Les auteurs proposent des memristors pour les synapses. En particulier, [Kendall et al., 2020] utilise des memristors comme synapses et des diodes comme neurones. Les synapses sont connectées sous forme de *cross-bar array* et les poids sont modifiés grâce à des impulsions spécifiques après calcul de la règle d'apprentissage. Le réseau est analogique, ce qui évite l'utilisation de convertisseurs analogique/numérique et inversement, car ils sont coûteux en place et en énergie. Cependant, cette implémentation nécessite de mesurer l'état de chaque neurone ainsi qu'une mémoire externe au réseau pour stocker ces états et le gradient d'erreur, puis un circuit externe pour calculer les impulsions à envoyer pour l'appliquer aux poids synaptiques. Les deux approches profitent de la localité de la règle d'apprentissage pour l'implémentation, de plus elles utilisent les mêmes composants pour les calculs dans les phases d'inférence et d'apprentissage, ce qui est une autre caractéristique hautement souhaitable pour les systèmes neuromorphiques car elle simplifie grandement les circuits. En revanche, cet algorithme nécessite de mesurer les états des neurones du réseau lors de l'exécution, et de les garder en mémoire. Pour ce faire, il est nécessaire de disposer d'un circuit externe au réseau, ce qui augmente la consommation d'énergie et l'espace utilisé. L'apprentissage n'est donc pas intrinsèquement local dans le temps pour ces implémentations.

1.6 Contributions de la thèse

Le développement de puces neuromorphiques avec des memristors pour les synapses est prometteur pour une réduction d'énergie et de temps de calcul des réseaux de neurones. En revanche, l'apprentissage sur puce reste compliqué, car *BP* demande d'embarquer sur la puce des circuits externes au réseau. Pour éviter d'avoir à rétro propager le gradient, des algorithmes avec une règle d'apprentissage locale, comme *Eq-prop* ou la *STDP*, peuvent être utilisés. De plus, il est possible de tirer parti de la mécanique de la modification des memristors avec des impulsions pour profiter des impulsions des *SNNs*. Des implémentations de la *STDP* ont été proposées pour ainsi permettre un apprentissage intrinsèque, c'est-à-dire qui calcule localement les gradients des poids et qui applique ce gradient directement à la synapse grâce aux impulsions. En revanche, ces implémentations obtiennent des performances sur des problèmes de classification inférieures à *BP*. Malheureusement, il n'existe pas encore d'algorithme avec une règle d'apprentissage locale et intrinsèque à la synapse. En permettant un apprentissage intrinsèque tout en ayant des résultats similaire à l'état de l'art grâce à des nanocomposants émergents et un algorithme adapté, il serait possible d'avoir des circuits neuromorphiques qui consomment peu d'énergie et comprenant une forte densité de neurones et de synapses, tout en évitant une partie des circuits périphériques au réseau.

Dans cette thèse, je regarde des solutions pour obtenir un apprentissage intrinsèque pour un réseau avec des neurones à impulsions et des memristors pour synapse.

Dans le chapitre 2, nous proposons un algorithme, *EqSpike*, inspiré de *Equilibrium propagation* pour un apprentissage intrinsèque. Nous expliquerons pourquoi et comment nous avons converti *Eq-prop* pour des neurones à impulsions puis comment nous avons rendu la modification des poids synaptiques possible grâce aux impulsions de la synapse. Une étude théorique sur les performances énergétiques et la vitesse de calcul avec cet algorithme a été effectuée, prédisant une réduction de 2 ordres de grandeurs de l'énergie pour l'apprentissage et de 3 ordres de grandeurs pour l'inférence comparé à des *GPUs*. De plus, la mise en évidence d'un lien entre la *STDP* et *EqSpike* a été effectué. Les travaux présentés dans ce chapitre ont mené à une publication dans un journal [Martin et al., 2021], une présentation dans la conférence *NAISys*

2020 ainsi qu'un poster pour la conférence *COSYNE 2021*.

Dans le chapitre 3, une implémentation d'*EqSpike* avec des memristors pour synapses en simulation est proposée. L'implémentation utilise le comportement des memristors pour obtenir un apprentissage intrinsèque. La simulation de différentes variations inter-composants montre que la précision de classification est d'autant plus élevée que le memristor présente des variations faibles de ses pentes conductance vs. tension, ainsi que de son seuil. Le circuit proposé fait l'objet du dépôt de brevet *Thales* [*FR2101311 + PCT/EP2022/053026*].

Pour finir, le chapitre 4 présente une implémentation alternative d'*EqSpike* grâce exploitant l'accumulation des impulsions des neurones comme dans les implémentations memristives de la *STDP*, pour des circuits potentiellement plus compacts. Malgré une performance plus faible que *EqSpike* sur la version de base, différentes variantes de cet algorithme sont proposés en fonction des spécificités de l'algorithme et de la complexité des implémentations physiques souhaité, comme de la synchronisation des neurones ou leur désynchronisation. Les variantes les plus performantes permettent d'obtenir théoriquement les mêmes résultats que l'état de l'art.

Les travaux effectué dans cette thèse ont mené aux contributions suivantes :

— une publication dans un journal [Martin et al., 2021] sur l'algorithme *EqSpike*,
— une présentation à la conférence *NAISys 2020* sur l'algorithme *EqSpike*,
— un poster à la conférence *COSYNE 2021*) sur l'algorithme *EqSpike*,
— un brevet déposé [*FR2101311 + PCT/EP2022/053026*] et article en préparation sur l'adaptation de *EqSpike* pour un circuit avec des memristor comme synapses.
— Grâce à mon expertise en ML, j'ai également pu contribuer à des travaux de spintronique qui ne sont pas couverts dans cette thèse. Je suis notamment co-auteur d'un article de journal proposant des méthodes pour réaliser les opérations de multiplication et d'accumulation novatrices avec des synapses implémentées par des résonateurs spintroniques [Leroux et al., 2021].

Chapitre 2

EqSpike : un algorithme pour un apprentissage local et intrinsèque pour des réseaux de neurones à impulsions

Dans ce chapitre, un nouvel algorithme sera présenté, appelé *EqSpike*, basé sur *Equilibrium-Propagation* (*Eq-Prop*). Les travaux de ce chapitre ont fait l'objet d'une publication [Martin et al., 2021] et des présentations aux conférences multi-disciplinaires *NAISys 2020* et *COSYNE 2021*.

La première contribution de cette thèse permet d'obtenir un apprentissage intrinsèque, pour être adapté au matériel à faible consommation d'énergie et avec une forte bio-plausibilité. Ces objectifs seront réalisés grâce aux neurones à impulsions et au rajout d'un bloc par neurone permettant de calculer le gradient grâce à la dynamique du réseau et de modifier les synapses grâce aux impulsions des neurones. Le calcul local du gradient, avec son application aux synapses, est plus avantageux en espace et en énergie, comparé à la mesure de l'état de chaque neurone et au calcul du gradient sur un circuit externe. De plus, le nombre de neurones étant significativement moins grand que le nombre de synapses, il est préférable de placer la complexité d'implémentation des composants dans les neurones plutôt que dans les synapses.

Le but de cette thèse est de trouver un modèle d'apprentissage pour des réseaux de neurones utilisant des nanocomposants. À cette échelle, le coût de transfert de données est important si les composants sont éloignés. L'approche proposée pour *EqSpike* est de réduire au maximum le transfert et le stockage d'information pour l'apprentissage sur des circuits externes au réseau. Un apprentissage local en espace pour le calcul des gradients d'erreur permet d'éviter de transférer l'information sur un circuit externe, et permet donc d'économiser cette dépense énergétique. De plus, un apprentissage local en temps permet d'éviter d'enregistrer l'information de modification de la synapse, ce qui évite ce surcoût de consommation d'énergie.

2.1 Introduction

Embarquer l'apprentissage dans une puce neuromorphique requiert l'utilisation de circuits externes pour optimiser les poids avec l'algorithme de la rétro-propagation du gradient. Si l'on veut un réseau de neurones capable d'apprendre sur puce sans circuit additionnel, le réseau doit calculer ses propres gradients d'erreur et les appliquer grâce aux signaux des neurones. L'apprentissage doit être local en espace et en temps pour éviter des transferts d'information ou son stockage. Nous appelons cette forme d'apprentissage : *l'apprentissage intrinsèque*.

Eq-Prop (détaillé au chapitre précèdent 1.5.4) permet de calculer localement le gradient d'erreur grâce à la dynamique du réseau. Il fonctionne en deux phases consécutives. La première phase, la phase d'inférence, reçoit en entrée les données à classer jusqu'à ce que le réseau atteigne un équilibre. Pour la deuxième phase, la phase d'apprentissage, le réseau reçoit, en plus des entrées, une perturbation de ses sorties avec une force proportionnelle à l'erreur du réseau jusqu'à un nouvel équilibre. La différence entre les deux équilibres du réseau donne le gradient d'erreur. Le réseau évolue donc dans le temps et ce changement, avec les entrées et les sorties du réseau influencées par l'algorithme, encode le gradient d'erreur.

Eq-Prop obtient des gradients similaires à BPTT, l'état de l'art pour les réseaux de neurones dynamiques [Ernoult et al., 2019b]. *Eq-Prop* est local dans l'espace et présente des avantages clés pour les implémentations neuromorphiques [Kendall et al., 2020, Zoppo et al., 2020, Dillavou et al., 2022]. Contrairement à la rétro-propagation du gradient, *Eq-Prop* utilise les mêmes composants pour les calculs dans les phases d'inférence et d'apprentissage, ce qui est une autre caractéristique hautement souhaitable pour les systèmes neuromorphiques car elle simplifie grandement les circuits.

En revanche, l'algorithme nécessite de mesurer les états des neurones du réseau lors de l'exécution, et de les garder en mémoire. Pour ce faire, il est nécessaire d'avoir un circuit externe au réseau, ce qui augmente la consommation d'énergie et l'espace utilisé. L'apprentissage n'est donc pas intrinsèquement local dans le temps. De plus, le temps d'attente pour atteindre l'équilibre du réseau pour chaque phase peut être long et n'est pas toujours garanti. Pour palier à ce problème, le temps est souvent fixé sans calculer l'équilibre, et on fait l'hypothèse que le réseau est suffisamment stable et les fonctions d'activation des neurones sont souvent bornées.

Les réseaux de neurones à impulsions sont capables d'atteindre une faible consommation d'énergie sur des puces neuromorphiques [Merolla et al., 2014]. Des algorithmes de type local en espace et en temps, comme la *STDP*, peuvent être utilisés pour entraîner les réseaux de neurones à impulsions sur des circuits compacts (voir section 1.4.3). Cet algorithme est non supervisé et n'arrive pas à atteindre les performances de la rétro-propagation du gradient [Falez et al., 2019].

Eq-Prop fonctionne avec des neurones sans impulsions, mais une possible implémentation utilisant les impulsions est suggérée dans l'article original. Cependant, la fréquence des impulsions doit être mesurée et la modification de la synapse ne se fait pas automatiquement. L'apprentissage ne sera donc pas intrinsèque. Pour rendre *Eq-Prop* intrinsèque, cet algorithme sera adapté aux réseaux de neurones à impulsions pour calculer la règle d'apprentissage de *Eq-Prop* et appliquer la valeur calculée aux synapses connectées de manière intrinsèque grâce à la dynamique du réseau et aux impulsions des neurones.

Dans un premier temps, il faut adapter la loi d'apprentissage d'*Eq-Prop* au réseau de neurones à impulsions puis développer un algorithme capable de faire cet apprentissage en se basant sur l'algorithme précédemment développé. La thèse se situe dans un cadre d'apprentissage supervisé, l'algorithme développé sera testé sur un jeu de données d'images représentant des chiffres manuscrits (MNIST [LeCun et al., 2010]).

2.2 Adaptation d'*Equilibrium Propagation* dans un réseau de neurone à impulsions

Equilibrium Propagation permet de calculer le gradient d'erreur localement dans l'espace grâce à la différence entre la sortie des neurones à l'équilibre de la phase d'inférence et à l'équilibre de la phase d'apprentissage, selon la règle d'apprentissage rappelée ici :

$$\Delta W_{ij} \sim (\sigma_i)_n(\sigma_j)_n - (\sigma_i)_f(\sigma_j)_f, \tag{2.1}$$

où W_{ij} représente le poids synaptique entre les neurones i et j, σ est la sortie du neurone, c.-à-d. la valeur de la fonction d'activation. Le produit $\sigma_i\sigma_j$ est mesuré à l'équilibre, à la fin des phases d'apprentissage, n (nudge), et respectivement d'inférence, f (free) [Scellier and Bengio, 2016].

Cette règle peut être étendue au cas où les poids sont continuellement mis à jour pendant la phase d'apprentissage [Ernoult et al., 2020], l'algorithme est appelé *Continual Equilibrium Propagation* :

$$\frac{dW_{ij}}{dt} \sim \dot{\sigma_j}\sigma_i + \dot{\sigma_i}\sigma_j, \tag{2.2}$$

où W_{ij} est le poids synaptique connectant deux neurones i et j, et σ_i, σ_j sont les sorties des deux neurones. Pendant la phase d'apprentissage, les poids synaptiques évoluent à chaque itération en fonction des activités des neurones connectés à la synapse. Avec cet algorithme, l'apprentissage devient local en temps et en espace.

Continual Equilibrium Propagation fonctionne avec des neurones formels, à fonction d'activation continue. Le neurone possède une valeur interne, appelé état, qui est la somme de toutes ses entrées pondérées par les synapses connectées, et qui se stabilise pendant les différentes phases. Le calcul de la sortie du neurone se fait en appliquant la fonction d'activation du neurone sur cet état.

Or, la réponse en fréquence d'un neurone à impulsions, par rapport à ses entrées, peut être considéré comme équivalente à sa fonction d'activation. De cette façon σ, la sortie du neurone dans *Continual Equilibrium Propagation*, devient la fréquence d'impulsion du neurone pour des neurones à impulsions et qui sera notée ρ. Pour l'adaptation de *Continual Equilibrium Propagation* à des neurones à impulsions, il faut pouvoir avoir un modèle capable de générer un train d'impulsions avec une fréquence en fonction de l'entrée du neurone. Et cette fonction d'activation doit être proche des fonctions d'activation utilisées pour *Continual Equilibrium Propagation*.

Fonction d'activation des neurones *LIF*

Plusieurs fonctions d'activation existent pour les réseaux de neurones, telles que présentées dans le chapitre de l'état de l'art. Dans un réseau de neurones formel, la fonction d'activation des neurones doit être non linéaire, monotone et dérivable par parties. Parmi elles, l'une des fonctions d'activation les plus utilisées est la *Rectified Linear Unit* (ReLU) donnée par :

$$f(x) = \begin{cases} x, & \text{si } x > 0 \\ 0, & \text{sinon} \end{cases}. \tag{2.3}$$

Cette fonction est souvent utilisée pour sa faible complexité de calcul et celle de sa dérivée. Mais elle présente le désavantage, pour les SNN et les réseaux de neurones physiques, de ne pas être bornée.

En effet, *Eq-Prop* doit se stabiliser pendant ses phases. Si les fonctions d'activations utilisées sont non bornées, il est possible que le réseau ne converge pas vers un état stable. Par exemple, si la fonction utilisée est *ReLu*, un neurone peut augmenter la fréquence d'un neurone connecté par une synapse avec un poids élevé, c'est-à-dire supérieur à 1. Comme le réseau est bidirectionnel, les neurones vont augmenter mutuellement leurs sorties à chaque pas de temps, et donc diverger.

En particulier, dans *Eq-Prop*, la fonction d'activation est :

$$f(x) = \begin{cases} x, & \text{si } x > 0 \text{ et } x < 1. \\ 1, & \text{si } x > 1. \\ 0, & \text{sinon} \end{cases} \tag{2.4}$$

Cette fonction est appelée *hard sigmoid*. Un neurone à impulsions a une fréquence maximale donnée par son temps de réfraction, $T_{refract}$, sa fréquence est donc bornée à une fréquence entre 0 et $1/T_{refract}$.

Choix du modèle des neurones à impulsion

Nous allons présenter les choix effectués pour le modèle de neurone à impulsions et ses paramètres, en vue de l'implémentation d'*Eq-Prop* à impulsions. Il a été choisi d'utiliser le modèle de neurone *LIF*, car c'est un modèle très utilisé dans la littérature neuromorphique et simple à simuler (chapitre 1.3). Toutefois, le potentiel de membrane, u, ne sera pas remis à u_{rest} lors d'une émission d'impulsion, mais sera seulement réduit de la valeur du seuil d'émission θ. Un des défauts des neurones *LIF* est l'oubli des signaux reçus précédemment lors de l'émission d'une impulsion. Ne pas remettre à u_{rest} le potentiel de membrane, mais soustraire θ à la place, permet de garder une trace des anciennes impulsions reçues par le neurone (*reset-by-substraction* [Rueckauer et al., 2017]).

L'algorithme développé, *EqSpike*, doit pouvoir s'adapter à différents paramètres des composants et à différents types de neurones. Nous avons fait le choix d'avoir une dynamique des neurones la plus agnostique possible aux paramètres physiques pour être le plus indépendant possible aux ordres de grandeur des paramètres internes des neurones, comme la résistance ou la conductance dans le cas du neurone *LIF*. Les neurones sont considérés comme parfaits et sans bruit. L'équation des neurones *LIF* utilisés est :

$$\frac{du_i}{dt} = -u_i\gamma_{LIF} + S_i(t) \tag{2.5}$$

avec γ_{LIF} le facteur de fuite du neurone et $S_i(t)$ les impulsions d'entrées au temps t.

La Figure 2.1 correspond à la réponse en fréquence, ρ du neurone en fonction d'une stimulation par un signal continu, I, constant. L'estimation de la fréquence a été réalisée sur une fenêtre temporelle de $100dt$ et nous avons $T_{refract} = 2dt$, avec dt la durée d'un pas de temps lors de la simulation. Le seuil du neurone a été mis à 1 et $\gamma_{LIF} = 0.001$ pour approcher une *hard-sigmoïd*. La fonction est bornée par le temps de réfraction du neurone, $f_{max} = 1/T_{refract}$. Par la suite, le temps de réfraction sera notre valeur étalon pour toutes les autres unités temporelles. Cela permet de traiter des neurones avec une fréquence maximale rapide comme lente sans impacter l'algorithme.

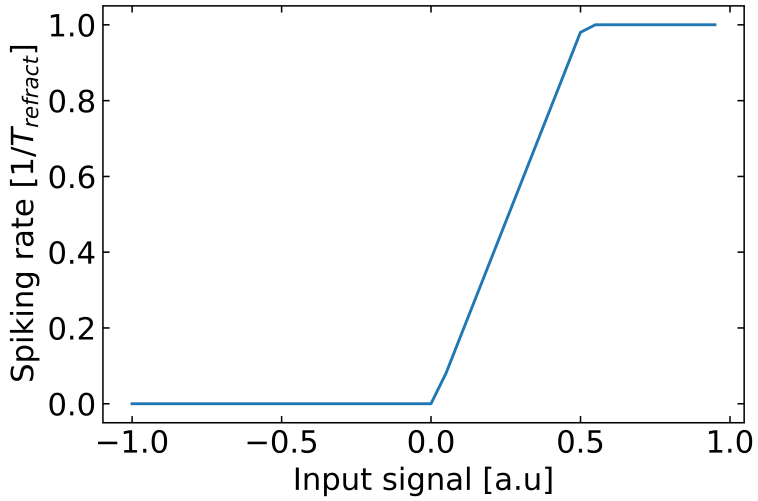


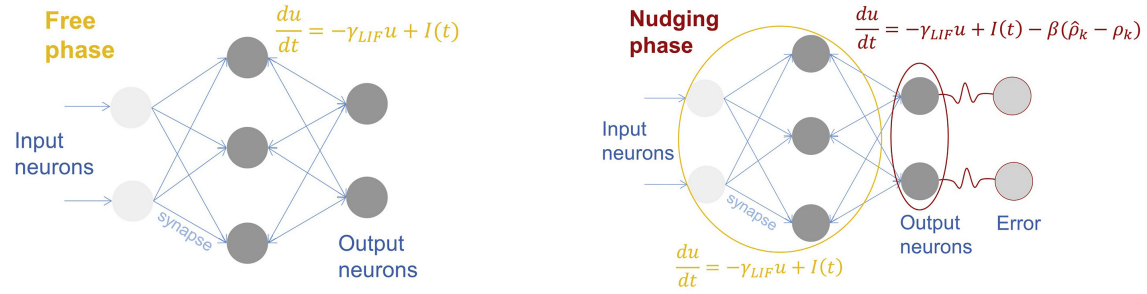
Figure 2.1 – Fonction d'activation des neurones *LIF* : Réponse en fréquence, ρ, du neurone à impulsions *LIF* stimulé par un signal continu I. Fréquence calculée en simulation sur 100 pas de temps et $\gamma_{LIF} = 0.001$.

2.2.1 *Equilibrium Propagation* avec des neurones à impulsion

Il faut maintenant construire la règle d'apprentissage avec ce modèle des neurones *LIF* et leur fonction d'activation présentée plus tôt. Cette règle d'apprentissage se base sur *Continual Equilibrium Propagation*. La dérivée de l'erreur est calculée directement grâce à l'évolution de la dynamique du réseau. La modification des poids dépend de la dérivée de la fonction d'activation d'un neurone $\dot{\rho}$ et de la fréquence de l'autre neurone connecté ρ, et réciproquement. Elle est représentée par l'équation 2.2, adaptée aux neurones à impulsions comme suit :

$$\frac{dW_{ij}}{dt} \sim \dot{\rho}_j\rho_i + \dot{\rho}_i\rho_j \tag{2.6}$$

La formulation de cette règle d'apprentissage basée sur la fréquence dans un réseau de neurones à impulsions est donc la suivante : "chaque fois que le neurone i émet une impulsion, le poids doit être mis à jour par une quantité proportionnelle à la dérivée de la fréquence du neurone j, $\dot{\rho}_j$ (premier terme de l'équation), et réciproquement" [Scellier and Bengio, 2016].



(a) dynamique de la phase d'inférence.

(b) dynamique de la phase d'apprentissage, avec l'erreur en rouge.

Figure 2.2 – Dynamique d'*EqSpike* : Équation de la dynamique des deux phases pour un réseau de neurones avec une couche cachée. Le réseau va converger vers un minimum énergétique en suivant cette équation soumise à une entrée continue.

La nouvelle équation de la dynamique du réseau avec des neurones à impulsions, avec les neurones *LIF*, est représentée sur la Figure 2.2a pour la phase d'inférence et Figure 2.2b pour la phase d'apprentissage. Pendant la phase d'apprentissage, la dynamique du réseau ne dépend que des entrées et des neurones qui suivent l'équation 2.5 (en jaune dans les figures). Lors de la deuxième phase, les neurones de sortie ont une dynamique qui dépend aussi de la force de perturbation proportionnelle à l'erreur (en rouge), avec $\hat{\rho}_k$ la prédiction et ρ_k la cible (l'étiquette de la donnée d'entrée).

Nous avons ainsi développé l'Algorithme 1.

La première boucle correspond à la phase d'inférence. Chaque neurone intègre ses entrées et émet une impulsion si le seuil est dépassé. La deuxième boucle correspond à la phase d'apprentissage avec la perturbation du signal par le gradient d'erreur (par rapport à la fonction de coût) ∇e_o, la force de perturbation β et le même comportement pour les neurones que dans la première phase. Dans la deuxième phase, les synapses sont modifiées par la loi d'apprentissage.

Mais pour appliquer la loi d'apprentissage 2.6, il est nécessaire de mesurer et de calculer pour chaque neurone la fréquence et sa dérivée. Pour l'instant, cet algorithme ne montre pas comment les calculer localement ni comment appliquer la modification au poids synaptique. Pour obtenir un apprentissage intrinsèque, il faut pourtant pouvoir calculer $\dot{\rho}$ localement et modifier, grâce aux impulsions des neurones, le poids synaptique en accord avec la loi d'apprentissage.

Data : input image, Network, Loss function, Length Free phase T_{free}, Length Nudging phase T_{nudge}, Parameters $\gamma_{LIF}, u_{th}, \beta, W_{ij}$
Result : Trained weights for input image : W_{ij} and go to next image/next epoch
for $t < T_{free}$ **do**
  for *each neuron* j **do**
    Update membrane potential $u_j(\gamma_{LIF}, I_j)$
    if $u_j > u_{th}$ **then**
      Emit a spike (t_j)
      $u_j \leftarrow u_j - u_{th}$
    end
  end
end

for $t \in [T_{free}, T_{free} + T_{nudge}]$ **do**
  for *each output neuron* o **do**
    Compute error gradient ∇e_o
    Nudge neuron : $u_o \leftarrow u_o - \beta \cdot \nabla e_o$
  end
  for *each neuron* k **do**
    Update $u_k(\gamma_{LIF}, I_k)$
    if $u_k > u_{th}$ **then**
      Emit a spike (t_k)
      $u_k \leftarrow u_k - u_{th}$
    end
  end
  for *each synapse* w_{ij} **do**
    if *neuron j emits a spike* **then**
      $w_{ij} \leftarrow w_{ij} + \dot{\rho}_i$
    end
    if *neuron i emits a spike* **then**
      $w_{ij} \leftarrow w_{ij} + \dot{\rho}_j$
    end
  end
end

Algorithme 1 : Algorithme *Continual Equilibrium Propagation* avec des neurones à impulsions

2.3 L'algorithme *EqSpike*

Pour appliquer intrinsèquement à la synapse w_{ij} la loi d'apprentissage *Continual Equilibrium Propagation*, il faut pouvoir calculer la fréquence des neurones connectés et leur dérivée. Pour un circuit neuromorphique ceci nécessite des capteurs et des mémoires. Pour la suite, nous proposons une implémentation intrinsèque pour calculer ces paramètres et modifier le poids synaptique.

2.3.1 Calcul local des paramètres de la loi d'apprentissage et application du gradient d'erreur à la synapse

Pour appliquer la loi d'apprentissage, il faut pouvoir calculer la dérivée de la fréquence et l'appliquer au poids synaptique pour chaque impulsion du neurone connecté. Le but de la thèse est d'être le plus compatible possible avec des composants physiques. Il faut utiliser un minimum de composants pour rester dans l'objectif de ces travaux. De plus, les éléments rajoutés doivent pouvoir être compatibles CMOS. Comme il y a beaucoup moins de neurones que de synapses il est préférable de placer la complexité dans les neurones. La solution choisie est de mettre en œuvre un bloc par neurone pour calculer la dérivée de la fréquence et un dispositif par neurone, ici un interrupteur, pour modifier la synapse.

La Figure 2.3 représente, sous la forme d'un schéma, deux neurones à impulsions reliés par une synapse. Notre algorithme étant local et intrinsèque, il est donc possible de représenter seulement deux neurones et la synapse. Pour un réseau complet, il suffit de reproduire ce schéma pour chaque neurone et synapse. Le schéma comprend des éléments de détection d'impulsions à la sortie de chaque neurone, ainsi que des blocs dédiés qui approximent la dérivée de la fréquence des trains d'impulsions de chaque neurone en temps réel, afin de mettre à jour les synapses en conséquence. Lors d'une impulsion d'un neurone, le détecteur d'impulsions permet de modifier la synapse d'une valeur proportionnelle à l'estimation de $\dot{\rho}$. Sur le schéma, le détecteur active la modification de la synapse grâce à un interrupteur, mais le mécanisme peut être adapté en fonction de la méthode pour modifier le poids synaptique du composant utilisé pour la synapse.

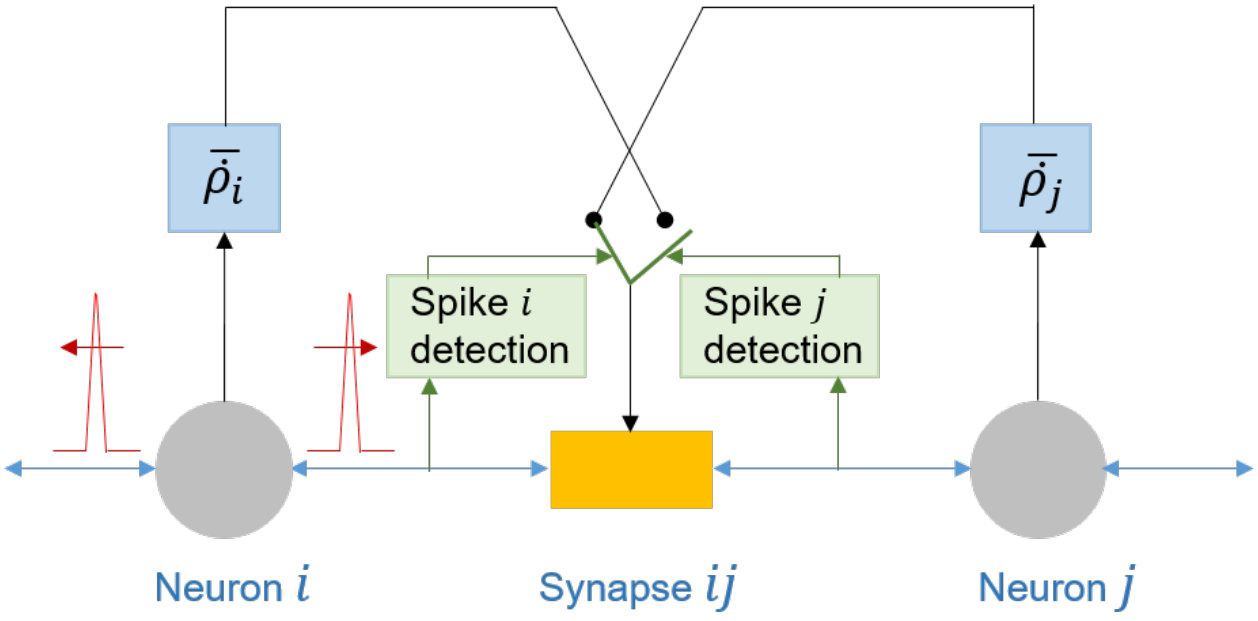


Figure 2.3 – Implémentation locale d'*EqSpike*. Le détecteur d'impulsions permet de modifier le poids synaptique à chaque impulsion du neurone connecté, par une valeur proportionnelle à la dérivée de la fréquence.

Pour extraire la dérivée de la fréquence, il faut d'abord extraire la fréquence des neurones puis soustraire à cette fréquence une fréquence extraite antérieurement. Pour extraire la dérivée de la fréquence $\dot{\rho}$ pour chaque neurone, nous proposons comme implémentation le bloc $\overline{\dot{\rho}}$, illustré dans la Figure 2.4.

Le bloc $\overline{\dot{\rho}}$ est constitué d'un intégrateur, noté *LI*, avec un facteur de fuite γ_{LI}, d'un retardateur temporel, noté *delay*, d'un système de soustraction et d'un filtre passe-bas.

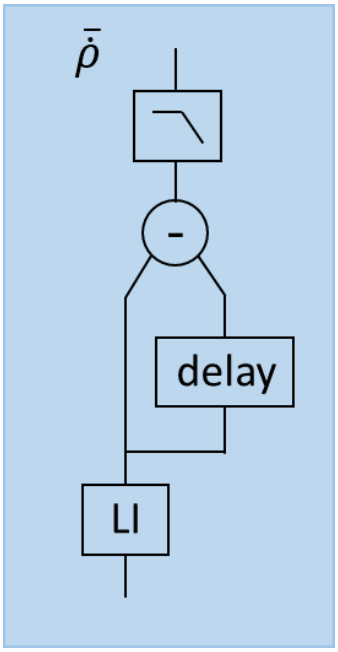


Figure 2.4 – Bloc $\bar{\dot{\rho}}$, qui sert à calculer la dérivée de la fréquence des neurones. Il contient un intégrateur avec une fuite, suivi d'un retardateur et un mécanisme de soustraction. La sortie de la soustraction est ensuite filtrée par un filtre passe-bas.

L'intégrateur intègre le train d'impulsions émis par le neurone auquel il est connecté, tel que sa valeur V_{LI} est :

$$\frac{V_{LI}}{dt} = -V_{LI} * \gamma_{LI} + S(t) \tag{2.7}$$

avec $S(t)$ l'entrée de l'intégrateur (les impulsions émises par le neurone) à un temps t. Grâce à la fuite de l'intégrateur et s'il est soumis à un signal périodique, l'intégrateur convergera vers une valeur d'équilibre, $\frac{V_{LI}}{dt} = 0$, quand le signal d'entrée est égal à la fuite de l'intégrateur, tel que :

$$V_{LI} * \gamma_{LI} = \overline{S(t)} \tag{2.8}$$

avec $\overline{S(t)}$ la moyenne des entrées dans le temps. La valeur de l'intégrateur avec fuite est proportionnelle à la fréquence du neurone [Navarro et al., 2020, Gerstner et al., 2014] :

$$V_{LI} \sim \frac{\rho}{\gamma_{LI}} \tag{2.9}$$

Démonstration. Posons S(t) un train d'impulsions d'amplitude 1 et de période T, et t le temps avec t=kT+q

$$V_{LI}(t) = (1 - \gamma_{LI})V_{LI}(t-1) + S(t) = LV_{LI}(t-1) + S(t)$$

Avec $L = (1 - \gamma_{LI})$ et en partant du premier terme $V_{LI}(0)$, on a :

$$V_{LI}(t) = L^t V_{LI}(0) + \sum_{j=0}^{t} I(j) L^{t-j}$$

I(j) est égal à 1 tous les T pas de temps donc on peut écrire :

$$V_{LI}(t) = L^t V_{LI}(0) + \sum_{n=0}^{k} L^{q+nT}$$

En remplaçant le dernier terme par la somme d'une suite géométrique, on obtient :

$$V_{LI}(t) = L^t V_{LI}(0) + \sum_{j=0}^{k} I(j) L^{q+jT} = L^t V_{LI}(0) + \frac{L^q - L^{t+1}}{1 - L^T}$$

Ce qui peut se simplifier par :

$$V_{LI}(t) = L^t(V_{LI}(0) - \frac{L}{1-L^T}) + \frac{L^q}{1-L^T}$$

ce qui donne vers l'infini :

$$\lim_{t\to\infty} V_{LI}(t) = \frac{L^q}{1-L^T}$$

L'intégrateur oscille donc sur un nombre de T-1 valeurs. La moyenne de l'intégrateur vers l'infini est donc :

$$\frac{1}{T\cdot(1-L^T)}\sum_{n=0}^{T-1} L^n$$

En remplaçant encore le dernier terme par la somme d'une suite géométrique et avec $\rho = \frac{1}{T}$, on obtient :

$$\frac{\rho}{(1-L^T)}\frac{1-L^T}{1-L} = \frac{\rho}{\gamma_{LI}}$$

□

Pour approcher la dérivée, nous retardons ce signal d'une durée τ et soustrayons la valeur réelle à la valeur retardée :

$$V_{delay} = V_{LI}(t) - V_{LI}(t-\tau) \cong \tau\frac{\delta V_{LI}}{\delta t} \propto \frac{\tau}{\gamma_{LI}}\dot{\rho} \tag{2.10}$$

Comme démontré au-dessus, la valeur de l'intégrateur $\frac{\rho}{\gamma_{LI}}$ est une valeur moyenne. La Figure 2.5a correspond à 4 fréquences d'impulsions de neurones différents où les impulsions sont envoyées au bloc $\overline{\dot{\rho}}$ de leur neurone respectif. La Figure 2.5b correspond à la valeur de la dérivée de la fréquence, calculée par le bloc $\overline{\dot{\rho}}$ sans filtre, pour les exemples de fréquence de la Figure 2.5a. La valeur de la sortie attendue du bloc $\overline{\dot{\rho}}$ est positive pendant l'accélération des fréquences, négative pendant la décélération des fréquences et autour de 0 avec une fréquence constante. La Figure 2.5b représente la valeur de l'approximation de la dérivée par l'intégrateur et les retardateurs, comme expliqué au-dessus, pour différentes fréquences et changements de fréquence comme montré dans la Figure 2.5c. La sortie du bloc $\overline{\dot{\rho}}$ est bruitée à cause des oscillations autour de la moyenne $\frac{\rho}{\gamma_{LI}}$ de V_{LI}.

Pour une meilleure approximation de la loi d'apprentissage, il est préférable de lisser la sortie du bloc grâce à un filtre pour obtenir la valeur moyenne de l'intégrateur. Le filtre choisi est un filtre passe-bas avec une simple fenêtre glissante. Il est possible de placer le filtre entre la sortie du neurone et l'entrée de l'intégrateur pour filtrer le signal du neurone 2.5d, ou de le placer à la sortie du bloc, comme illustré par la Figure 2.5c. Comme montré sur la Figure 2.4, nous avons choisi de placer le filtre à la sortie du bloc et non à l'entrée, de manière arbitraire, car les deux choix sont équivalents.

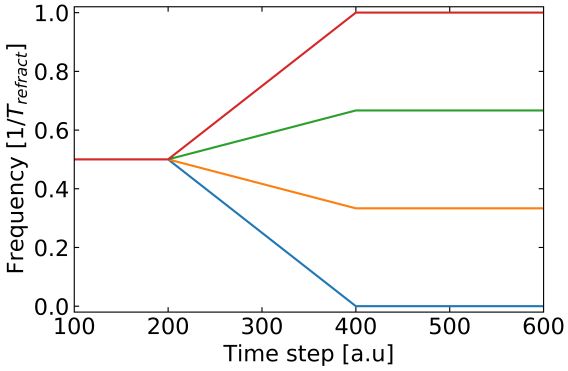
Le filtre est simulé en utilisant une moyenne sur N_{filt} étapes de la simulation :

$$\overline{x(t)} = \frac{1}{N_{filt}}\sum_{i=0}^{N_{filt}-1} x(t-idt), \tag{2.11}$$

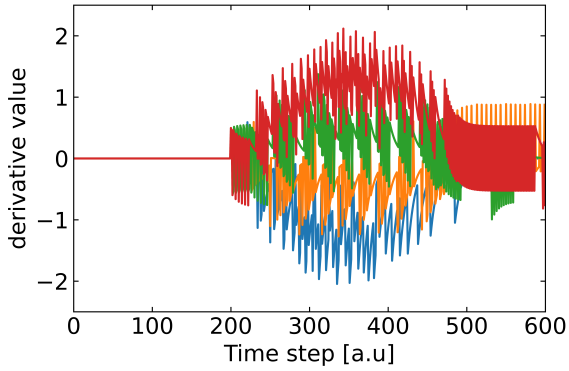
où dt est le pas de temps de simulation. La sortie du filtre, qui se rapproche de $\frac{\tau}{\gamma_{LI}}\overline{\dot{\rho}}$, est ensuite multipliée par le coefficient η_{lr}. Les mises à jour des poids correspondants sont $\Delta W_{ij} = \eta_{lr}\frac{\tau}{\gamma_{LI}}\overline{\dot{\rho}_i}$, ce qui correspond à un taux d'apprentissage effectif :

$$lr = \eta_{lr}\frac{\tau}{\gamma_{LI}} \tag{2.12}$$

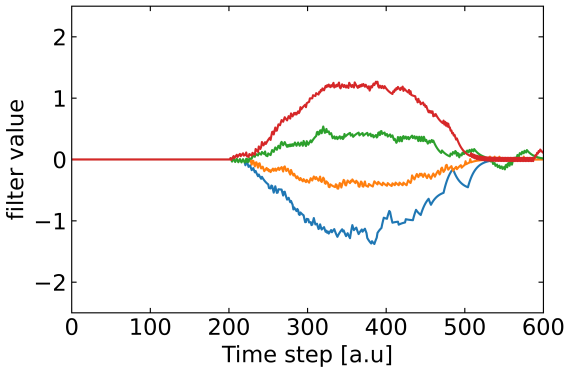
L'implémentation du bloc $\overline{\dot{\rho}}$, permet, pour le coût d'un bloc par neurone, de calculer la dérivée de la fréquence. Avec le détecteur d'impulsions, il est possible de calculer et appliquer le gradient d'erreur à la synapse localement en temps et en espace.



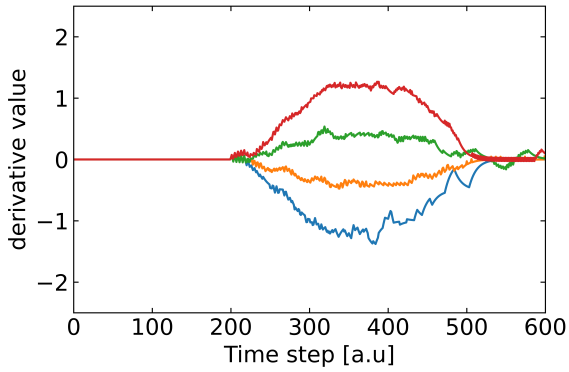
(a) Quatre fréquences de neurones et leur évolution au cours du temps. Les fréquences sont celles utilisées pour générer les graphes (b), (c), (d). Les fréquences sont comprises entre o et 1$T_{refract}^{-1}$.



(b) Valeur du bloc $\overline{\dot{\rho}}$ sans filtre, pour les fréquences de l'image (a).



(c) Valeur du bloc $\overline{\dot{\rho}}$ avec un filtre passe-bas à la sortie, pour les fréquences de l'image (a).



(d) Valeur du bloc $\overline{\dot{\rho}}$ avec un filtre passe-bas à l'entrée, pour les fréquences de l'image (a).

Figure 2.5 – Valeur de la fréquence des neurones et de la sortie du bloc $\overline{\dot{\rho}}$, (b) sans filtre, (c) avec le filtre à la fin du bloc, ou (d) au début.

L'approche proposée utilise des technologies connues, car les neurones *LIF*, les intégrateurs à fuite, les retardateurs et les filtres passe-bas sont des éléments de circuit qui peuvent être efficacement mis en œuvre avec des technologies CMOS [Mead and Ismail, 2012]. Les synapses bidirectionnelles pourraient être mises en œuvre avec des nanodispositifs émergents compatibles CMOS tels que les memristors [Ishii et al., 2019, Marković et al., 2020, Wan et al., 2020], ce qui sera traité au chapitre suivant.

2.3.2 Vérification de la loi d'apprentissage

La sous-section précédente a montré comment récupérer les paramètres nécessaires à la loi d'apprentissage et comment les appliquer à la synapse. Nous allons maintenant établir une méthodologie en simulation afin de vérifier l'application de la loi d'apprentissage sur deux neurones et une synapse.

Pour ce faire, nous fixons alternativement les fréquences, et respectivement, les accélérations de chaque neurone et utiliserons *EqSpike* pour mettre à jour le poids synaptique. La simulation a été testée avec 100 valeurs différentes uniformément réparties entre les limites de fréquence $[0, 1]\ T_{refract}^{-1}$.

Dans cette expérience, les neurones *LIF* sont pilotés par un signal d'entrée contrôlé pendant une durée fixe $T_{test} = 600dt$ durant laquelle ils émettent des impulsions à une fréquence proportionnelle au signal d'entrée. Cette fréquence d'émission d'impulsions en fonction de l'entrée est mesurée et représentée sur la Figure 2.1 et, comme expliqué précédemment, est équivalente à la fonction d'activation dans un réseau de neurones.

Pour montrer la proportionnalité de la loi d'apprentissage à la fréquence et à la dérivée des fréquences (leurs accélérations) $\rho_i\dot{\rho}_j$, nous regardons deux neurones connectés par une synapse. Les paramètres des neurones sont $\gamma_{LIF} = 0.001$ et $T_{refract} = 2dt$. Les paramètres de l'intégrateur à fuite sont $\gamma_{LI} = 0.1$, $N_{filter} = 10dt$ et $\tau = 100dt$. La fréquence des neurones est calculée par le nombre moyen d'impulsions dans une fenêtre glissante de $100\ dt$. Un signal d'entrée commande la fréquence de chaque neurone et un neurone n'intègre pas les impulsions envoyées par les autres neurones. Le facteur d'apprentissage η_{lr} est un facteur de proportionnalité sur dW. Le but de ces expériences étant de montrer la dépendance aux paramètres de la loi d'apprentissage, la valeur de η_{lr} n'est pas importante pour montrer cette proportionnalité, car elle changerait seulement l'ordre de grandeur de l'axe y. Elle est donc fixée à 1 pour plus de simplicité.

Pour montrer la dépendance à la fréquence ρ_i, le neurone postsynaptique i émet des impulsions à une fréquence comprise entre 0 et $1\ T_{refract}^{-1}$. Le neurone présynaptique émet un signal en trois parties de 200 dt ($100\ T_{refract}$) chacune, la première partie a une fréquence égale à $0.5\ T_{refract}^{-1}$, la deuxième partie a une accélération constante de $0.5/100\ T_{refract}^{-2}$, et la troisième partie est un signal constant avec une fréquence de $1T_{refract}^{-1}$.

Pour montrer la dépendance à l'accélération $\dot{\rho}_j$, le neurone postsynaptique i émet des impulsions à une fréquence de $0.5\ T_{refract}^{-1}$. Le neurone présynaptique j émet un signal en trois parties de 200 pas de temps chacune, la première partie a un taux égal à $0.5\ T_{refract}^{-1}$, la deuxième partie a une accélération constante entre $-0.5/100\ T_{refract}^{-2}$ et $0.5/100\ T_{refract}^{-2}$, et la troisième partie est un signal constant avec le même fréquence à la fin de l'accélération. La Figure 2.6 représente certaines configurations des signaux d'entrée pour les deux neurones en fonction du scénario.

Le poids synaptique est modifié selon *EqSpike* et la valeur finale au temps 600 dt est mesurée et reportée sur la Figure 2.7 pour chaque configuration de simulation différente. La courbe orange est une régression linéaire de tous les points. On observe une dépendance linéaire aux paramètres ρ et $\dot{\rho}$ de la loi d'apprentissage, validant en simulation l'évolution des poids synaptiques conformément à *EqSpike*.

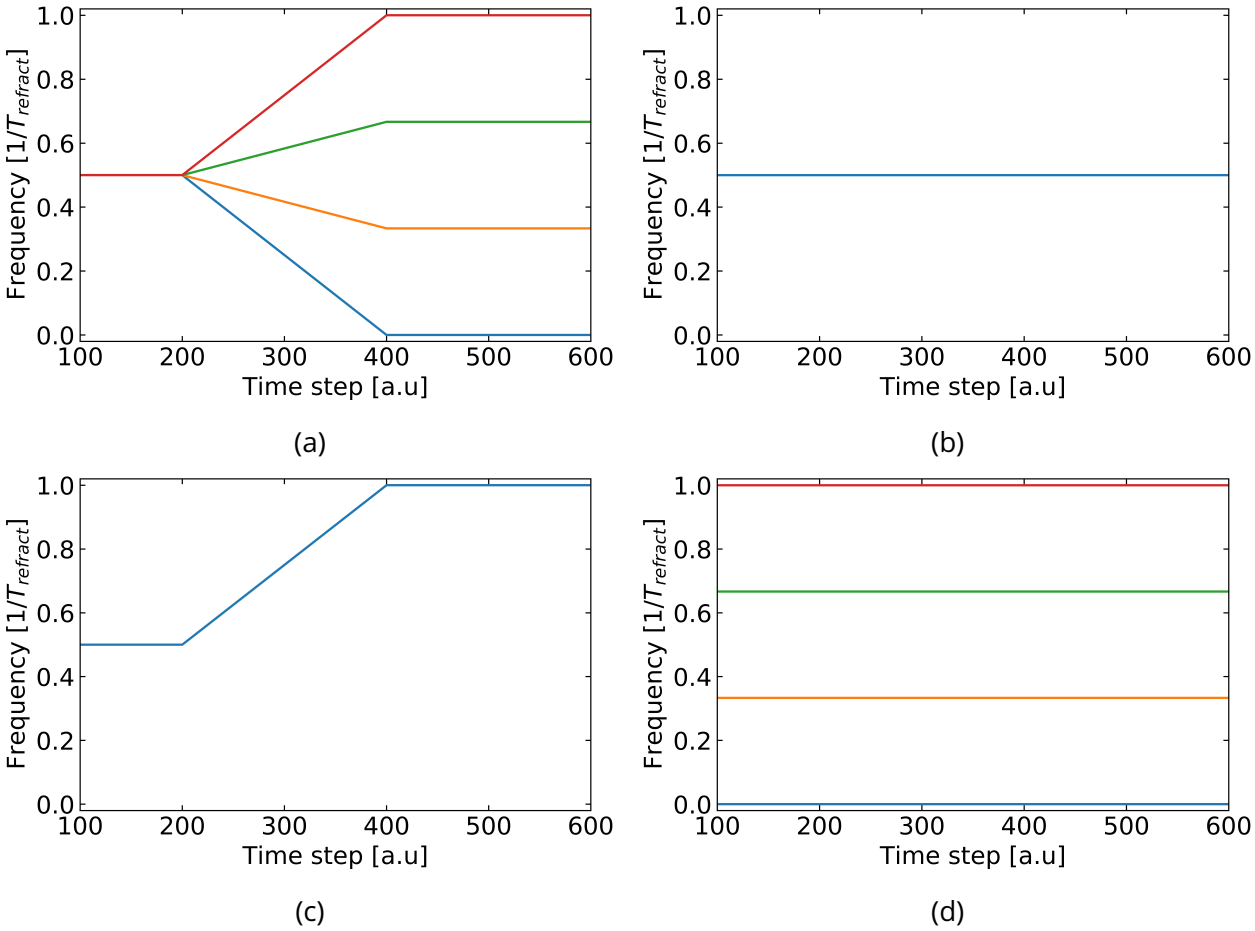
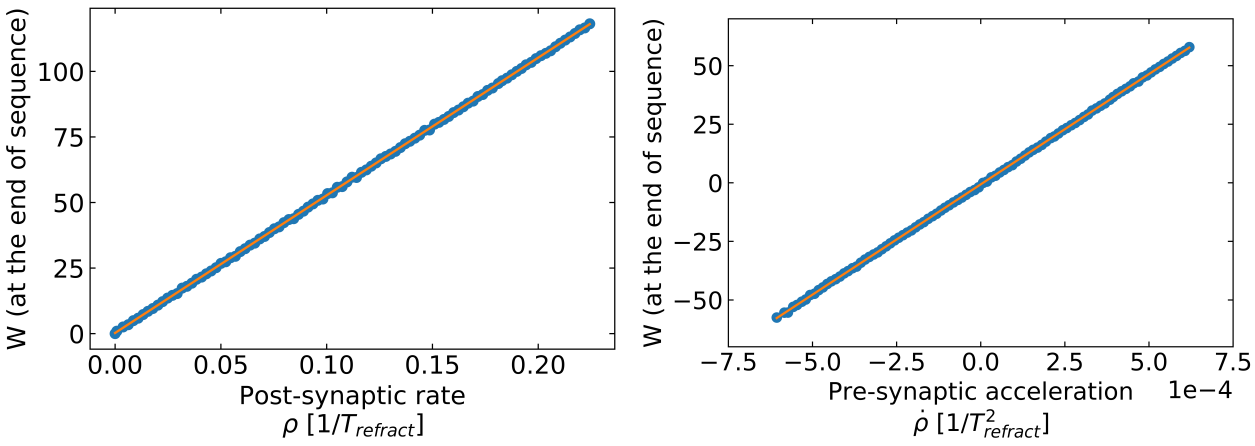


Figure 2.6 – Différentes évolutions de fréquence d'impulsion de neurones présynaptique à gauche et postsynaptique à droite. Le couple (a) et (b) est utilisé pour montrer la dépendance à l'accélération de la fréquence. Le couple (c) et (d) est utilisé pour montrer la dépendance à la fréquence.



(a) Dépendance la fréquence.

(b) Dépendance à l'accélération de la fréquence.

Figure 2.7 – Dépendance aux paramètres ρ et $\dot{\rho}$ de la loi d'apprentissage d'*EqSpike*.

2.3.3 Algorithme généralisé pour un réseau complet

La loi d'apprentissage est respectée dans le cas de deux neurones et d'une synapse. Pour généraliser l'algorithme à un réseau complet, il faut appliquer le bloc $\overline{\dot{\rho}}$ à chaque neurone. Le pseudo-code correspondant à la généralisation est donné dans l'Algorithme 2. Nous avons appelé cet algorithme : *EqSpike*, pour *Eq-Prop* avec des *spikes* (impulsions). L'algorithme dépend de plusieurs paramètres, leur description exacte ainsi que leur impact sur l'algorithme sont précisés dans la section suivante.

Data : Input, Network, Loss Function, Length Free Phase T_{free}, Length Nudging Phase T_{nudge}, Integrator leak γ_{LI}, Neuron leak γ_{LIF}, Neuron threshold u_{th}, Nudging Strength β, Learning rate η_{lr} , Time to compute derivative τ, Size of the Moving Window of the filter N_{filt}, Synaptic weight W_{ij}
Result : Trained weights for input image : W_{ij}
for $t < T_{free}$ **do**
  for *each neuron* j **do**
    Update membrane potential $u_j(\gamma_{LIF}, I_j)$
    if $u_j > u_{th}$ **then**
      Emit a spike (t_j)
      $u_j \leftarrow u_j - u_{th}$
    end
    Update $\rho_j(t_j, \gamma_{LI})$
  end
end
for $t \in [T_{free}, T_{free} + T_{nudge}]$ **do**
  for *each output neuron* o **do**
    Compute error gradient ∇e_o
    Nudge neuron : $u_o \leftarrow u_o - \beta \cdot \nabla e_o$
  end
  for *each neuron* k **do**
    Update $u_k(\gamma_{LIF}, I_k)$
    if $u_k > u_{th}$ **then**
      Emit a spike (t_k)
      $u_k \leftarrow u_k - u_{th}$
    end
    Update $\rho_k(t_k, \gamma_{LI})$ Compute smoothed : $\overline{\dot{\rho}_k}((\rho_k(t_k), \rho_k(t_k - \tau)), ..., N_{filt}$
  end
  for *each synapse* w_{ij} **do**
    if *neuron j emits a spike* **then**
      $w_{ij} \leftarrow w_{ij} + \eta_{lr} \cdot \frac{\tau}{\gamma_{LI}} \overline{\dot{\rho}_i}$
    end
    if *neuron i emits a spike* **then**
      $w_{ij} \leftarrow w_{ij} + \eta_{lr} \cdot \frac{\tau}{\gamma_{LI}} \overline{\dot{\rho}_j}$
    end
  end
end

Algorithme 2 : Algorithme *EqSpike* pour un réseau de neurones génériques

Pour cet algorithme, les fréquences doivent être stockées localement pour une durée τ, afin d'obtenir une dérivée de la fréquence avec un τ plus petit que la durée de la phase d'inférence et de la durée de la phase d'apprentissage. Pour cette raison, la fréquence des neurones est calculée même pendant la phase d'inférence. En effet, le calcul de $\overline{\dot{\rho}(t)}$ dans la phase d'apprentissage requiert les valeurs de la fréquence ρ au temps $t - \tau$, instant pouvant se situer dans la phase d'inférence lors de la jonction des deux phases.

Grâce à l'ajout localement du bloc $\overline{\dot{\rho}}$, notre algorithme *EqSpike* peut être considéré comme local en espace et en temps, à un τ près (ce qui correspond à une capacité dans un circuit électronique).

L'algorithme présenté est relatif à une donnée d'entrée, par exemple une image à classifier. Entre chaque nouvelle donnée, le réseau de neurones doit réinitialiser certains paramètres. Les potentiels de membrane u des neurones, la valeur des intégrateurs et leurs filtres sont réinitialisés à 0.

Il a été montré expérimentalement [Laborieux et al., 2021] qu'il est plus avantageux de modifier le signe de β et le taux d'apprentissage η_{lr} (conjointement) de manière aléatoire, pour gagner en précision, comparé à un signe de β fixe. Dans notre cas, nous avons choisi arbitrairement la probabilité de changer le signe de β à une sur deux. Ce changement n'est pas illustré dans le pseudo-code, mais a été utilisé pour obtenir les résultats dans les parties suivantes.

2.4 Classification d'images avec l'algorithme *EqSpike*

Dans le cadre de cette thèse, les problèmes traités sont des problèmes de classification d'images naturelles. Nous avons choisi le jeu de données *MNIST* comme référence, car il s'agit d'un jeu de données standard pour la communauté neuromorphique intéressée par l'entraînement sur puce des réseaux de neurones à impulsions et que *EqSpike* nécessite des entrées statiques. Cette section comprendra tout d'abord une présentation des jeux de données utilisés, puis une discussion sur les hyperparamètres d'*EqSpike*, pour finir sur les résultats sur le jeu de données *MNIST* et leur comparaison à la littérature.

2.4.1 Jeux de données et fonction de coût

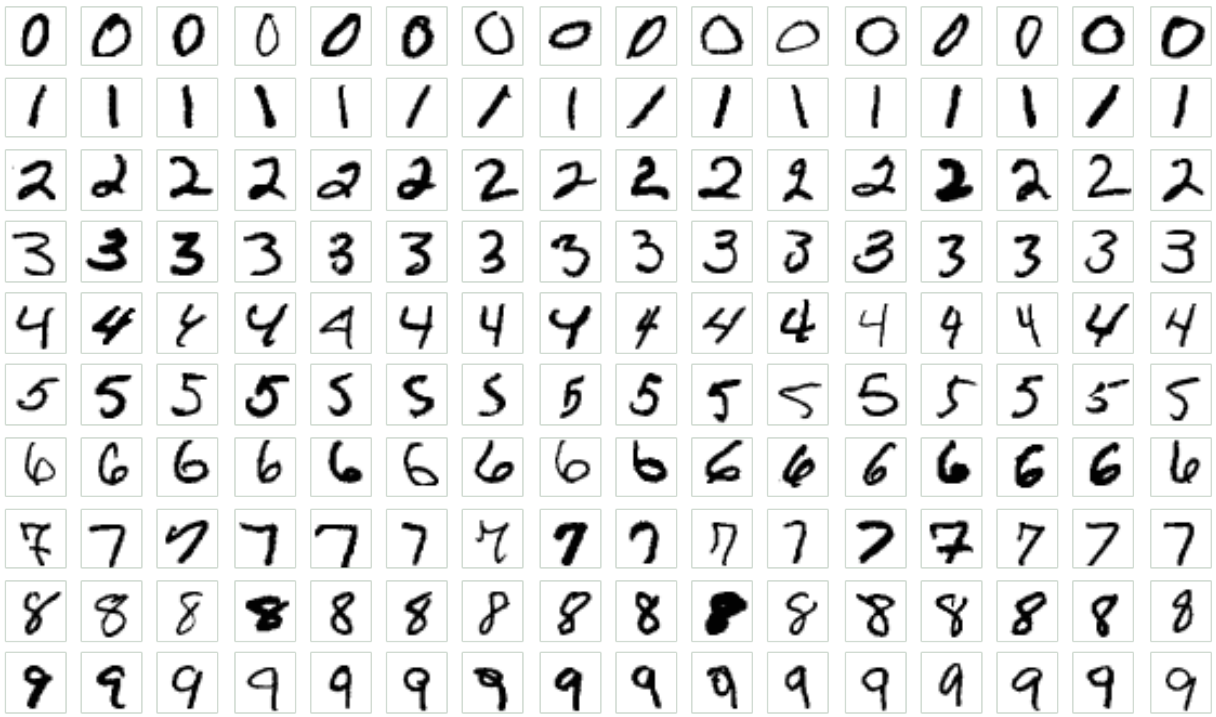


Figure 2.8 – Exemple d'images de *MNIST*, pris sur https://en.wikipedia.org/wiki/MNIST_database (les couleurs sont inversées pour une meilleure lisibilité).

Le jeu de données *MNIST* [LeCun et al., 2010] contient 60 000 images d'entraînement et 10 000 images de test. Il représente des images de chiffres manuscrits de 28x28 pixels, les pixels représentant des niveaux de gris. La normalisation de la valeur des pixels a été mise entre 0 et 0.5 (f_{max}). La Figure 2.8 montre 160 exemples d'images. Les pixels sont envoyés avec un signal continu de leur valeur sur les neurones d'entrée. Nous avons choisi le jeu de données *MNIST* comme référence, car il s'agit d'un jeu de données standard pour la communauté neuromorphique intéressée par l'entraînement sur puce des réseaux de neurones à impulsions [Fouda et al., 2019a, Pfeiffer and Pfeil, 2018, Tavanaei et al., 2019]. La fonction de coût est la

fonction à minimiser pour le réseau. La fonction utilisée est l'erreur quadratique moyenne, rappelée ici :

$$L(\hat{y}, y) = \frac{1}{M} \sum_{k=0}^{M} (\hat{\rho_k} - \rho_k)^2,$$

où la somme est calculée sur tous les neurones de sortie. La cible $\hat{\rho_k}$ correspond à un neurone avec la fréquence maximum f_{max} si la classe est correcte, et à 0 sinon.

Durant la phase d'apprentissage, les neurones de sortie intègrent la dérivée de l'erreur $(\hat{\rho_k} - \rho_k)$ pondérée par le facteur de perturbation β en plus des entrées des neurones de la couche cachée. La fréquence de sortie, ρ_k, est estimée en comptant le nombre d'impulsions sur 100 dt de chaque neurone de sortie. L'erreur est calculée à chaque dt, ainsi que sa dérivée.

2.4.2 Discussions sur le choix des hyperparamètres

L'algorithme *EqSpike* contient un nombre important d'hyperparamètres. Cette sous-section traite des paramètres jugés les plus importants et de leur effet, ainsi que de leur signification physique pour une puce neuromorphique. Ces hyperparamètres ont été fixés avec différentes expérimentations et une recherche par grille pour certains d'entre eux. Les valeurs essayées seront spécifiées lors de leur description. La recherche exhaustive d'une précision optimale est infaisable sans heuristiques élaborées au vu de la combinatoire de tous les paramètres. Ces paramètres sont interdépendants et leur interaction est difficile à modéliser. Par conséquent, cette sous-section se limite à comprendre leur impact individuel sur la précision de réseau.

Pour des raisons de lenteur d'exécution, en simulation, nous avons d'abord utilisé *EqSpike* sur un jeu de données que nous appelons *Digits*. Ce jeu de données comprend 1797 images en 8x8 pixels en niveaux de gris, représentant des chiffres manuscrits. Ce jeu de données nous a permis de pré-calibrer nos paramètres et d'en comprendre l'impact, avant de passer à des problèmes plus complexes comme *MNIST*.

$T_{refract}$ est le temps pendant lequel un neurone ne peut pas émettre de nouvelles impulsions après en avoir émis une. $T_{refract}$ est notre unité temporelle servant d'étalon pour toutes les autres. Il est possible de réaliser des neurones avec des fréquences de l'ordre du Hertz comme des neurones avec des fréquences de l'ordre du MHz. *EqSpike* se veut le plus général possible pour s'adapter aux différents neurones physiques. Nous avons décidé d'utiliser le temps de réfraction des neurones pour définir nos autres temps. Cela signifie que notre algorithme peut fonctionner pour n'importe quelle vitesse de neurones, à condition que les autres contraintes et paramètres soient respectés pour l'algorithme.

T_{free} correspond au temps de la phase d'inférence. Dans la théorie d'*Eq-Prop*, le réseau doit converger dans la première phase, et à ce moment-là, on mesure l'état des neurones et on passe à la phase suivante. Dans la perspective où notre réseau tourne sur une puce neuromorphique, il serait coûteux en place et en énergie d'avoir des capteurs sur les neurones pour mesurer leur état pour savoir s'ils ont convergé. Nous avons donc fait le choix de rendre fixe le temps des deux phases en estimant que le réseau aura convergé à la fin des deux phases. Plusieurs dizaines de réseaux de neurones ont été appris avec différentes valeurs de T_{free}. Au-delà de la valeur choisie dans la table 2.1, la précision n'augmentait plus, indiquant une possible convergence dans la phase d'inférence.

T_{nudge} correspond au temps de la phase d'apprentissage. Il est fixe pour les mêmes raisons que le temps de la phase d'inférence. Si le réseau de neurones n'a pas fini de converger à la fin de la phase d'apprentissage, la dynamique aura rapproché l'état du réseau vers un état plus proche de l'état voulu. Dans ce cas, le réseau aura réduit l'erreur et le gradient d'erreur aura le bon signe, mais une norme approximée du gradient parfait. Les durées ont été choisies de manière expérimentale pour un réseau à une couche cachée. Elles doivent être adaptées au problème et à la topologie du réseau.

EqSpike utilise un temps discrétisé. Le pas de temps dt doit être un compromis entre précision de la simulation et temps d'exécution. Un pas petit signifie une meilleure précision dans la simulation au détriment du temps de calcul. Le gain d'un dt plus faible que $dt = 0.5T_{refract}$ est empiriquement considéré comme trop faible comparé au temps de calcul ajouté, le réseau étant déjà très lent à simuler sur des problèmes suffisamment complexes pour avoir besoin de réseaux de neurones.

En effet, le nombre d'itérations pour mettre à jour l'état de chaque neurone et synapse (nombre de fois où est calculée la sortie) de l'algorithme, noté ici O, est

$$O = N_{data} \cdot N_{epoch} \cdot (T_{free} + T_{nudge}) \cdot \frac{1}{dt} \tag{2.13}$$

avec N_{data} le nombre de données dans le jeu de données d'entrainement et N_{epoch} le nombre d'époques (le nombre de fois où ce jeu de données est vu). Le nombre de mises à jour est combinatoire et peut vite augmenter avec des changements d'ordre de grandeur de plusieurs paramètres. Par exemple, pour le jeu de données *MNIST* avec 50 époques et $(T_{free} + T_{nudge}) = 175T_{Refract}$, il y a 1 050 000 000 d'itérations par neurone et par synapse.

De plus, *EqSpike* étant séquentiel par nature, par synapse il est impossible d'exécuter le code plus rapidement grâce à la parallélisation. Sur une puce neuromorphique, le temps de calcul dépend de la vitesse des neurones, mais l'actualisation du réseau, soit un pas de temps, est réalisée de manière parallèle. Une mise à l'échelle sur des réseaux plus larges peut donc être effectuée sans surcoût de temps si les temps des phases ne changent pas (loi d'*Amdahl* [Amdahl, 1967]).

γ_{LIF} représente la fuite du potentiel de membrane des neurones *LIF*. Nous avons choisi une fuite faible pour avoir la fonction d'activation la plus proche possible d'une *hard sigmoid*. Sur un neurone CMOS, la fuite provient d'un condensateur. Une fuite trop importante empêcherait les neurones d'être sensibles aux petites fréquences. En effet, si une fréquence est trop basse, le potentiel de membrane ne pourra jamais atteindre le seuil u_{th} car la valeur intégrée d'une impulsion aura fuité avant l'impulsion suivante. Si la fuite est nulle, cela revient à avoir un neurone de type IF et ne devrait pas impacter les résultats d'*EqSpike*.

γ_{LI} représente la fuite de l'intégrateur pour estimer la dérivée de la fréquence, $\dot{\overline{\rho}}$. La mise en œuvre physique peut être réalisée de la même façon que γ_{LIF} sur une technologie CMOS. La fuite de l'intégrateur permet de faire converger la valeur de l'intégrateur vers une valeur proportionnelle à la fréquence du neurone connecté. L'intégrateur va converger vers un état d'équilibre où la valeur de fuite sera égale à la valeur intégrée. Rappelons que l'état d'équilibre est atteint quand :

$$V_{delay} = V_{LI}(t) - V_{LI}(t-\tau) \cong \tau \frac{\Delta V_{LI}}{\Delta t} \propto \frac{\tau}{\gamma_{LI}} \dot{\rho}$$

avec ΔV_{LI} la différence des tensions V_{LI} et $\Delta t = t - \tau$. Si cette fuite est trop faible comparée au temps de la phase d'apprentissage T_{nudge}, alors le temps de convergence de l'intégrateur ne sera pas assez rapide pour être pris en compte lors de l'apprentissage, la décélération de la fréquence au minimum sera encodée par la fuite. En revanche, si elle est trop forte, alors l'intégrateur retournera à 0 tous les un ou deux pas temps, perdant l'information obtenue. γ_{LI} sélectionne donc une fréquence minimum intégrable dans notre temps d'apprentissage.

u_{th} est le seuil du potentiel de membrane du neurone. Si ce seuil est atteint par u alors le neurone émet une impulsion et réduit u d'une valeur u_{th}. Dans notre algorithme, comme nous ne sommes pas soumis à des neurones avec des valeurs physiques prédéfinies, nous utilisons le seuil du neurone comme référence pour la tension d'impulsion du neurone qui sera d'amplitude u_{th}. La valeur de u_{th} dépendra de la technologie du neurone. La valeur de l'intégrale d'une impulsion est de $1u_{th}$, donc un neurone qui reçoit une impulsion avec un poids synaptique de 1 va émettre une impulsion à son tour. La fonction d'activation étant, dans notre cas, la *hard sigmoid*, nous avons choisi des valeurs d'impulsion et de seuil qui permettent de reproduire cette fonction facilement.

τ est le temps entre deux états de l'intégrateur pour calculer la dérivée de la fréquence, comme montré dans l'équation 2.10. Plus τ est élevé, moins l'estimation sera sensible au bruit de la valeur de l'intégrateur V_{LI}. Il est préférable que le réseau de neurones à un temps $T_{free} - \tau$ soit déjà dans un état stable pour ne prendre en compte que le changement de dynamique apporté par la phase d'apprentissage. En effet, si le réseau de neurones au temps $T_{free} - \tau$ n'est pas encore stabilisé, les premiers pas de temps dans la phase d'apprentissage vont prendre en compte la dérivée de la fréquence pendant la phase d'inférence.

η_{lr} est le taux d'apprentissage (*learning rate*, en anglais). Contrairement à l'algorithme de la rétro-propagation du gradient d'erreur, le taux d'apprentissage ne correspond pas à un taux sur le gradient d'erreur, mais à un taux sur la sortie du filtre des blocs $\dot{\overline{\rho}}$. Le taux d'apprentissage effectif est représenté par

l'équation 2.12, que nous rappelons ici :

$$lr = \eta_{lr}\frac{\tau}{\gamma_{LI}}$$

La valeur montrée dans le tableau 2.1 a été trouvée sur un test entre [2.10^{-4}, 1.10^{-4}, 2.10^{-5}, 1.10^{-5}, 2.10^{-6}, 1.10^{-6}] sur le jeu de données *Digits* sur 10 runs pour chaque valeur.

N_{filt} est la durée de la fenêtre du filtre passe-bas, en pas de temps. Le filtre sert à lisser l'approximation de la dérivée de la fréquence. Une valeur trop grande nous donne un trop grand délai de réaction, ce qui oblige à augmenter T_{nudge}. Une valeur trop faible ne nous permet pas de lisser l'approximation de la dérivée de la fréquence, rendant le filtre inutile.

β est un facteur exprimant la force de perturbation de l'erreur sur la dernière couche du réseau. Le réseau, qui va être perturbé pendant la phase d'apprentissage, à sa vitesse de changement de dynamique conditionnée par β. De la même façon qu'un taux d'apprentissage à 1 pour la rétro-propagation du gradient, un β élevé ne permet pas une bonne généralisation du réseau. Si β est trop grand, on forcera trop à modifier les valeurs des poids pour apprendre la donnée en entrée au détriment des valeurs déjà apprises et de la généralisation.

Le tableau 2.1 liste tous les hyperparamètres et les valeurs utilisées pour la suite du chapitre, sauf indication contraire.

simulation timestep, dt	γ_{LIF}	γ_{LI}	u_{th}	τ ($T_{refract}$)
0.5	0.01	0.1	1	50
η_{lr}	N_{filt}	β	T_{free} ($T_{refract}$)	T_{nudge} ($T_{refract}$)
3×10^{-6}	10	0.5	75	100

Table 2.1 – Hypeparamètres d'*EqSpike*

2.4.3 Classification d'images avec l'algorithme *EqSpike*

Nous évaluons maintenant les performances d'*EqSpike* sur la tâche de classification de chiffres manuscrits de 0 à 9 *MNIST* et *Digits*, en utilisant un réseau entièrement connecté (*fully connected*, en anglais) avec une couche cachée.

Les hyperparamètres ont été optimisés, avec une recherche par grille, pour obtenir la meilleure précision sur un réseau de neurones avec une topologie de 784 neurones d'entrée, 100 neurones en couche cachée et 10 neurones de sortie (784-100-10), et sont représentés sur la table 2.1.

Comme présenté dans le chapitre précédent, un élément important des réseaux de neurones est le biais. Dans *EqSpike*, le biais est mis en œuvre comme un neurone qui émet des impulsions à la fréquence maximum des neurones et qui est connecté à tous les neurones de la couche. Il peut être vu comme une entrée supplémentaire pour chaque couche de neurones. Comme le biais est unidirectionnel, la fréquence du biais ne change pas pendant l'exécution et la règle d'apprentissage est :

$$b_i = f_{max}\dot{\rho}_i \tag{2.14}$$

Les précisions obtenues, sur *MNIST*, lors de l'entraînement (en orange) et du test (en bleu) sont présentées à la Figure 2.9 en fonction du nombre d'époques d'entraînement. Le tableau 2.3 compare les résultats obtenus avec les algorithmes *BPTT* et *Continual Equilibrium Propagation*, sur des réseaux de neurones formels. Les paramètres sont représentés dans la Table 2.2. Le *batch size* correspond au nombre de données présentées au réseau avant de modifier les poids synaptiques. Si le *batch size* est supérieur à 1 alors la modification des poids, dW, correspond à la moyenne des dW calculée pour chaque image du *batch*. Cette technique permet un dW plus généralisé grâce à l'effet de moyenne. Le but de ces travaux est d'implémenter, à terme, *EqSpike* sur une puce neuromorphique. Pour garder les avantages énergétiques et neuromorphiques, il n'est pas envisageable d'enregistrer les modifications pour chaque époque et de calculer ensuite la moyenne.

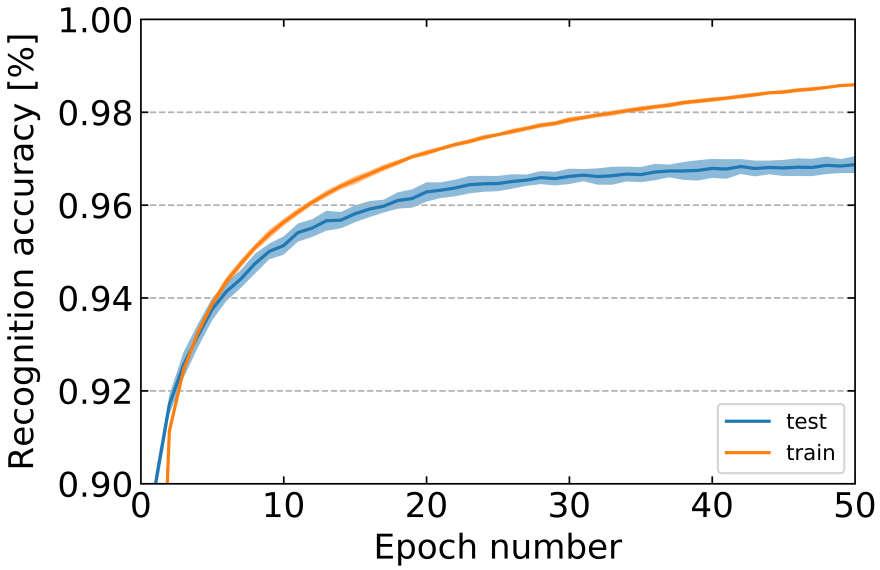


Figure 2.9 – Précision d'*EqSpike* sur le jeu de données de *MNIST*, moyenne obtenue sur 6 exécutions, avec un réseau de taille 764-100-10 initialisé aléatoirement (*Xavier initialization*).

	Algo-rithme	Topo-logie	Acti-vation	Temps 1ère phase	Temps 2ème phase	β	Taux d'ap-pren-tissage
MNIST	BPTT	784-100-10	*hard sigmoid*	30	15	NA	0.003 -0.0015
MNIST	C-EP	784-100-10	*hard sigmoid*	30	15	0.5	0.003 -0.0015

Table 2.2 – Hyperparametres de *BPTT* et de *Continual Equilibrium Propagation*

Pour permettre une comparaison rigoureuse avec les autres algorithmes, la même taille de *batch*, 1, a été considérée pour tous les algorithmes.

Algorithme	BPTT 784-100-10	Continual Eq-Prop 784-100-10
MNIST	Test : $97.11\% \pm 0.23\%$ Train : $99.06\% \pm 0.15\%$	Test : $96.97\% \pm 0.12\%$ Train : $99.8\% \pm 0.04\%$
Algorithme	*EqSpike* 100 784-100-10	*EqSpike* 300 784-300-10
MNIST	Test : $96.87\% \pm 0.18\%$ Train : $98.59\% \pm 0.03\%$	Test : $97.59\% \pm 0.1\%$ Train : $98.91\% \pm 0.03\%$

Table 2.3 – Comparaison des résultats entre *BPTT*, *Continual Equilibrium Propagation* et *EqSpike*, avec la même procédure d'initialisation

Nous avons lancé 5 fois les réseaux de neurones avec des poids synaptiques tirés aléatoirement sur une gaussienne de moyenne 0 et de variance $(\sqrt{\frac{1}{k}})^2$, avec k le nombre d'entrées du neurone. Cette initialisation est appelée *Xavier initialization* [Glorot and Bengio, 2010]. L'avantage de cette méthode est d'avoir une entrée du neurone avec une moyenne à 0 et une déviation standard à 1, ce qui permet d'éviter les explosions ou disparitions du gradient.

La précision de test de *EqSpike* correspond étroitement à la précision de la descente de gradient stochastique par BPTT sur la même architecture de réseau, compte-tenu de la marge d'erreur, tableau 2.3. Avec une couche cachée de 300 neurones, *EqSpike* atteint une précision de test de 97,59%. Les réseaux de neurones à impulsions entièrement connectés atteignent généralement des taux de reconnaissance de l'ordre de 96% à 98%, tableau 2.4. *EqSpike* obtient une précision similaire à celle des autres réseaux de neurones à impulsions sur le jeu de données *MNIST*, mais en évitant de calculer la rétro-propagation du gradient et en étant local et intrinsèque.

Modèle	Architecture	Algorithme	Localité et règle d'apprentissage	Précision sur *MNIST* (%)
[Lee et al., 2016]	Deep SNN (784-300-300-10)	Back-propagation	Non Local	98.8
[Zhang et al., 2020a]	SNN (784-800-10)	Spike-Timing-Dependent Back-Propagation	Non local	98.5
[Sakemi et al., 2020]	SNN (784-800-10)	Temporal Backpropagation	Non local	98.04
[Neftci et al., 2017]	Deep SNN (784-500-500-10)	Event-driven random BP	Non Local	97.98
[Comsa et al., 2020]	SNN (784-340-10)	Temporal Backpropagation	Non local	97.9
[Park et al., 2020]	SNN (784-200-200-10)	SD algorithme	Non local	97.83
EqSpike	Bidirectional SNN (784-300-10)	EqSpike	Local	97.6
[Kheradpisheh and Masquelier, 2020]	SNN (784-400-10)	Temporal Backpropagation	Non local	97.4
[Mostafa, 2018]	SNN (784-800-10)	Temporal Backpropagation	Non Local	97.14
[Querlioz et al., 2013]	SNN (784-300)	*STDP*	Local	93.5

Table 2.4 – Comparaison des précisions entre *EqSpike* et d'autres algorithmes d'apprentissage sur *MNIST*

2.5 Performance théorique sur un matériel électronique

Il est intéressant, pour les applications neuromorphiques, de quantifier le nombre d'impulsions nécessaires pour réaliser l'inférence et le temps nécessaire pour atteindre une haute précision. Un fonctionnement avec moins d'impulsions est souhaitable, car il réduit à la fois le temps d'exécution et la consommation d'énergie.

La ligne pointillée verticale rouge dans la Figure 2.10 indique le temps, en moyenne, de la première impulsion sur l'ensemble des neurones dans la couche de sortie. La moyenne est effectuée sur toutes les images présentées et correspondant à $t \cong \frac{3.5}{f_{max}}$. A $t \cong \frac{10}{f_{max}}$, la précision de l'inférence de la première impulsion (courbe bleue) atteint 95,11% $\pm$ 0,78%, à moins de 1,4% de perte de précision avec la fréquence maximale (courbe orange).

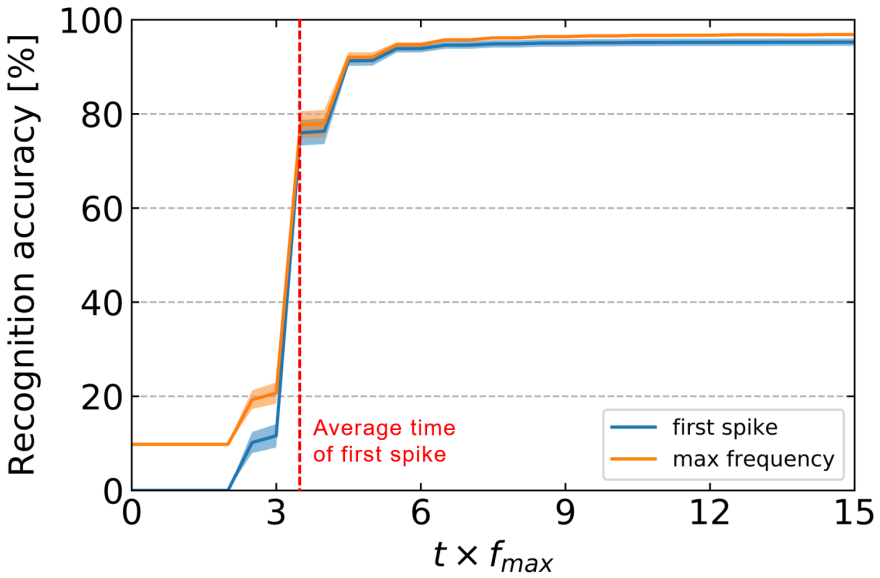


Figure 2.10 – Temps d'inférence : précision de la reconnaissance sur le jeu de données de test de *MNIST* en fonction du temps, multiplié par la fréquence maximum des neurones f_{max}.

Ce résultat montre que, même si *EqSpike* est initialement basé sur l'estimation de la fréquence maximale à la fin de la phase d'inférence (à l'équilibre), une seule impulsion à la sortie suffit dans la plupart des cas pour déterminer la classe correcte avec une bonne précision, une caractéristique qui est très attrayante pour une inférence économe en énergie sur les puces neuromorphiques.

Cela signifie que l'inférence peut être réalisée en 100μs pour des neurones électroniques avec une fréquence maximale de $100kHz$, disponibles dans les puces neuromorphiques fonctionnant en temps accéléré par rapport à la biologie [Schemmel et al., 2010], et en 1μs pour des neurones électroniques avec une fréquence maximale de $10MHz$ qui peuvent être produits, par exemple, avec les nanotechnologies émergentes [Li et al., 2015].

Le débit de calcul correspondant est respectivement de 10 000 et 1 million d'images par seconde, ce qui correspond aux implémentations actuelles des réseaux de neurones à impulsions [Pfeiffer and Pfeil, 2018, Park et al., 2020]. Comme les opérations du réseau sont parfaitement parallèles, ces ordres de grandeur seront conservés pour des réseaux plus larges. Les simulations d'*Eq-Prop* sur des réseaux plus profonds, indiquent que le temps de convergence augmente d'un facteur d'environ huit pour un réseau à quatre couches cachées par rapport à un réseau à une couche cachée comme ici [Laborieux et al., 2021].

Il convient de noter que dans l'implémentation actuelle, nous présentons des entrées statiques au réseau, ce qui signifie que les neurones d'entrée doivent intégrer ces signaux avant d'émettre les premières impulsions qui se propageront ensuite aux couches suivantes. La vitesse d'inférence pourrait être augmentée à l'avenir en présentant des entrées directement encodées dans les impulsions, par exemple à partir de capteurs de vision neuromorphiques [Pfeiffer and Pfeil, 2018] (à condition que ces entrées soient statiques).

Une estimation de la consommation d'énergie d'un réseau de neurones à impulsions sur une puce de silicium neuromorphique peut être effectuée en comptant le nombre d'opérations synaptiques impliquées.

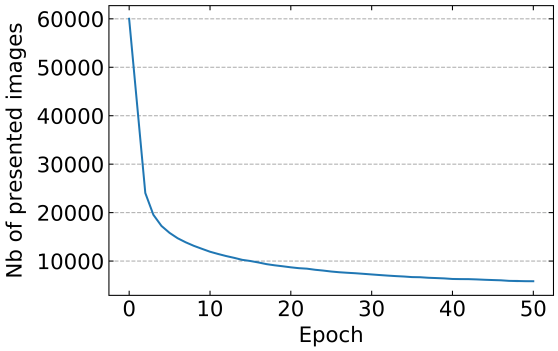
Les opérations synaptiques (SynOps) sont définies comme le nombre total d'impulsions transitant par les synapses du réseau. [Frenkel et al., 2019a] montrent qu'une SynOp sur une puce neuromorphique nécessite peu d'énergie, $10pJ$. Le nombre total d'opérations synaptiques nécessaires pour l'inférence dépend de la précision de reconnaissance visée et, par conséquent, de la durée de l'inférence, comme présenté dans la Figure 2.12.

Le taux de reconnaissance sature à $t \cong \frac{10}{f_{max}}$. Le nombre correspondant de SynOps mesurées, à ce temps, est de 150000 en moyenne. C'est moins que ce qui serait attendu si tous les neurones étaient en train d'émettre des impulsions de manière continue. C'est comparable au nombre de SynOps nécessaires à l'inférence pour l'algorithme [Neftci et al., 2017].

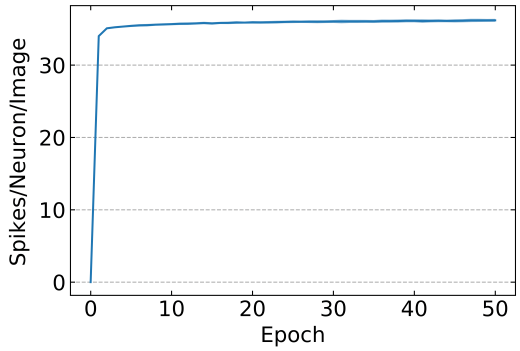
En considérant $10pJ/SynOps$, chaque inférence d'*EqSpike* pourrait potentiellement consommer $1.5\mu J$. Cela signifie que le test des 10000 images de l'ensemble des données *MNIST* pourrait être réalisé avec une puce neuromorphique tout en consommant seulement $15mJ$, c'est-à-dire trois ordres de grandeur de moins qu'avec un GPU [Joseph and Nagarajan, 2020].

Dans notre implémentation actuelle d'*EqSpike*, la couche d'entrée est celle qui émet la plupart des impulsions et des SynOps : avec seulement 16% des pixels blancs en moyenne dans *MNIST*, la couche d'entrée émet 87.5% de toutes les impulsions et 98.6% des SynOps se produisent entre la couche d'entrée et la couche cachée.

Nous ne nous sommes pas concentrés sur la réduction du nombre d'impulsions avec l'encodage, mais un meilleur encodage de l'entrée peut réduire considérablement la consommation d'énergie. [Kheradpisheh and Masquelier, 2020] ont montré qu'avec un encodage temporel, le nombre total d'impulsions dans le réseau avant la première impulsion de sortie peut être réduit à 200 avec une couche cachée quatre fois plus grande que la couche cachée de notre réseau. Dans notre cas, 678 impulsions au total sont émises en moyenne avant la première impulsion de sortie. Une adaptation d'*EqSpike* à l'encodage temporel n'est pas simple, car cet algorithme a été pensé avec des entrées statiques et continues, mais ce nombre pourrait potentiellement être diminué dans le futur en réduisant la fréquence d'encodage de l'entrée.



(a) Nombre d'images vues lors de la phase d'apprentissage par époque en fonction du nombre époques.



(b) Nombre d'impulsions/neurone/image durant la phase d'apprentissage en fonction des époques.

Figure 2.11 – Performance d'entraînement d'*EqSpike* pour le nombre d'images apprises par époque et le nombre d'impulsions par neurone par image

L'entraînement avec *EqSpike* nécessite d'effectuer la phase d'inférence, puis la phase d'apprentissage, pendant laquelle les poids synaptiques sont mis à jour. Une façon d'accélérer l'apprentissage, et de réduire le nombre total de SynOps, est d'effectuer la phase d'apprentissage uniquement sur les exemples mal classés, et d'éviter les mises à jour lorsque la précision est satisfaisante. De plus, cela évite de sur-apprendre des exemples bien classés. Nous appliquons cette stratégie inspirée de [Park et al., 2020] : quant à la fin de la phase d'inférence, la somme des différences entre les cibles et les fréquences de sortie, $(\hat{\rho_k} - \rho_k)$ est supérieure à 1%, alors la phase d'apprentissage n'est pas effectuée. La Figure 2.11a montre le nombre d'exemples qui sont vus pendant la phase d'apprentissage par époques en fonction du nombre d'époques. Dans les 20 dernières époques, environ 15% seulement du jeu de données d'entraînement nécessitent encore une phase d'apprentissage. Pour *MNIST*, nous effectuons donc la phase d'inférence sur l'ensemble du jeu de données (3×10^6 images pour les 50 époques), et la phase d'apprentissage sur 489000 images. Compte-tenu des durées

de chaque phase, nous pouvons estimer le temps d'entrainement du réseau à $T_{training} \cong \frac{2.74\times10^8}{f_{max}}$. Pour les neurones électroniques avec une fréquence maximale de $100kHz$ [Schemmel et al., 2010], cela conduit à $T_{training} \cong 45min$, et pour les neurones électroniques avec une fréquence maximale de $10MHz$ [Li et al., 2015] à $T_{training} \cong 30s$. Comme nos réseaux sont par nature entièrement parallèles, ces temps d'entraînement seraient les mêmes pour des réseaux beaucoup plus larges et augmenteraient d'un facteur huit seulement avec quatre couches cachées [Laborieux et al., 2021].

Comme *EqSpike* est dérivé d'une approche basée sur la fréquence, il est intéressant de comparer les fréquences réelles des neurones du réseau pendant l'apprentissage à leur fréquence maximale f_{max}. Pour les applications neuromorphiques, des fréquences moyennes basses sont en effet souhaitables pour réduire la consommation d'énergie. La Figure 2.11b montre le nombre moyen d'impulsions émises par chaque neurone pour une présentation d'image dans l'ensemble de données d'entraînement, en fonction de l'époque. Nous avons trouvé que pour les conditions d'entraînement de la Figure 2.9, il y a en moyenne 36 impulsions/neurone/image. Cela signifie que les neurones du réseau produisent des impulsions en moyenne avec une fréquence de l'ordre de 20% de f_{max}, bien en dessous de f_{max}, ce qui est prometteur pour des implémentations neuromorphiques. Encore une fois, ce nombre pourrait être réduit à l'avenir en optimisant l'encodage de l'entrée au niveau de la première couche. La Figure 2.12 montre le nombre d'opérations synaptiques nécessaires à l'entraînement en fonction de la précision. Le nombre total d'opérations synaptiques après 50 époques est de $4,23\times10^{12}$, ce qui est du même ordre de grandeur que l'algorithme avec rétro-propagation du gradient basée sur [Neftci et al., 2017] pour une précision similaire et inférieur à l'entraînement de *MNIST*.

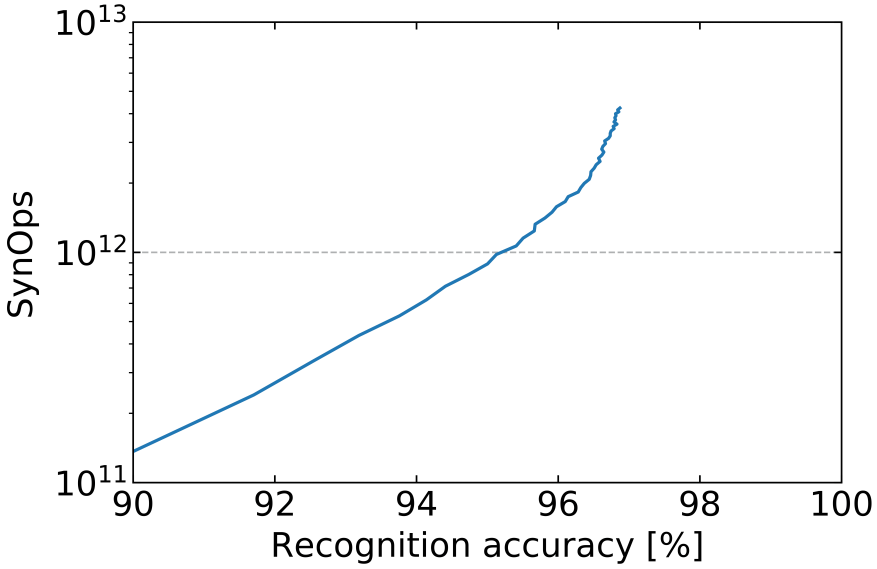


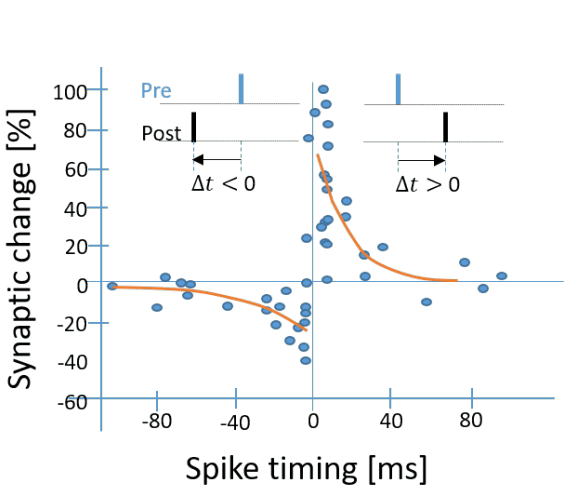
Figure 2.12 – SynOps : nombre d'impulsions durant les deux phases en fonction de la précision.

Avec $10pJ$ par SynOps [Frenkel et al., 2019a], la phase d'entraînement d'*EqSpike* sur une puce neuromorphique pourrait consommer aussi peu que $42J$, soit deux ordres de grandeur de moins qu'avec un GPU [Joseph and Nagarajan, 2020].

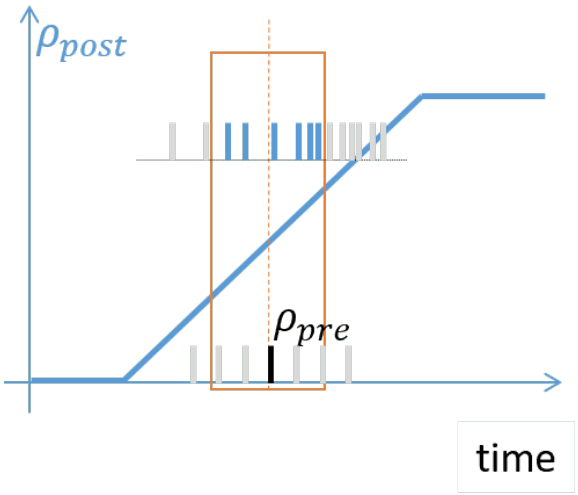
2.6 Lien entre *EqSpike* et *STDP*

Dans les sections précédentes, il a été montré que *EqSpike*, qui s'inspire de *Eq-Prop*, est en un algorithme efficace pour les circuits neuromorphiques. Cette section montre une bio-plausibilité de l'apprentissage d'*EqSpike*. [Scellier and Bengio, 2016] ont mis en évidence un lien entre la règle d'apprentissage d'*Eq-Prop* de la Figure 2.13b et le *STDP* 2.13a. La règle d'apprentissage *STDP*, illustrée dans la Figure 2.13b, renforce la corrélation entre les impulsions des neurones pré et postsynaptiques dans les réseaux avec synapses unidirectionnelles. Lors d'une impulsion postsynaptique, on calcule le temps Δt, qui correspond à la différence temporelle entre l'impulsion présynaptique et l'impulsion postsynaptique. Si $\Delta t > 0$ alors il y a corrélation et le poids synaptique est augmenté. Dans le cas contraire, si $\Delta t < 0$ alors le poids est diminué. Cette règle a été détaillée dans la section 1.3.1 du chapitre précédent.

Considérons une situation dans laquelle le neurone présynaptique génère une impulsion et le neurone postsynaptique produit une accélération, comme illustré sur la Figure 2.13b. Selon la règle d'apprentissage *Eq-Prop*, dans un réseau avec des synapses unidirectionnelles $\frac{dW_{ij}}{dt} \propto \dot{\rho}_{post}\rho_{pre}$, une mise à jour positive du poids devrait être appliquée. Dans l'exemple de la Figure 2.13b, en raison de l'accélération du neurone postsynaptique, il y a moins d'impulsions du neurone postsynaptique (2 impulsions) avant l'impulsion du neurone présynaptique qu'après (4 impulsions) dans une fenêtre temporelle. Par conséquent, $t_{post} - t_{pre}$ est positif en moyenne, ce qui donne une mise à jour de poids positive par la *STDP*.



(a) Illustration de la règle d'apprentissage *STDP*, reproduite avec les données [Bi and Poo, 2001].



(b) Illustration du lien entre *Equilibrium Propagation* et de la règle d'apprentissage *STDP*; illustration reproduite de [Bengio et al., 2017].

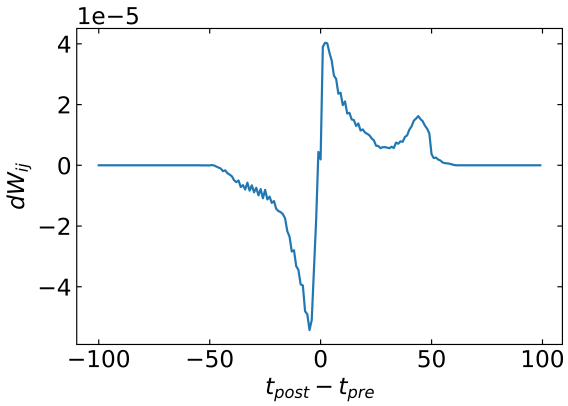
Figure 2.13 – Similarité entre la *STDP* et *Equilibrium Propagation*.

Il est intéressant de savoir si la mise à jour des poids de type *STDP* apparaissait au cours de l'apprentissage dans les simulations d'*EqSpike*, en surveillant les variations de poids dans les synapses qui relient les neurones d'entrée aux neurones de la couche cachée. Rappelons que ces synapses sont unidirectionnelles et que les neurones d'entrée ont une fréquence fixée. Les courbes ont été générées sur 100 images pendant la première époque sur l'ensemble de données *MNIST*. La courbe de la Figure 2.15 montre les mises à jour du poids moyen dans la première couche en fonction de la différence de temps entre les impulsions des neurones postsynaptiques et le temps moyen des impulsions des neurones présynaptiques dans une fenêtre de 200 pas de temps avant l'impulsion du neurone postsynaptique. La courbe obtenue, centrée sur zéro, présente en effet une forme de type *STDP*. Elle a été obtenue en filtrant les très basses fréquences inférieures à 0,05 $T_{refract}^{-1}$. Les neurones avec une fréquence faible dans la phase d'inférence induisent, en effet, d'importantes mises à jour de poids au début de la phase d'apprentissage, en raison de l'accélération soudaine de la fréquence proche de zéro à une fréquence non nulle, induisant un bruit supplémentaire dans la courbe. La différence entre l'algorithme sans et avec seuillage est illustrée entre la Figure 2.14a (sans seuillage) et la Figure 2.14b (avec seuillage).

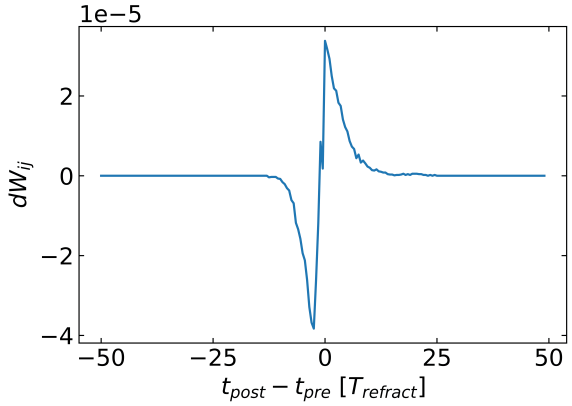
Pour plus de clarté sur la mesure de la courbe de la *STDP*, l'Algorithme 3 montre comment les courbes de *STDP* ont été générées sans le seuillage des fréquences.

La Figure 2.15 montre que l'amplitude et la fenêtre temporelle de la *STDP* varient avec la force β. Dans *Eq-Prop*, les changements de poids sont entraînés par les accélérations de la fréquence des neurones. Lorsque, pendant la phase d'apprentissage, on applique une force qui accélère la fréquence des neurones, pour des β plus grands, on obtient des changements de poids plus importants, ce qui conduit à une plus grande amplitude de la courbe de modification des poids en fonction de la différence du temps d'émission des impulsions.

Ces résultats confirment la connexion possible entre *STDP* et *Eq-Prop* soulignée dans [Scellier and Bengio, 2016], malgré le fait que les temps d'impulsions individuelles sont perdus par le calcul de la moyenne. Ils montrent qu'un comportement de type *STDP* peut être obtenu pendant l'apprentissage dans des synapses unidirectionnelles sans une implémentation explicite de la règle originale, basée sur la corrélation. Ils posent



(a) Courbe similaire à la *STDP* pendant l'apprentissage d'*EqSpike* sans seuillage basses fréquences.



(b) Courbe similaire à la *STDP* pendant l'apprentissage d'*EqSpike* avec seuillage basses fréquences.

Figure 2.14 – Comparaison du comportement d'*EqSpike* avec ou sans filtre basse fréquence.

Data : $N_{window}, T_{free}, T_{nudge}$, weight log w, spike log of each neuoron ξ
Result : λ : weight variation in temporal window centered on the pre-synaptic spike
for $t_{pre} \in [T_{free}, T_{free} + T_{nudge}]$ **do**
  for *each pre-synaptic neuron k* **do**
    if *k emits a spike at* t_{pre} **then**
      for *each connected synapse i* **do**
        $\Delta W_i \leftarrow w_i(t_{pre}) - w_i(t_{pre} - 10)$
        compute $t_{post} \leftarrow \frac{1}{N_{window}} \sum_{i=t_{pre}-N_{window}}^{t_{pre}} \xi_i * i$
        $\Delta t_{effectif} \leftarrow t_{post} - (t_{pre} - \frac{N_{window}}{2})$
        store $(\Delta W_i, \Delta t_{effectif})$
      end
    end
  end
end
for $j \in (-\frac{N_{window}}{2}, \frac{N_{window}}{2})$ **do**
  compute an average λ_j on ΔW of all couple with $\Delta t_{effectif}$ between j and j+1
end
return λ

Algorithme 3 : Algorithme pour calculer la corrélation entre *EqSpike* et la *STDP*

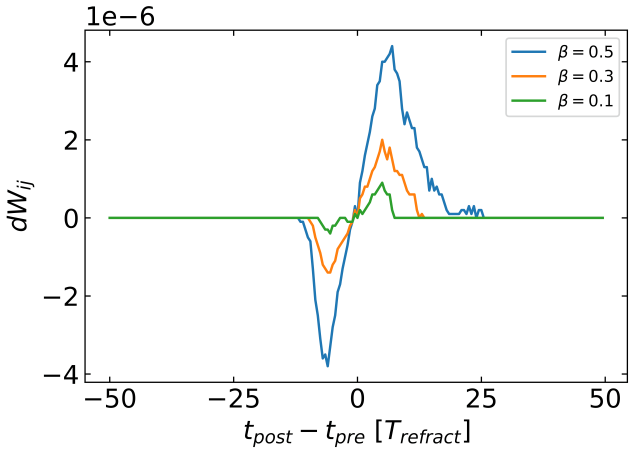


Figure 2.15 – Courbe similaire à la *STDP* pendant l'apprentissage d'*EqSpike* avec différents β.

également la question de savoir si les réseaux multicouches peuvent être entraînés avec la règle d'apprentissage *STDP* locale en utilisant la phase d'apprentissage d'*Eq-Prop*, éventuellement dans la version étendue appelée *Vector-field Equilibrium Propagation* dans laquelle les synapses sont unidirectionnelles [Scellier et al., 2018].

2.7 Conclusion

Dans ce chapitre, nous avons présenté un nouvel algorithme pour les réseaux de neurones à impulsions avec des entrées statiques, *EqSpike*, compatible avec des systèmes neuromorphiques, et obtenu une précision de 97.6% pour la reconnaissance des chiffres manuscrits *MNIST*, précision du même ordre de grandeur que l'état de l'art *BPTT* pour cette taille de réseau. Nous montrons que *EqSpike* met en œuvre la règle d'apprentissage d'*Eq-Prop* de manière locale et intrinsèque. Les gradients d'erreur sont calculés par la dynamique du système grâce à l'ajout d'un seul bloc au neurone. Et les poids sont modifiés par chaque impulsion des neurones connectés. Ceci peut conduire à des réseaux de neurones matériels, à impulsions, qui n'ont pas besoin d'un circuit externe pour calculer les gradients d'erreur donnés par la rétro-propagation du gradient et apprennent de manière autonome, simplement en présentant des entrées et en perturbant les sorties en fonction des erreurs.

Le nombre de SynOps pour obtenir ces résultats sur le jeu de données *MNIST* montre que la consommation d'énergie, pour *EqSpike*, est théoriquement inférieure de deux ordres de grandeur à celle d'un *GPU* pour l'entraînement pour le même temps de calcul avec des neurones haute fréquence, passant de $4000J$ à $42J$. Le gain d'énergie pour l'inférence est de 3 ordres de grandeur comparé au GPU, passant de $35J$ à $15mJ$ pour 10000 images. L'inférence, après l'entraînement du réseau, peut être accélérée en considérant uniquement la première impulsion de sortie, plutôt que de considérer le neurone avec le plus d'impulsions à la sortie. Ce qui fait passer le temps d'inférence à $t = \frac{10}{f_{max}}$ plutôt que $\frac{75}{f_{max}}$, avec une perte de précision de moins de 2.4%. Ces calculs ont été réalisés avec des neurones et des synapses CMOS.

Enfin, nous avons montré que les mises à jour des poids d'*EqSpike* présentent des similitudes avec la règle d'apprentissage *STDP* pendant l'apprentissage, ce qui renforce la bio-plausibilité d'*EqSpike*. Cela pourrait permettre d'implémenter *EqSpike* avec des synapses dans du matériel neuromorphique qui présente un comportement de type *STDP* pour obtenir une plus faible consommation d'énergie et une plus grande densité de surface, par exemple les memristors. [Querlioz et al., 2013].

Dans le chapitre suivant, nous allons voir comment adapter *EqSpike* à des synapses réalisées avec des memristors, permettant un gain énergétique espéré encore plus important que des synapses CMOS.

Chapitre 3

Adaptation de *EqSpike* pour l'utilisation de memristors comme synapses

Dans ce chapitre, il sera présenté comment implémenter *EqSpike* avec des memristors pour synapses afin d'aller plus loin dans la réduction de l'énergie des réseaux de neurones physiques. Dans un premier temps, nous étudierons les capacités du réseau avec des memristors idéaux. De plus, les memristors étant une technologie moins mature que les transistors, les composants souffrent d'une variabilité interne importante. Dans un second temps, la robustesse de l'algorithme à différentes sources de variabilité sera étudiée.

3.1 Introduction

Grâce à *EqSpike*, il est possible d'avoir un algorithme d'apprentissage intrinsèque qui est compatible avec une implémentation matérielle. Dans le chapitre 2, nous avons présenté l'algorithme sans rentrer dans les détails d'une implémentation physique pour rester générique au type de composant utilisé. Il est nécessaire pour concevoir une éventuelle future puce neuromorphique d'adapter *EqSpike* aux composants utilisés pour les synapses afin de profiter de la modification locale des synapses.

Les puces neuromorphiques les plus avancées utilisent des transistors pour les synapses et les neurones (section 1.4). Cependant, pour réaliser physiquement un neurone ou une synapse, il faut des dizaines de transistors. Pour aller plus loin dans la réduction d'énergie et l'augmentation de la densité des puces neuromorphiques, il est intéressant d'utiliser des nanocomposants émergents comme présenté dans la section 1.4.1. Dans cette situation, il est nécessaire de choisir un composant qui s'adapte bien à l'algorithme *EqSpike*. Pour cela, nous avons besoin d'un composant qui peut être programmé par des impulsions afin que les neurones à impulsions modifient d'eux-mêmes le poids synaptique et ainsi préserver l'aspect intrinsèque de l'apprentissage. Ce composant doit aussi être non volatile, c'est-à-dire que la valeur physique du poids synaptique ne doit pas changer sans stimulation externe. Nous nous concentrons en priorité sur les synapses avec des composants émergents, car elles sont souvent bien plus nombreuse que les neurones, dans un réseau de neurones.

Un composant prometteur pour réaliser des synapses matérielles est le memristor. Comme montré dans l'état de l'art, ce composant est programmable par une impulsion en tension. Ce composant est intéressant pour sa consommation d'énergie, sa non-volatilité et son implémentation dans des réseaux matriciels.

Dans ce chapitre, nous proposons une adaptation de *EqSpike* pour un réseau avec des memristors comme synapses. Nous montrerons un schéma d'implémentation possible de ce réseau, en utilisant un réseau matriciel de memristors et des neurones *LIF* (implémentables en *CMOS*). Les memristors ayant une variance plus importante que les transistors CMOS, nous étudierons la résistance du réseau à la variabilité intra et intercomposants.

3.2 Fonctionnement des memristors

Un memristor est un composant non-volatile dont la résistance peut être programmée par des impulsions en tension. La conductance d'un memristor, G, encode le poids synaptique dans un réseau de neurones. La variation de conductance, ΔG, dépend de l'amplitude et de la durée des impulsions de tension appliquées aux bornes du composant. Un modèle simplifié et général, qui s'est avéré bien reproduire le comportement de la modification des poids synaptiques de memristors basé sur des matériaux différents [Querlioz et al., 2011], est :

$$\frac{\Delta G}{t_{pulse}} = \begin{cases} f^+(V)e^{\eta^+ \frac{G-G_{min}}{G_{max}-G_{min}}}, & \text{si } V > 0. \\ f^-(V)e^{\eta^- \frac{G_{max}-G}{G_{max}-G_{min}}}, & \text{si } V < 0. \end{cases} \tag{3.1}$$

G_{min} et G_{max} définissent la conductance minimale et la conductance maximale que le composant peut atteindre. $\eta^\pm$ est un facteur de non-linéarité représentant le fait que l'amplitude des changements de conductance peut dépendre de l'état de conductance du memristor au moment de la modification et t_{pulse} est la durée de l'impulsion de tension appliquée. f^+ et f^- sont des fonctions monotones qui dépendent de V, la tension appliquée aux bornes du memristor.

Pour pouvoir modéliser et simuler les memristors, nous avons simplifié les fonctions $f^\pm$. Nous les considérons comme égales à 0 jusqu'à un seuil $V_{th}^\pm$, puis croissante avec un taux de croissance S^+ pour f^+ ou décroissante avec un taux de décroissance S^- pour f^-. Ce modèle, repris de [Querlioz et al., 2011], est donné par :

$$\frac{\Delta G}{t_{pulse}} = \begin{cases} S^+(V - V_{th}^+)e^{\eta^+ \frac{G-G_{min}}{G_{max}-G_{min}}}, & \text{si } V > V_{th}^+ > 0. \\ S^-(V - V_{th}^-)e^{\eta^- \frac{G_{max}-G}{G_{max}-G_{min}}}, & \text{si } V < V_{th}^- < 0. \\ 0, & \text{sinon} \end{cases} \tag{3.2}$$

l'Équation (3.2), est exponentiellement non-linéaire et asymétrique. Les pentes, les facteurs de non-linéarité et les seuils sont différents pour chaque memristor. Leur variance est suffisamment importante pour impacter la précision d'un réseau de neurones réalisé avec ces composants (par exemple [Querlioz et al., 2013], plus de détails section 1.4.3). Elle doit donc être prise en compte dans l'implémentation.

Pour illustration, la Figure 3.1 montre comment la conductance, G, évolue en fonction de la tension appliquée aux bornes du memristor linéaire ($\eta^{\pm} = 0$). Si la tension appliquée est suffisamment élevée pour franchir le seuil du memristor, noté V_{th}^{+} pour la variation positive et V_{th}^{-} pour la variation négative, la conductance du memristor est modifiée d'une manière qui dépend linéairement de la tension. En l'absence de franchissement du seuil, la valeur de la conductance n'est pas modifiée.

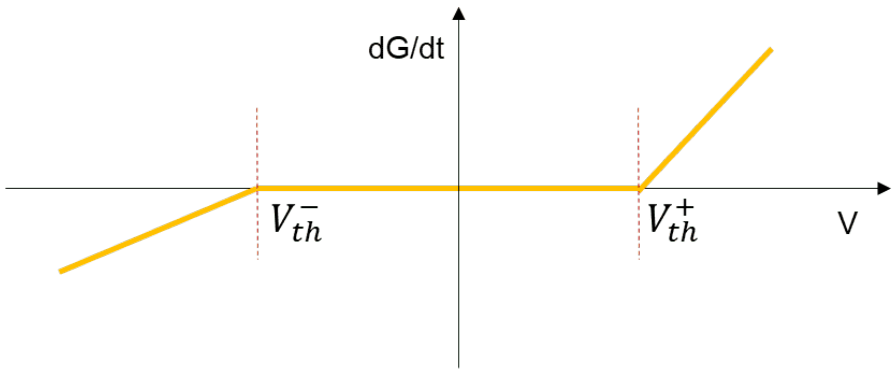


Figure 3.1 – Illustration de la modification de la conductance d'un memristor avec un facteur de non-linéarité nul, $\eta^{\pm} = 0$, en fonction de la tension appliquée à ses bornes.

Lors de l'entraînement d'un réseau de neurones, les poids synaptiques prennent généralement des valeurs positives et négatives. Il est donc nécessaire de reproduire ce comportement dans une implémentation avec des memristors. Une conductance ne pouvant pas être négative, une implémentation possible consiste à utiliser deux memristors [Chang et al., 2018], dont les conductances sont notées G^{+} et G^{-}, et à réaliser le poids synaptique avec :

$$G = G^{+} - G^{-} \tag{3.3}$$

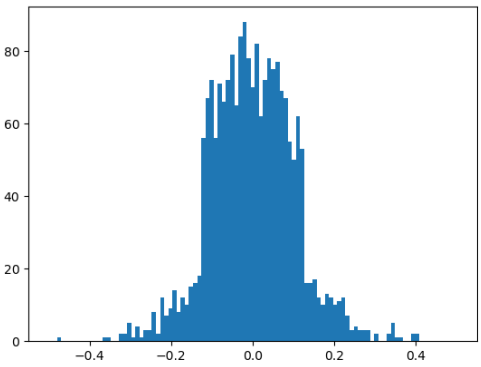
En pratique, il est possible soit de modifier les deux memristors, ce qui accroît les plages de conductance possibles, mais augmente les possibles erreurs d'écriture, soit de modifier un seul memristor et de fixer l'autre à une valeur intermédiaire, ce qui réduit la plage de valeurs, mais réduit aussi les erreurs de modification du memristor.

3.3 Memristors comme synapses dans *EqSpike*

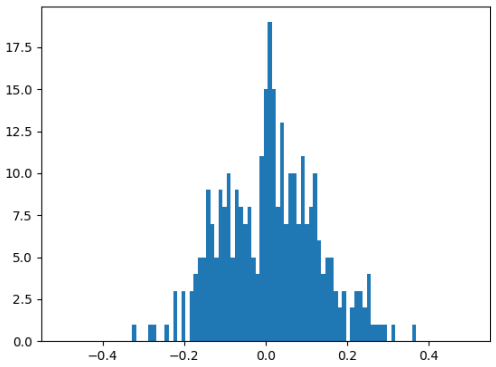
Dans un premier temps, les memristors seront considérés comme des composants idéaux, avec des comportements similaires pour des tensions appliquées positives et négatives ($S^{+} = S^{-}$, $V_{th}^{+} = V_{th}^{-}$) et des changements linéaires de conductance ($\eta^{\pm} = 0$). Ensuite, dans la section 3.7, les paramètres des memristors seront modifiés pour étudier la résistance aux non-idéalités des réseaux de neurones à base de nanocomposants.

Les conductances de différents types de memristors peuvent varier de plusieurs ordres de grandeur selon les différentes technologies de réalisation, mais sont toujours bornées. Il est donc nécessaire de borner les poids synaptiques dans les simulations pour bien représenter les conductances des memristors.

Pour fixer les bornes des poids synaptiques, nous avons regardé l'évolution des poids sur un réseau de neurones sans memristor après apprentissage d'une tâche avec *EqSpike*. Nous avons entraîné le réseau de neurones sur la base d'images *Digits* avec les paramètres utilisés dans le chapitre 2 sur 50 époques. Nous avons analysé la distribution des poids synaptiques à la fin des 50 époques, telle que présentée sur les Figures 3.2a et 3.2b.



(a) Distribution des poids sur la première couche du réseau de neurones après 50 époques d'entraînement sur le jeu de données *Digits* avec *EqSpike*.



(b) Distribution des poids sur la dernière couche du réseau de neurones après 50 époques d'entraînement sur le jeu de données *Digits* avec *EqSpike*.

Figure 3.2 – Distribution des poids pour les deux couches de synapses après 50 époques d'entraînement sur le jeu de données *Digits* avec *EqSpike*.

Pour les deux couches de neurones, la distribution des poids s'apparente à une gaussienne centrée en 0 et la majorité des poids sont compris dans l'intervalle $[-0.5, 0.5]$. Nous avons donc choisi cet intervalle pour la représentation des poids avec des memristors. Nous avons normalisé la conductance par rapport à la plage de conductances possibles du memristor $G_{max} - G_{min}$. Pour être le plus robuste possible à la variation des composants, nous avons choisi de représenter les poids négatifs en fixant G^- à une valeur intermédiaire et de faire varier G^+. Il faut donc que la conductance normalisée de chaque memristor se situe dans un intervalle $[0, 1]$. Les poids synaptiques W, qui correspondent à la différence normalisée des deux conductances, seront compris dans les simulations de ce chapitre entre $[-0.5, 0.501[$ avec

$$W = \frac{G}{G_{max} - G_{min}} - 0.5 \tag{3.4}$$

en considérant $G_{min} = 0.1\%$ de G_{max} et $G_{max} = 100\%$. Cela correspond à une conductance modifiable G^+ et à une conductance constante G^- choisie au milieu de sa plage de valeurs.

L'implémentation d'*EqSpike* avec des synapses mettant en œuvre des memristors est basée sur l'utilisation de la superposition des impulsions des neurones pour modifier le poids synaptique (conductance du memristor) selon sa loi d'apprentissage (Équation (2.6)). Deux neurones i et j connectés par une synapse, vont envoyer des impulsions de tension lors de l'exécution de l'algorithme *EqSpike* et modifier la tension aux bornes du memristor selon : $V_{ij} = V_i - V_j$. Avec une tension supérieure au seuil d'écriture du memristor, sa conductance sera modifiée en accord avec l'Équation (3.2).

Pour réaliser cette modification de conductance, pendant la phase d'apprentissage, nous avons proposé l'implémentation suivante. Chaque neurone envoie deux impulsions successives : la première impulsion, que nous appellerons *impulsion de communication*, permet de transmettre l'information et présente une tension faible, par rapport au seuil, pour ne pas écrire le memristor. La seconde impulsion, que nous appelons *impulsion de programmation*, est de tension égale au seuil du memristor, positivement puis négativement pendant un temps t_{pulse}. Une tension égale au seuil permet de ne pas modifier la conductance du memristor avec l'impulsion uniquement, tout en permettant que la superposition avec un autre signal puisse le reprogrammer. La superposition de cette dernière avec un signal proportionnel à $\dot{\rho}$, envoyé par l'autre neurone, provoque une modification du poids synaptique (conductance du memristor) selon l'Équation 2.6, rappelée ici :

$$\frac{dW_{ij}}{dt} \sim \dot{\rho}_j \rho_i + \dot{\rho}_i \rho_j$$

Le comportement adopté avec l'impulsion de programmation et le signal $\overline{\rho}$ en fonction du temps est illustré schématiquement dans la Figure 3.3. Le signal du haut correspond à un neurone i envoyant une impulsion de communication, faible en tension comparé à V_{th}, suivie d'une impulsion de programmation,

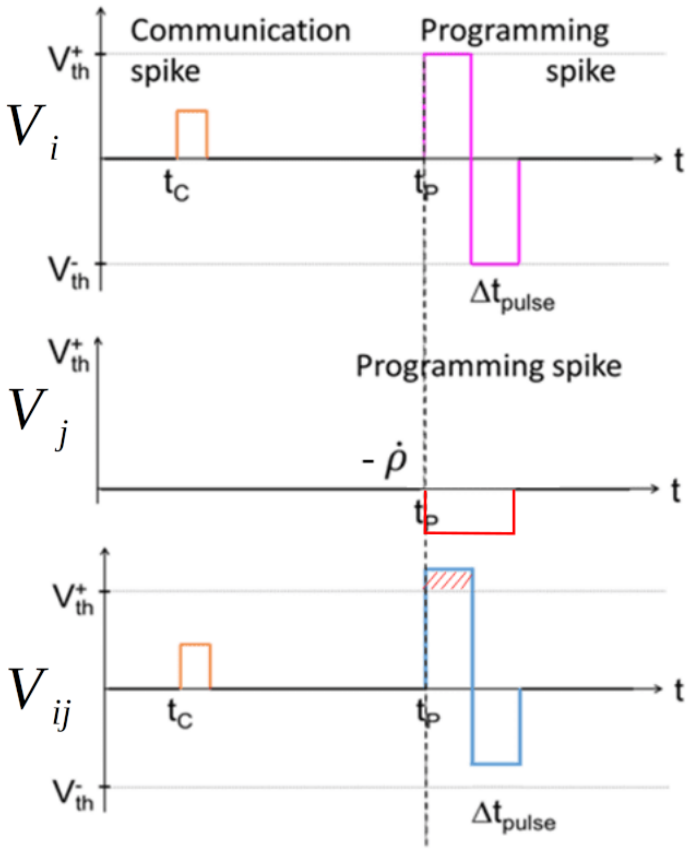


Figure 3.3 – Superposition des impulsions en fonction du temps lors d'une communication entre deux neurones. En haut, le neurone i envoyant une impulsion de communication suivie d'une impulsion de programmation. Le signal du milieu représente le neurone j envoyant son estimation de la dérivée. En bas, le signal aux bornes du memristor, $V_{ij} = V_i - V_j$, avec une impulsion de communication et l'impulsion de programmation soustraite à la dérivée de la fréquence envoyée par le neurone j.

d'amplitude V_{th} puis $-V_{th}$. Le signal du milieu correspond à un neurone j envoyant son estimation de la dérivée pendant l'impulsion de programmation de l'autre neurone au même moment que le neurone i envoie son impulsion de programmation. En bas est représenté le signal aux bornes du memristor, V_{ij}. La tension V aux bornes du memristor pendant l'impulsion de programmation du neurone connecté est la différence de la sortie du bloc $\overline{\dot{\rho}_i}$, qui estime la dérivée de la fréquence et de la tension de l'impulsion de programmation d'amplitude $V_{th}^{\pm}$, ce qui donne :

$$V_{ij} = \overline{\dot{\rho}_i} + V_{th}^{\pm} \tag{3.5}$$

L'Équation 2.10 du chapitre précédent montre $\overline{\dot{\rho}} = \dot{\rho}\frac{\tau}{\gamma_{LI}}\gamma_{lr}$, ce qui donne :

$$V_{ij} = \dot{\rho}_i \frac{\tau}{\gamma_{LI}}\gamma_{lr} + V_{th}^{\pm}, \tag{3.6}$$

avec τ la constante de temps pour estimer la dérivée, γ_{LI} le facteur de décroissance de l'intégrateur du bloc $\overline{\dot{\rho}_i}$, $\dot{\rho}_i\frac{\tau}{\gamma_{LI}}$ le signal de l'estimation de la dérivée et γ_{lr} le taux d'apprentissage.

En remplaçant V dans l'Équation 3.2 par l'Équation 3.6, on obtient la règle d'apprentissage suivante :

$$\Delta W = \frac{S^+}{G_{max} - G_{min}}(\dot{\rho}\frac{\tau}{\gamma_{LI}}\gamma_{lr} + V_{th}^{+} - V_{th}^{+}) \cdot t_{pulse} \cdot e^{\eta^+ \frac{G-G_{min}}{G_{max}-G_{min}}} \tag{3.7}$$

Avec l'hypothèse de connaître le seuil exact du memristor pour que l'impulsion de programmation soit à la valeur V_{th}, nous utiliserons un facteur ϕ_{lr} permettant de normaliser l'équation des memristors pour être indépendant de la valeur de certains des paramètres tel que :

$$\phi_{lr} = \frac{S^{\pm} \cdot t_{pulse}\gamma_{lr}}{G_{max} - G_{min}} \tag{3.8}$$

On obtient alors :

$$\Delta W = \phi_{lr}\dot{\rho}\frac{\tau}{\gamma_{LI}} \cdot e^{\eta^+ \frac{G-G_{min}}{G_{max}-G_{min}}} \tag{3.9}$$

où ϕ_{lr} est un paramètre constant qui dépend du taux d'apprentissage et peut être réalisé électroniquement par un amplificateur ou un diviseur de tension à la sortie du bloc $\overline{\dot{\rho}}$.

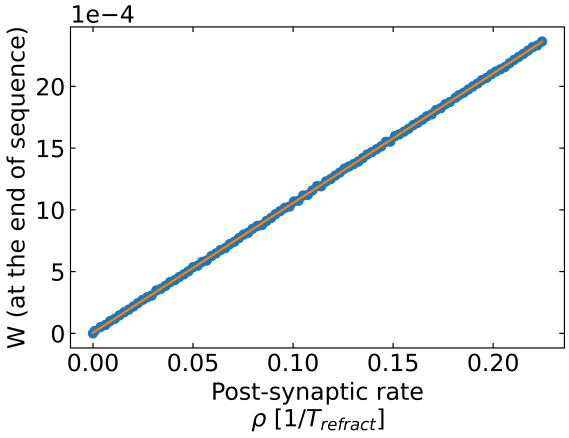
Grâce à l'impulsion de programmation, au bloc $\overline{\dot{\rho}}$ et à la synchronisation des impulsions, il est théoriquement possible d'implémenter *EqSpike* avec des memristors.

3.4 Vérification de la loi d'apprentissage

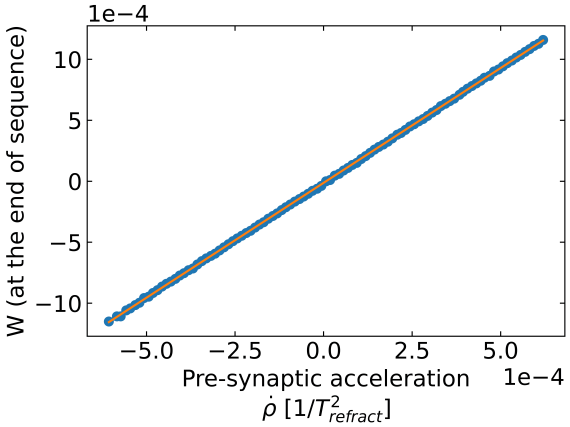
Avant de simuler intégralement le réseau de neurones à impulsions avec des memristors comme synapses, il faut vérifier que dans le cas de deux neurones connectés par une synapse, la loi d'apprentissage est bien approximée. Le protocole est le même que pour vérifier la loi d'*EqSpike* du chapitre 2. Les fréquences et, respectivement, les accélérations de chaque neurone sont fixées alternativement, en laissant le poids synaptique se mettre à jour grâce à la loi d'apprentissage de l'algorithme *EqSpike*. Le poids final au temps $600dt$ est mesuré et reporté sur la Figure 3.4 pour chaque configuration de simulation. La différence comparée à la méthode du chapitre précédent est l'application des équations des memristors pour la modification des poids.

Dans cette expérience, le memristor est considéré comme parfait, symétrique et linéaire. L'impact des non-idéalités est étudié dans la section 3.7. Les neurones *LIF* sont pilotés par un signal d'entrée contrôlé pendant une durée fixe T_{test} durant laquelle ils émettent des impulsions à une fréquence proportionnelle au signal d'entrée.

La Figure 3.4a montre la dépendance de la variation du poids avec différentes fréquences postsynaptiques et avec une seule accélération de la fréquence pour le neurone présynaptique. Dans la Figure 3.4b, nous montrons la dépendance de la variation du poids avec les différentes accélérations du neurone présynaptique et une fréquence postsynaptique. Les deux figures montrent une évolution linéaire, passant par 0 avec une fréquence ou une dérivée de fréquence nulle. La valeur obtenue pour une fréquence de $0.2\ T_{refract}^{-1}$ avec



(a) Dépendance de la modification du poids pour différentes fréquences ρ.



(b) Dépendance de la modification du poids pour différentes accélérations de la fréquence $\dot{\rho}$.

Figure 3.4 – Vérification de la dépendance aux paramètres de la loi d'apprentissage d'*EqSpike* avec des memristors idéaux comme synapses.

une accélération de $5 \cdot 10^{-4}\ T_{refract}^{-2}$ avec les paramètres $\gamma_{LI} = 0.5, \tau = 50$ et $\phi_{lr} = 0.0002$ pendant $time = 100 T_{refract}$ doit être de $\rho_i \cdot \dot{\rho}_j \frac{\tau}{\gamma_{LI}} \cdot \phi_{lr} \cdot time = 0.2 \cdot 5 \cdot 10^{-4} \frac{50}{0.1} \cdot 200 \cdot 0.0002 = 20 \cdot 10^{-4}$. De même, les valeurs prévues lors de la vérification de la dépendance à l'accélération, avec une fréquence ρ deux fois plus faible, soit de 10 et -10, en fonction du signe de l'accélération. Les valeurs théoriques et celles obtenues sur les Figures 3.4a et 3.4b sont proches, ce qui montre une bonne application de la loi d'apprentissage d'*EqSpike*. De plus, les figures sont fortement similaires à celles d'*EqSpike* sans memristors (à un facteur près dû à un taux d'apprentissage différent). Ces Figures 3.4 confirment que la règle d'apprentissage *EqSpike* est satisfaite sur les plages de fréquences testées. La règle d'apprentissage complète est obtenue grâce à la symétrie du dispositif et à la bidirectionnalité du réseau. Cette symétrie a également été vérifiée, mais n'est pas montrée, car les graphiques sont identiques en tous points.

3.5 Construction d'un circuit neuronal avec des memristors

Maintenant que nous avons montré que l'algorithme *EqSpike* s'adapte aux memristors, nous proposons un schéma permettant l'implémentation physique d'un tel réseau de neurones qui a donné suite à un dépôt de brevet *[FR2101311]*.

Pour obtenir une couche d'un réseau de neurones, les memristors sont connectés dans un réseau matriciel (*crossbar array*, en anglais), comme présenté dans la Figure 1.14 du chapitre 1.4.1, ce qui permet le calcul de toutes les sommes pondérées pour les neurones de la couche de sortie en un seul passage. Pour effectuer des poids négatifs, deux *crossbar array* sont utilisés et les valeurs des tensions sont soustraites de telle façon que :

$$o_j = \sum_i G_{ij}^+ V_i - \sum_i G_{ij}^- V_i \tag{3.10}$$

Avec o_j la somme pondérée pour le neurone de sortie j.

Les memristors pondèrent directement les signaux de tension appliqués par les neurones d'entrée, d'une manière compacte et économe en énergie, avantageuse pour des circuits neuromorphiques [Fouda et al., 2019a, Burr et al., 2017].

La Figure 3.5 montre un schéma fonctionnel du circuit que nous proposons afin d'implémenter un réseau de neurones à impulsions avec la règle d'apprentissage d'*EqSpike* (Équation 2.6). Nous ne présentons qu'une seule couche pour des raisons de clarté, mais cette implémentation est extensible sur un réseau profond avec le même schéma répété pour chaque couche.

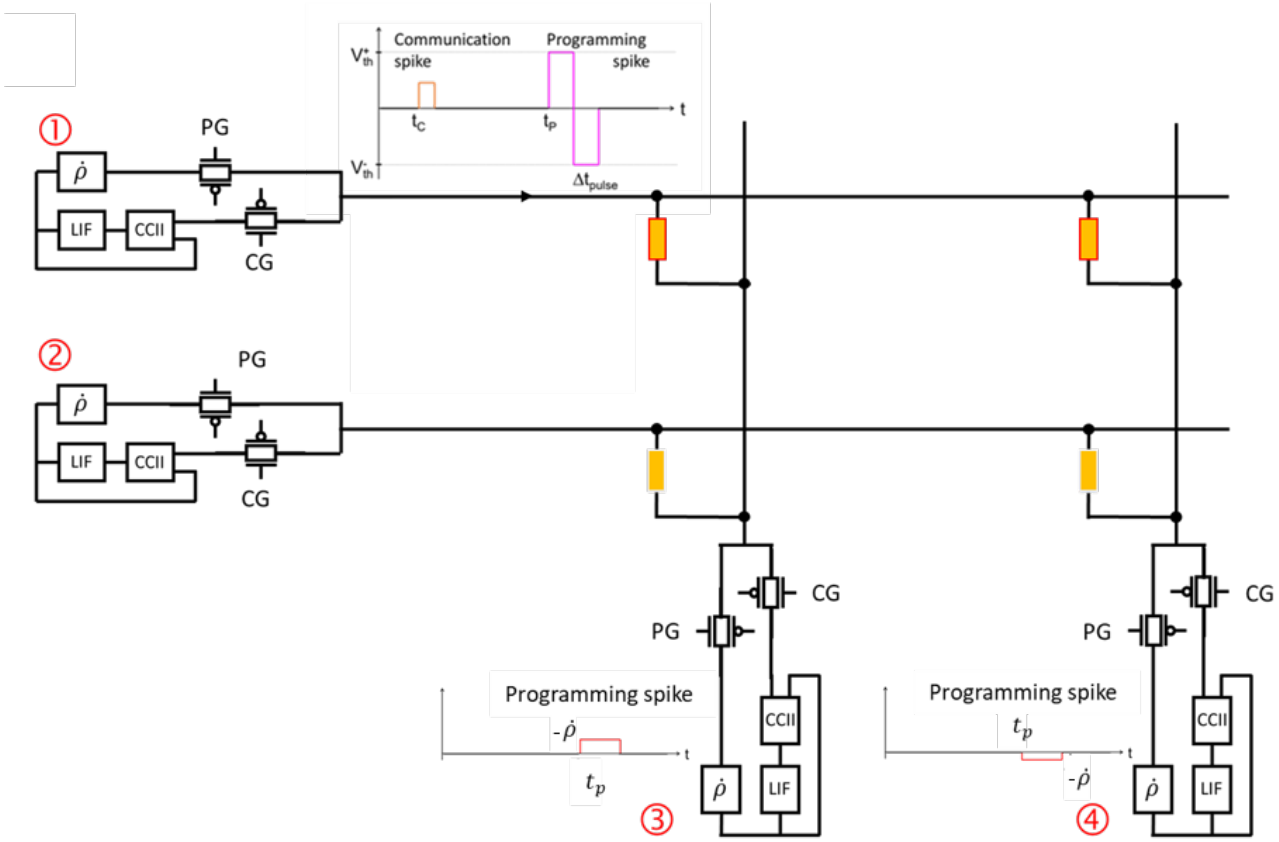


Figure 3.5 – Schéma électrique d'une couche 2x2 de neurones avec des memristors comme synapses. Les blocs 1 et 2 implémentent les neurones d'entrée, les blocs 3 et 4 sont les neurones de sortie, et les memristors sont indiqués par les rectangles jaunes qui connectent les neurones. Grâce à l'impulsion du neurone 1 et aux signaux $\dot{\rho}$ des neurones 3 et 4, les synapses encadrées en rouge sont modifiées.

Les memristors, comme expliqué dans la section précédente, connectent les neurones *LIF* qui possèdent un convoyeur de courant (*CCII*) et un bloc $\dot{\overline{\rho}}$ pour calculer la dérivée de leur fréquence, ainsi que des interrupteurs de programmation *PG* et *CG*. Dans la Figure 3.5, nous présentons une implémentation possible de la couche neuronale, pour un exemple illustratif de 2x2 neurones.

Comme présenté ultérieurement, les neurones *LIF* intègrent le courant qu'ils reçoivent. Ils émettent des impulsions lorsque leur potentiel de membrane atteint un seuil, ensuite le potentiel de membrane est réduit de la valeur du seuil de membrane, et ils subissent une période réfractaire pendant laquelle ils n'émettent aucune impulsion.

Le bloc nommé "$\dot{\overline{\rho}}$" prend en entrée le train d'impulsions émis par le neurone *LIF* auquel il est connecté et émet une impulsion proportionnelle à la dérivée de la fréquence d'impulsion.

Les convoyeurs de courant (*CCII* dans la Figure 3.5) assurent que les signaux émis par les neurones se propagent de manière bidirectionnelle [Payvand et al., 2020, Scellier and Bengio, 2016], comme cela est nécessaire pour *EqSpike*. Un *CCII* permet d'envoyer l'impulsion du neurone sur la couche suivante et d'imposer la valeur du neurone sur la couche précédente. S'il reçoit une impulsion, le *CCII* transmet à son neurone le signal des neurones connectés à son entrée.

Le transistor *CG* est activé quand le neurone intègre un signal ou émet une impulsion. Si une couche de neurones émet une impulsion de programmation, la couche connectée ferme leurs transistors *CG* et ouvre leurs transistors *PG* pour envoyer le signal du bloc $\dot{\overline{\rho}}$ et ainsi écrire les memristors.

Pendant l'inférence, correspondant à la phase de l'algorithme où les neurones de sortie sont libres et le réseau laissé se relaxer jusqu'à l'équilibre, les neurones *LIF* émettent une unique impulsion, appelé impulsion de communication dans la Figure 3.3 pour le neurone i, à chaque fois qu'ils doivent émettre une impulsion. Le but de cette impulsion est de mettre à jour tous les neurones *LIF* j auxquels le neurone i est connecté, signal pondéré par la conductance des memristors. Il consiste en une courte impulsion positive d'une amplitude bien inférieure au seuil de programmation du memristor. Lorsqu'elle est émise, un signal est envoyé aux grilles des transistors *CG* des neurones j pour les rendre conducteurs. Un courant circule à l'entrée des neurones *LIF* j, égal à la valeur de la tension émise par i, multipliée par la conductance des memristors. Le courant est ensuite intégré par les neurones j.

Lors de la phase d'apprentissage de l'algorithme, pendant laquelle les neurones de sortie sont tirés vers un autre état et les synapses mises à jour, une impulsion de programmation, V_{prog}, suit l'impulsion de communication. L'impulsion met à jour toutes les synapses auxquelles le neurone i est connecté comme présenté plus tôt. Pour programmer les memristors, un signal est appliqué aux transistors *PG* des neurones j pour permettre au signal $\dot{\overline{\rho}}$ de passer.

Ce schéma est générique et peut être réalisé avec différents types de neurones et paramètres internes pour ainsi s'adapter aux différents memristors. Cette implémentation présente l'avantage de rajouter la complexité uniquement dans les neurones, moins nombreux que les synapses.

De plus, pour minimiser l'effet des trajets parasites de courant (*sneak path*, en anglais), qui est un problème connu de ce type d'implémentation [Fouda et al., 2019a], ce schéma peut être implémenté de manière simple avec un transistor par composant pour construire un système plus robuste. Cette implémentation, grâce à l'apprentissage intrinsèque permis par *EqSpike*, permet d'utiliser seulement un transistor par synapse, moins que d'autres implémentations neuromorphiques avec un apprentissage supervisé, comme *BP* qui peuvent atteindre 8 transistors par synapse [Payvand et al., 2020].

3.6 Choix des hyperparamètres pour une implémentation d'*EqSpike* avec memristors

Dans toute implémentation d'apprentissage avec des réseaux de neurones, il y a un certain nombre d'hyperparamètres à régler pour aboutir à une performance acceptable. Nous nous concentrons maintenant sur le choix d'hyperparamètres d'*EqSpike*, en particulier la durée de la phase d'apprentissage et le taux d'apprentissage. Ce sont des paramètres clés pour trouver un compromis satisfaisant entre la précision, la vitesse et la consommation d'énergie. Dans l'implémentation proposée dans ce chapitre, l'impulsion

de programmation utilise plus d'énergie que l'impulsion de communication, en raison de son amplitude supérieure. Nous nous sommes ainsi concentrés sur la réduction du nombre d'impulsions de programmation pour réduire l'énergie consommée. Un des paramètres qui a le plus d'influence sur la vitesse de calcul et la consommation d'énergie, est la durée de la phase d'apprentissage, car le nombre d'impulsions de programmation est proportionnel à cette durée. Dans ce qui suit, nous cherchons à définir une valeur acceptable pour cet hyperparamètre avant d'injecter des non-idéalités dans le modèle des synapses et des non-idéalités dans les neurones *LIF*.

Un autre hyperparamètre important dans l'entraînement d'un réseau neuronal est le taux d'apprentissage. Il régit l'étape de mise à jour des poids, et dans notre cas, les changements de conductance des memristors, en les modifiant progressivement avec seulement une fraction de l'erreur réelle. Les memristors réels ont une faible précision, entre deux états et 128 états [Xi et al., 2021]. Il est donc préférable que le pas de temps de mise à jour du poids soit aussi grand que possible pour s'adapter au nombre limité de valeurs de conductance possibles. Le pas de changement de conductance peut être réglé par le biais du taux d'apprentissage comme suit : en augmentant ce dernier, le pas de changement de poids synaptique augmente et permet donc d'utiliser un memristor moins précis. En général, un memristor optimisé présente une précision d'environ $4-7$ bits [Xi et al., 2021], ce qui est beaucoup moins que l'opération en virgule flottante de 32 bits ou 64 bits disponible dans les processeurs généralistes. Dans ce chapitre, il ne sera pas présenté de technique pour adapter *EqSpike* à la précision de 8 bits ou moins, mais les taux d'apprentissage seront réglés pour les rendre pertinents pour des memristors de faible précision.

Afin de définir une précision de base pour l'implémentation memristive d'*EqSpike*, les memristors sont d'abord considérés parfaits (symétriques, avec une pente et un seuil connus, linéaires et identiques). La durée de la phase d'apprentissage et le nombre de mises à jour des poids synaptiques seront variés, dans un intervalle prédéterminé pour chaque paramètre, pour trouver une valeur plus adéquate pour le compromis précision/énergie. Une fois qu'une valeur satisfaisante a été trouvée, cette valeur sera fixée pour les simulations suivantes en prenant en compte les non-idéalités de memristors, ainsi que celles des neurones.

3.6.1 Recherche d'une durée optimale de la phase d'apprentissage

Pour réduire la consommation énergétique globale du réseau neuronal, il est important de réduire le nombre d'impulsions de programmation et de communication à chaque itération. Une méthode simple pour réduire le nombre d'impulsions est de réduire le temps de la phase d'apprentissage. En effet, cette phase est la plus consommatrice en énergie, car les impulsions nécessaires pour mettre à jour le memristor (les impulsions de programmation) sont plus consommatrices en énergie que les impulsions de communication du fait de leur tension plus importante. Dans *EqSpike*, pour éviter d'avoir à mesurer la convergence de l'état des neurones *LIF*, et pour réduire la consommation d'énergie, le nombre de pas de temps pendant la phase d'apprentissage peut être prédéfini, ce qui est déjà utilisé pour plusieurs implémentations de *EqProp* [Laborieux et al., 2021, Ernoult et al., 2020].

Le jeu de données utilisé pour les expériences de ce chapitre est la base d'images de chiffres manuscrits *Digits* à cause du temps de calcul trop long sur des jeux de données plus gros, comme *MNIST*, qui empêcherait d'avoir suffisamment de résultats. La précision de reconnaissance obtenue sur ce jeu de données avec des composants idéaux sera le point de comparaison pour l'algorithme *EqSpike*.

La Figure 3.6, montre l'évolution de la précision de reconnaissance sur *Digits* en fonction de la durée de la phase d'apprentissage, avec les paramètres du tableau 3.1 et un réseau de taille $64 \times 32 \times 10$. Chaque point est le résultat d'une moyenne sur 10 exécutions de 30 époques chacune. Les poids sont initialisés en utilisant l'initialisation de Xavier [Glorot and Bengio, 2010]. La précision augmente rapidement quand le temps de la phase d'apprentissage augmente jusqu'à un temps de $20\ T_{refract}$ puis augmente lentement ensuite. Une augmentation du temps de la phase d'apprentissage au-delà de $62.5\ T_{refract}$ (point rouge sur la figure) n'augmente pas significativement la précision alors que la consommation d'énergie augmente proportionnellement (environ 5 fois moins d'énergie comparé à 90 époques et un $T_{nudge} = 150$). Pour les prochaines expériences, la durée de la phase d'apprentissage est, par conséquent, fixée à $62.5\ T_{refract}$ avec une précision de 96.95%. Cette durée est la plus adéquate pour ce réseau et cet ensemble de données. Pour d'autres topologies et/ou jeux de données, l'analyse du temps d'apprentissage le plus efficace doit être ajustée en conséquence.

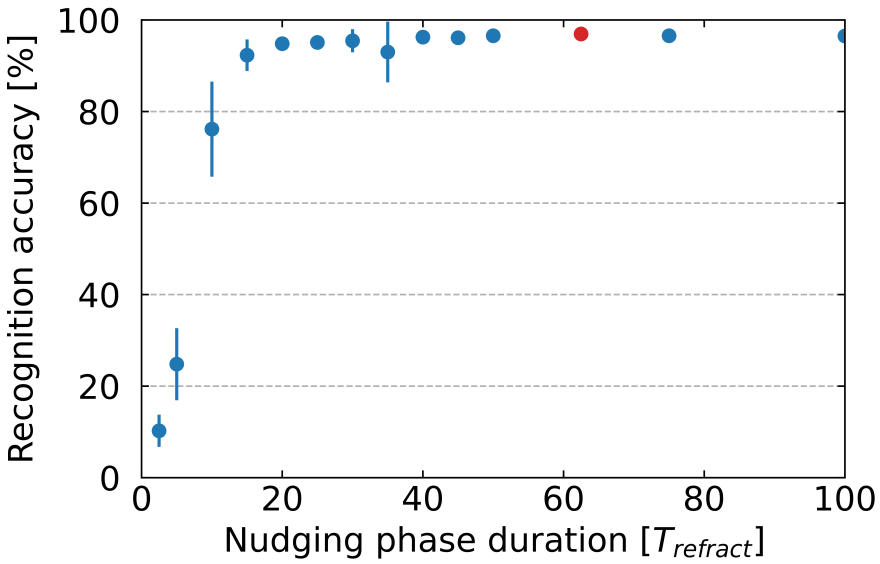


Figure 3.6 – Précision de classification sur la base de test *Digits* en fonction de la durée de la phase d'apprentissage, sur 10 exécutions avec les paramètres dans le Tableau 3.1

Les paramètres ont été choisis sur la base des recherches d'hyperparamètres faites dans le chapitre 2 ainsi que des résultats des paragraphes précédents et sont les suivants :

Pas de temps ($T_{refract}$)	Fuite du neurone : γ_{LIF}	Fuite de l'intégrateur : γ_{LI}	Seuil du neurone : u_{th}	τ ($T_{refract}$)
0.5	0.01	0.1	1	50
Taux d'apprentissage : η_r	N_{filt}	β	T_{free} ($T_{refract}$)	T_{nudge} ($T_{refract}$)
6×10^{-5}	10	0.5	75	62.5

Table 3.1 – Hyperparamètres d'*EqSpike* avec des memristors pour le jeu de données *Digits*.

3.6.2 Modification parcimonieuse des poids synaptiques

Une méthode complémentaire pour réduire le nombre d'impulsions de programmation consiste à ne pas actualiser systématiquement le memristor après chaque impulsion de communication. Comme expliqué ci-dessus, dans cette implémentation d'*EqSpike*, une impulsion de programmation est envoyée après chaque impulsion de communication pendant la phase d'apprentissage. Pour réduire le nombre d'impulsions de programmation, il est possible de mettre à jour les synapses après plusieurs impulsions de communication plutôt qu'après chacune d'elles. Cependant, si le memristor est modifié non pas après chaque impulsion, mais toutes les $1/p$ impulsions en moyenne et que le taux d'apprentissage augmente proportionnellement avec un facteur p, les poids synaptiques seront modifiés de la même valeur, en moyenne, que dans *EqSpike*. Cela permet à la fois l'augmentation du taux d'apprentissage, qui augmente de façon inversement proportionnelle à la probabilité des modifications, donc le besoin d'une précision plus faible sur les memristors, et la réduction du nombre d'impulsions de programmation qui sont coûteuses en énergie. Comme un bon compromis entre l'économie d'énergie et la perte de précision est souhaité, l'objectif est de changer d'au moins un ordre de grandeur l'amplitude des modifications des poids synaptiques de façon à avoir un impact significatif sur le taux d'apprentissage. Nous proposons deux méthodes de mise à jour des synapses.

Méthode déterministe - Une première méthode est une méthode déterministe, qui met à jour les synapses en fonction de la fréquence des neurones, en actualisant les synapses toutes les N impulsions des neurones. Dans ce cas, si les neurones connectés ont une fréquence f trop faible : $f * T_{refract} < 1/N$, les synapses ne sont pas mises à jour, ce qui induit une perte de précision dans certains cas. La Figure

3.7 montre la perte de précision en fonction du nombre d'impulsions avant modification. La précision est fortement impactée dès 5 impulsions avant une modification, l'écart-type étant important avec un nombre d'impulsions avant modification.

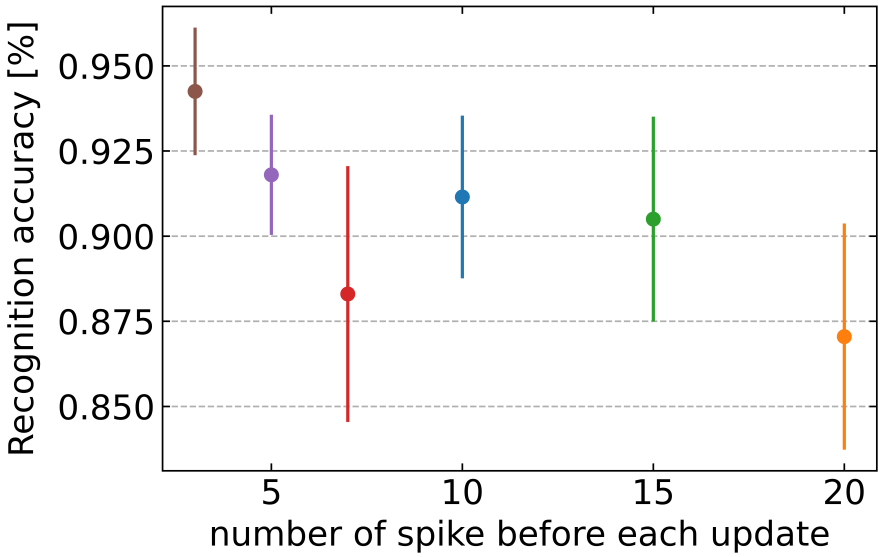


Figure 3.7 – Précision de classification sur la base de test *Digits*, sur 10 exécutions avec les paramètres de la Table 3.1 en fonction du nombre d'impulsions N utilisé pour mettre à jour le memristor pendant la phase d'apprentissage.

Méthode probabiliste - Si la mise à jour des poids est réalisée avec une probabilité p, la perte de précision est moins grande comparée à la méthode déterministe, comme le montre la Figure 3.8. La précision décroît avec une probabilité d'écriture qui décroît. Nous cherchons à choisir la probabilité de mise à jour la plus faible permettant d'obtenir un compromis entre perte de précision et faible probabilité d'écriture. Notre objectif est un gain d'un ordre de grandeur pour le taux d'apprentissage, nous voulons donc une valeur $p < 0.1$. Pour assurer une précision satisfaisante, nous fixons l'objectif de perte de précision à moins de 3 % tout en laissant une marge sur le choix de p. Ainsi la valeur que nous avons choisie satisfaisant ce critère est $p = \frac{1}{15} \approx 0.066$ (en rouge sur la figure).

Avec $p \approx 0.066$, la perte de précision permet d'obtenir un taux d'apprentissage 15 fois plus élevé et d'être ainsi plus robuste aux memristors de précision limitée.

En résumé, la configuration suivante est un bon compromis entre la réduction des impulsions et la perte de précision : $125dt$ pour la durée de la phase d'apprentissage et une probabilité de mise à jour des synapses de $p = 1/15$. Ces paramètres seront donc utilisés dans la suite de ce chapitre, sauf mention contraire. La précision résultante de 93% est maintenant notre référence de base à utiliser avec des composants non idéaux. La réduction du nombre d'impulsions de programmation permet d'augmenter le taux d'apprentissage d'un ordre de grandeur et ainsi d'avoir une modification des poids de l'ordre de 10^{-4} par impulsions. Cela correspond à une incrémentation d'environ 10Ω pour un memristor allant de $10k\Omega$ à $100k\Omega$ contre un incrément $> 1\Omega$ sans l'augmentation du taux d'apprentissage (10 000 états de conductance contre 100 000).

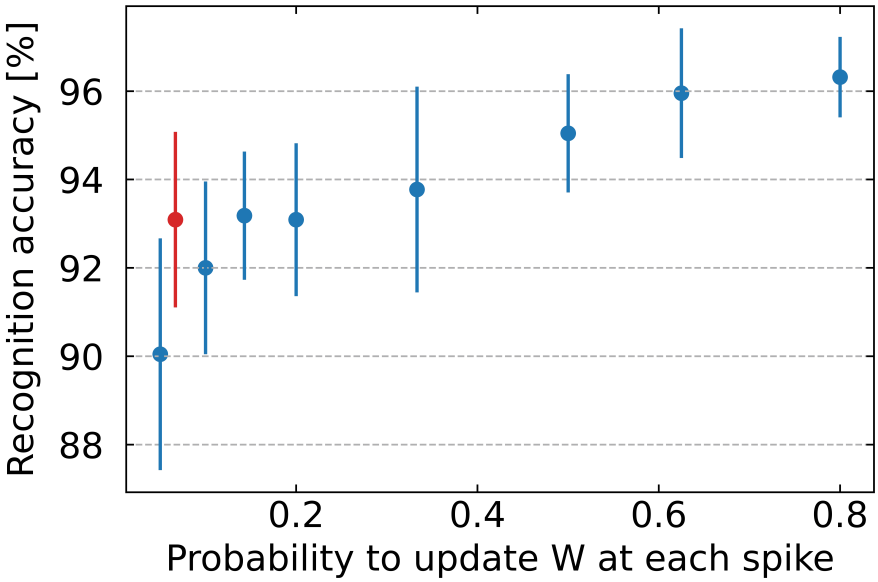


Figure 3.8 – Précision de classification sur la base de test *Digits*, sur 10 exécutions avec les paramètres de la Table 3.1 en fonction d'une probabilité p de mise à jour du memristor pendant la phase d'apprentissage.

3.7 Impact sur la précision de classification des différentes sources de variabilité

Cette section étudie les effets de différents facteurs de variabilité sur la précision de reconnaissance d'image par rapport à la variation entre composants pour voir pour sur quels paramètres il serait intéressant de chercher à réduire la variance et quels paramètres ont peu d'impact sur le taux de reconnaissance avec l'algorithme *EqSpike*. Nous comparerons *EqSpike* avec les deux variantes de *BP* présentées ci-dessus.

Dans un premier temps, les memristors seront considérés comme symétriques, linéaires avec leurs paramètres connus. La sous-section 3.8.1 traite de la variabilité des neurones, la sous-section 3.8.2 considère un bruit intrinsèque lors de la modification des poids synaptiques. Ensuite, nous modifierons les memristors pour ajouter une non-linéarité, puis nous étudierons l'impact de la variabilité des pentes (symétriques ou non). Enfin, nous modifierons les seuils d'écriture des memristors. Nous nous concentrons sur le modèle générique des memristor présenté plus tôt dans ce chapitre (Équation 3.2), les variations entre les différents types de composants sont montrés section 1.4.1.

Comme *BP* est l'algorithme de l'état de l'art pour cette tâche de classification d'images, avec des neurones à fonction d'activation continue et des connexions de type *feed-forward*. Cependant, *BP* n'a pas un apprentissage intrinsèque au niveau des synapses et nécessite donc des circuits externes pour stocker et appliquer les gradients d'erreur sur les poids. Pour pouvoir comparer la résistance aux variations de l'implémentation d'*EqSpike* avec des memristors et des neurones *LIF*, nous allons implémenter l'algorithme *BP* avec différents optimiseurs. Nous nous comparons à *BP*, car il n'y a pas d'implémentation avec la variabilité des composants pris en compte sur le jeu de données *Digits*. De plus, nous voulons voir si *EqSpike* est plus ou moins robuste comparé à *BP* grâce à l'apprentissage intrinsèque. Nous avons voulu une comparaison équivalente en topologie et en nombre de composants, c'est pour cela que nous nous comparons à *BP* et non à *BPTT* qui est plus proche de *EqSpike* mais qui nécessite de considérer un réseau déplié dans le temps pour l'apprentissage.

De plus, les paramètres choisis doivent être aussi similaires que possible pour la simulation des deux algorithmes. La consommation d'énergie et la robustesse du réseau dépendent des composants utilisés, pour permettre une comparaison équitable entre les algorithmes, il faut les comparer à matériel équivalent. Par conséquent, la comparaison est réalisée avec la même topologie du réseau, en particulier le même nombre de neurones et de synapses. De plus, il est nécessaire de simuler les synapses comme si elles étaient implémentées avec des memristors. Pour ceci, nous avons proposé de calculer le gradient d'erreur puis de le régulariser et

pour écrire la synapse, nous lui envoyons une impulsion avec une tension adéquate au changement voulu $V_{th} + \Delta W$ en appliquant les équations du changement de conductance du memristor (Équation 3.2).

Dans une implémentation matérielle, avec des neurones à impulsions, il est compliqué de multiplexer les neurones, car il faut garder en mémoire chaque potentiel de membrane. Pour simuler cette fonctionnalité, la taille du batch est fixée à 1 pour *BP* comme sur *EqSpike*. Nous n'employons pas de *dropout* [Hinton et al., 2012b], de *Batch Normalization* [Ioffe and Szegedy, 2015] ou d'autres procédés algorithmiques habituellement utilisés pour augmenter les performances de *BP*, car leur implémentation serait compliquée dans une puce neuromorphique à faible consommation et n'est pas utilisée dans EqSpike.

Pour *BP*, deux méthodes de descente de gradient stochastique seront utilisées (expliquées en détaille section 1.1) : *SGD* sans *momentum* comme performance de base, noté *BP-SGD*, et l'algorithme *ADAM* [Kingma and Ba, 2017], noté *BP-ADAM*, pour une performance plus élevée, puisque sa régularisation, son taux d'apprentissage adaptatif qui réduit ΔW lorsque le réseau converge et son *momentum* qui évite les changements brusques dans la direction de ΔW, pourraient réduire l'impact de la variabilité par rapport à *BP-SGD*.

En général, dans un réseau *fully connected* avec *BP*, la fonction d'activation utilisée pour la couche cachée est *ReLU*. Pour des raisons physiques, il est impossible de réaliser une fonction non bornée (en analogique). Un choix possible de fonction d'activation bornée entre 0 et 1, et qui est équivalent à la fonction utilisée pour *EqSpike*, est une *hard-sigmoïde* (Équation 2.4). Cependant, avec une *hard-sigmoïde* pour les neurones de la couche de sortie, le réseau entraîné avec *BP* obtient une très mauvaise précision (environ 70%). Pour pouvoir comparer les algorithmes à des précisions équivalentes, les fonctions d'activation de la couche de sortie dans *BP* sont remplacées par des sigmoïdes (proches de la *hard-sigmoïde*). La fonction sigmoïde utilisée pour les neurones de la couche de sortie est par conséquent :

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.11}$$

La fonction de coût L utilisée dans *BP* et *EqSpike* est l'erreur quadratique moyenne, avec $\hat{y}$ la valeur du neurone de sortie et y la cible :

$$L(\hat{y}, y) = \frac{1}{M} \sum_{k=0}^{M} (\hat{y_k} - y_k)^2, \tag{3.12}$$

Les paramètres utilisés pour l'apprentissage avec *BP* sont montrés dans le Tableau 3.2.

Optimiseur	Topologie	Fonction d'activation couche cachée	Fonction d'activation couche de sortie	Taux d'apprentissage
BP-SGD	784-100-10	*hard sigmoid*	sigmoid	0.1
BP-ADAM	784-100-10	*hard sigmoid*	sigmoid	0.001

Table 3.2 – Hyperparamètres les plus favorables de *BP-SGD* et *BP-ADAM* pour le jeu de données *Digits*.

3.8 Robustesse à la variance des composants

3.8.1 Impact de la variabilité des neurones sur la précision de classification

Dans une implémentation matérielle neuromorphique, la fonction d'activation du neurone est sujette à la variabilité des composants du neurone. L'optimisation des poids avec *BP* et *EqSpike* est réalisée grâce à la dérivée de la fonction d'activation. Une modification inconnue de la fonction d'activation ne permet pas de calculer le gradient exact et peut réduire la précision de classification du réseau.

Pour une comparaison équitable, nous appliquons les mêmes modifications pour introduire la variabilité dans les neurones *LIF* et pour les neurones sans impulsion, compatibles avec différentes fonctions d'activation et physiquement cohérentes avec les neurones CMOS. Les variations des fonctions d'activation seront les suivantes :

$$g(x) = f(\alpha x)$$

avec f la fonction d'activation idéale, x l'entrée du neurone et α une valeur constante pour chaque neurone initialisée à la création du réseau et tirée d'une distribution normale α de moyenne nulle et d'écart-type σ_{neuron}, avec α un réel positif. En considérant une fuite faible pour le neurone *LIF*, la modification de la fonction d'activation est équivalente à la modification de son seuil.

La variation des neurones est supposée inconnue pour l'optimiseur, car il serait compliqué sur une puce de mesurer tous les neurones et de modifier le calcul de la dérivée en conséquence. Le calcul de la dérivée pour *BP* ne peut donc pas prendre en compte α. Les dérivées calculées par l'algorithme de *BP* sont donc les dérivées exactes : $g'(x) = f'(x)$.

La Figure 3.9 montre la robustesse des algorithmes *EqSpike*, *BP-SGD* et *BP-ADAM* aux variations de la fonction d'activation. *BP* et *EqSpike* sont tous deux robustes à ce type de variation des neurones, avec des résultats légèrement supérieurs pour *BP-ADAM*, qui perd seulement 15% de précision pour un écart-type α de 100% peu probable en pratique, ce qui signifie qu'une partie des neurones ont un facteur $\alpha = 0$ (neurone inactif). *EqSpike* extraie les gradients directement des neurones du réseau sans hypothèse sur leurs propriétés, et donc prend directement en compte leur variabilité. Cependant, sa règle d'apprentissage est une approximation de BPTT. Nous observons que les deux effets se compensent, résultant pour la tâche simple étudiée en une meilleure précision pour *BP*. On observe en ce sens une perte de 17% pour $\sigma_{neuron} = 1$ malgré le fait que *EqSpike* calcul implicitement le gradient sans connaissance préalable de la fonction d'activation. La robustesse de *BP* comparée à *EqSpike* peut aussi être due à la simplicité de la tâche ou à la simplicité de la fonction d'activation qui devient une approximation d'une fonction binaire avec un α suffisamment grand. En revanche, sur un réseau profond les pertes de précision d'*EqSpike* devraient être moindres que sur *BP* qui propagerait l'erreur d'approximation de la dérivée sur chaque couche. De plus, la règle d'apprentissage d'*EqSpike* est plus avantageuse pour une implémentation matérielle, car elle est locale, intrinsèque et à deux facteurs, pour une précision similaire.

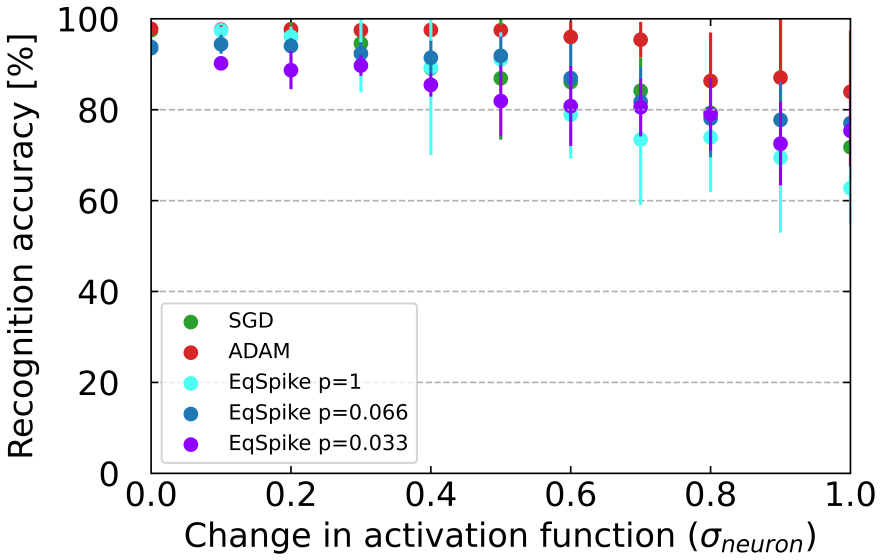


Figure 3.9 – Précision de classification pour le jeu de données de test *Digits* en fonction de la variation des neurones.

[Querlioz et al., 2013] modifient le seuil d'activation des neurones à impulsion pour en étudier l'impact sur un réseau de neurone CMOS et des synapses constituées de memristors entraînés avec *STDP*. Une variation du seuil d'activation des neurones change la fréquence d'impulsion des neurones (un seuil deux fois plus faible augmente la fréquence d'impulsion d'un facteur deux). Ce SNN sans homéostasie a ses performances de classification sur *MNIST* qui chutent à 22% pour une déviation de 25% du seuil d'activation, mais est robuste à la variation des neurones si l'homéostasie est implémentée. Nos pertes de précision sont

moindres, mais la comparaison n'est pas directe, car *Digits* est un jeu de données plus simple que *MNIST*. Elle pose la question d'une implémentation possible de l'homéostasie avec *EqSpike*.

3.8.2 Impact d'un bruit blanc lors de la modification du poids synaptique sur la précision de classification

La mise à jour du memristor est généralement bruitée et conduit à une imprécision lors du processus d'apprentissage. Pour modéliser les variations du memristor, nous utilisons un bruit blanc gaussien γ_{noise} additif de moyenne nulle et de variance σ^2_{noise}, comme dans [Ernoult et al., 2019a] : $\Delta W_{total} = \Delta W_{compute} + \gamma_{noise}$ avec $\gamma_{noise} = \mathcal{N}(0, \sigma^2_{noise})$. Cette section montre l'impact du bruit sur la précision de la reconnaissance. En revanche, le bruit sur les entrées n'est pas étudié, puisque c'est un problème générique dans les réseaux de neurones profonds et non dépendant des memristors.

La Figure 3.10 montre l'évolution de la précision de classification en fonction du bruit pour *BP* et *EqSpike* en échelle logarithmique sur l'abscisse. *BP-SGD* et *BP-ADAM* sont plus robustes au bruit que *EqSpike* qui subit une grosse perte de précision pour un ordre de grandeur de moins sur σ_{noise}. Comme expliqué dans la section précédente, les memristors sont mis à jour pour l'algorithme *BP* après le calcul du gradient et sa régularisation, et le bruit est ajouté à cette étape.

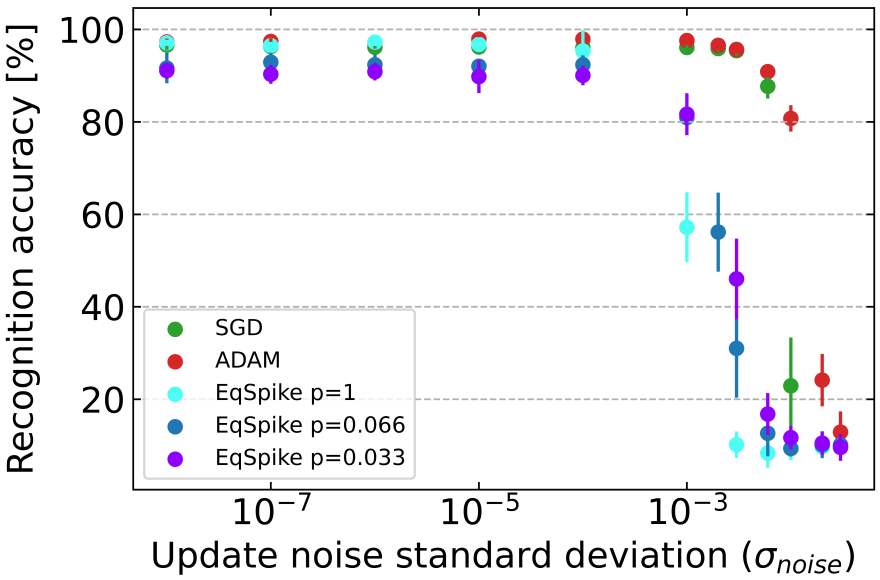


Figure 3.10 – Précision de classification en fonction du bruit lors des mises à jour des memristors sur les données de test *Digits*

Lors de la phase d'apprentissage, *EqSpike* peut modifier la valeur des poids synaptiques à chaque impulsion du neurone, contrairement à *BP* qui modifie le poids une fois par synapse après le calcul des gradients d'erreur. L'impact du bruit est donc plus important sur *EqSpike* que sur *BP*, probablement en raison de l'accumulation du bruit à chaque impulsion de programmation pendant la phase d'apprentissage. En effet, le bruit cumulé par phase d'apprentissage par synapse est

$$\gamma_{noise} = \mathcal{N}(0, n\sigma^2_{noise}) \tag{3.13}$$

avec n le nombre d'impulsions de programmation pour la phase d'apprentissage.

Avec une fréquence maximale de $1T_{refract}$ pour toute la phase d'apprentissage et une durée de $62,5T_{refract}$, il y a un maximum de $62T_{refract}$ impulsions pour les deux neurones connectés. Avec la probabilité de mise à jour de $p = 1/15$, on obtient environ $n \approx 8$ ce qui augmente de presque un ordre de grandeur la valeur du bruit pour *EqSpike* sur la Figure 3.10. En effet, si on regarde la courbe de *EqSpike* avec facteur $1/8$ sur le bruit, on a une perte similaire de précision entre *EqSpike* et *BP* pour un bruit total similaire. Cette hypothèse est confirmée avec le décalage de la perte de précision en fonction de la probabilité d'impulsion p.

La résistance au bruit est corrélée au nombre d'états de conductance que les memristors peuvent prendre. Un bruit inférieur à 10^{-4} est nécessaire pour avoir des performances de reconnaissance à l'état de l'art avec *EqSpike* dans l'implémentation proposée, ce qui est exigeant pour les composants actuels. Le bruit de conductance à l'écriture de ces derniers dépend beaucoup de la technologie utilisée, de la gamme de conductance d'opération, mais semble en général plutôt supérieur à 10^{-2} [Xi et al., 2021]. Avec un p petit, le nombre de modifications est réduit comparé à $p = 1$ ce qui réduit la sensibilité au bruit et ainsi permet de gagner presque un ordre de grandeur sur le nombre d'états nécessaires représentables par le memristor. Des stratégies supplémentaires restent à trouver pour mitiger l'effet du bruit des composants au niveau algorithmique, en exploitant par exemple la binarisation des réseaux [Laydevant et al., 2021].

L'article [Ernoult et al., 2019a] montre un réseau plus robuste au bruit, mais dans leur cas, les mises à jour synaptiques sont du même ordre de grandeur que le bruit, ce qui donne peu de perte de précision dans le cas de la Figure 3.10 quand cette condition est respectée. Cependant, le réseau présenté dans cet article est une machine de Boltzmann avec une topologie différente. Dans [An, 1996], il est montré que le bruit peut améliorer les performances de *BP*, mais aussi qu'il est réduit d'une façon exponentiellement inverse au cours de l'apprentissage. Ce n'est pas le cas dans une implémentation avec des memristors où le bruit reste du même ordre de grandeur.

3.8.3 Impact de la non-linéarité exponentielle des memristors sur la précision de classification

Nous allons maintenant nous concentrer sur l'étude des imperfections de la pente $S^{\pm}$, du seuil $V_{th}^{\pm}$ et du facteur de non-linéarité $\eta^{\pm}$ des memristors. Chaque paramètre de chaque memristor est initialisé à la création du réseau et ne change pas pendant l'exécution.

L'asymétrie de la non-linéarité est un problème important avec les memristors [Fouda et al., 2019b, Chang et al., 2018, Fouda et al., 2019a]. L'impact de l'asymétrie peut être réduit en envoyant un train d'impulsions spécifique pour changer les poids, ce qui est impossible à réaliser avec *EqSpike* sans une refonte de l'algorithme et l'adaptation du circuit sur chaque memristor [Park et al., 2016]. Certains memristors, en particulier les *PCMs*, possèdent une très forte asymétrie. [Burr et al., 2017] montrent que les défauts des *PCMs* induisent des chutes de performance sur le jeu de données *MNIST*, les paramètres les plus critiques étant le taux d'apprentissage, la pente de la non-linéarité et l'asymétrie.

La non-linéarité et la non-symétrie dans les memristors sont des problèmes souvent soulignés dans leur utilisation dans des réseaux de neurones. L'état de l'art des memristors cherche à minimiser la non-linéarité et l'asymétrie [Xi et al., 2021]. La non-linéarité est dissymétrique, mais l'espace d'exploration combinatoire devient prohibitif à simuler. Avec des fortes dissymétries, les résultats obtenus par quelques scénarios choisis empiriquement sont en deçà des performances visées. Cependant, en partant de l'hypothèse que ce problème sera amélioré dans les générations futures des memristors, nous avons concentré cette étude sur le cas générique $\eta^{+} = \eta^{-}$.

La Figure 3.11 montre que la précision de classification diminue lentement pour des ordres de grandeur de non-linéarité trouvables sur les composants expérimentaux du *survey* [Xi et al., 2021], sauf pour l'apprentissage avec *BP-SGD*. Les performances plus faibles obtenues avec *BP-SGD* peuvent être dues aux changements brusques de la direction du gradient entre deux itérations. En effet, l'exposant de non-linéarité est plus important quand la conductance G est faible et doit être augmentée ou, quand la conductance est élevée et doit être diminuée. Avec un gradient qui a tendance à être dans la même direction entre chaque modification, l'impact de la non-linéarité est moindre que si le gradient d'erreur change de signe régulièrement, ce qui augmente fortement l'exposant de la non-linéarité. L'optimiseur *BP-ADAM*, quant à lui, introduit un momentum pour garder une inertie pour la direction du gradient d'erreur et est donc moins sensible à cet effet. Dans *EqSpike*, la dynamique change lentement au cours d'une phase d'apprentissage, donc le gradient d'erreur garde globalement son signe pendant une même phase. La perte de précision supérieure pour $p = 0.033$ comparé à $p = 1$ et $p = 0.066$ peu être due à une approximation trop grande des gradients ainsi qu'à un taux d'apprentissage supérieur (qui est inversement proportionnel à p) [Burr et al., 2017] montrent une forte baisse de précision due à la non-linéarité sur un réseau composé de *PCM*, qui sont aussi fortement asymétriques. Leurs résultats montrent que l'asymétrie a un impact plus important que la non-linéarité des *PCMs*.

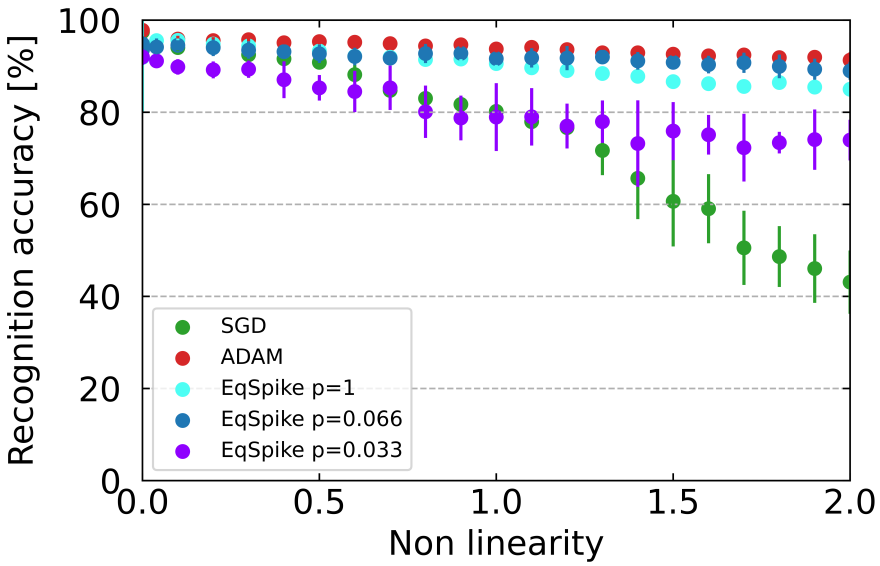


Figure 3.11 – Précision de classification en fonction de la non-linéarité des memristors sur les données de test *Digits*.

3.8.4 Impact de la variabilité de la pente dans le modèle des memristors sur la précision de classification

La variabilité de la pente correspond à la sensibilité à l'incrémentation ou à la décrémentation de la conductance du memristor par un signal supérieur aux seuils. La Figure 3.12 montre la robustesse à la variation de la pente. À l'initialisation du réseau, les pentes sont choisies aléatoirement avec une distribution normale pour chaque synapse, avec $S^+ = \mathcal{N}(0, \sigma^2_{slope})$ et $S^- = \mathcal{N}(0, \sigma^2_{slope})$ choisies indépendamment, les pentes des memristors sont donc initialisées de manière asymétrique. La précision diminue plus rapidement pour *EqSpike* que pour *BP*. La pente est un facteur multiplicatif, ce qui signifie qu'avec un gradient d'erreur élevé et une pente supérieure à 1, l'erreur risque d'être amplifiée. *EqSpike* montre en moyenne, comme précédemment, une modification du memristor par impulsion plus élevée que les deux autres méthodes, ce qui peut expliquer la différence de précision. Dans [Xi et al., 2021], les déviations en pratique de la pente sont d'environ $3 - 5\%$, ce qui garde la précision de classification du réseau dans les marges acceptables si l'on se réfère à la Figure 3.12.

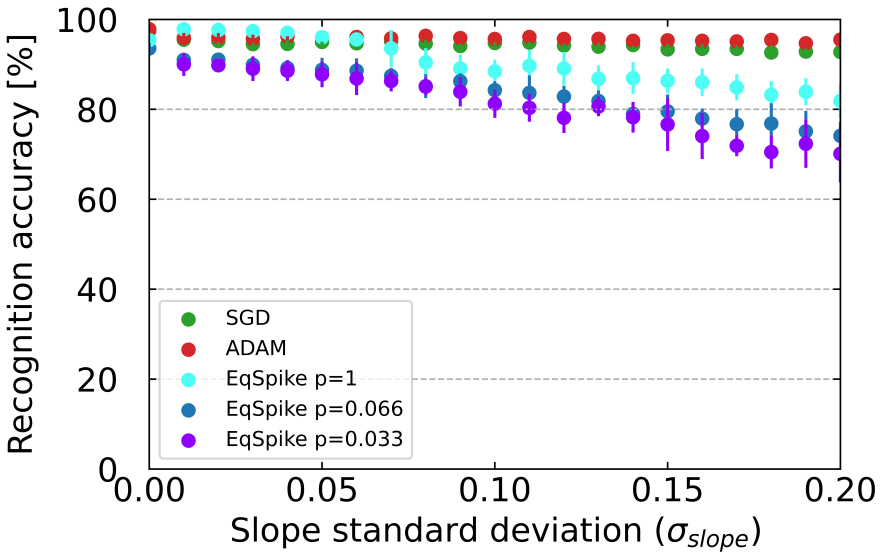


Figure 3.12 – Impact de la variation aléatoire dissymétrique de la pente $S^{\pm}$ du memristor.

Le cas de pente asymétrique est le plus défavorable. Lorsque l'on fait varier les pentes pour des memristors symétriques, *EqSpike* et *BP* sont robustes jusqu'à des écarts-types plus élevés que dans le cas non symétrique,

comme montré sur la Figure 3.13 où la variance maximale est 5 fois supérieure à la Figure 3.12. En effet, une pente symétrique peut être interprétée comme un taux d'apprentissage local à la synapse. Avec une pente symétrique, le gradient est multiplié par le même facteur $S^{\pm}$ lors d'une opération d'incrémentation ou de décrémentation, ce qui permet d'éviter une perte de précision importante.

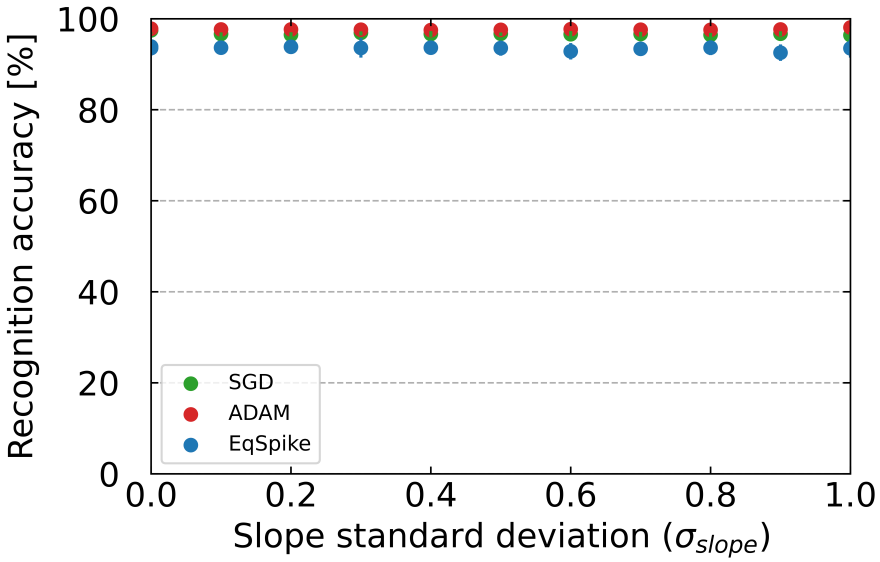


Figure 3.13 – Impact de la variation aléatoire symétrique de la pente $S^{\pm}$ du memristor.

3.8.5 Impact de la variabilité du seuil des memristors sur la précision de classification

Dans l'Équation 3.2, le seuil est additif et non multiplicatif au gradient. Dans le cas idéal, le seuil annule exactement le signal de l'impulsion de programmation et est simplifié dans l'Équation 3.2. l'Équation 3.2 peut être réécrite en incluant une variation du seuil :

$$V = \dot{\rho}\frac{\tau}{\gamma_{LI}}\gamma_{lr} + (V^{\pm} + eV_{th}^{\pm}), \tag{3.14}$$

où $e = \mathcal{N}(0, \sigma^2_{threshold})$ est la variation du seuil. En remplaçant V dans l'Équation 3.2 pour des memristors linéaires nous obtenons :

$$\Delta W = \dot{\rho}\frac{\tau}{\gamma_{LI}}\phi_{lr} + \frac{S^{\pm}t_{pulse}eV_{th}^{\pm}}{G_{max} - G_{min}} \tag{3.15}$$

Pour s'adapter aux différents ordres de grandeur des paramètres du memristor, ϕ_{lr} inclut une normalisation des paramètres du memristor, grâce à la compensation du seuil par l'impulsion de programmation, comme montré dans l'Équation 3.8. Avec la variation du seuil, il est nécessaire de connaître les paramètres du memristor pour calculer la partie droite de l'Équation : $\frac{S^{\pm}t_{pulse}V_{th}^{\pm}}{G_{max}-G_{min}}$. La pente, le seuil et la conductance sont inhérents au memristor mais l'impulsion de programmation est externe, c'est-à-dire que sa durée peut être modifiée. Avec une impulsion de programmation plus courte en temps, mais avec un ϕ_{lr} équivalent, la partie droite de l'Équation 3.15 est moins importante que la partie gauche $\bar{\dot{\rho}}\frac{\tau}{\gamma_{LI}}\phi_{lr}$.

La Figure 3.14 montre la précision en fonction de la variation du seuil pour différents modèles de memristors. Nous notons $speed$ ce que nous qualifions de vitesse de changement du memristor. $Speed$ est la variation de conductance, normalisée par sa gamme possible de conductance, pour une impulsion de durée $t_{pulse} = 50ns$ et d'amplitude $1V$:

$$speed = \frac{S^{\pm} \times 5.10^{-8}}{G_{max} - G_{min}} \tag{3.16}$$

Il est important de noter que, si le rapport entre l'erreur $\frac{s^{\pm}t_{pulse}V_{th}^{\pm}}{G_{max}-G_{min}}$ et le gradient d'erreur $\bar{\dot{\rho}}\frac{\tau}{\gamma_{LI}}\phi_{lr}$ diminue, sans changement de seuil et de longueur de l'impulsion de programmation, alors cela est équivalent

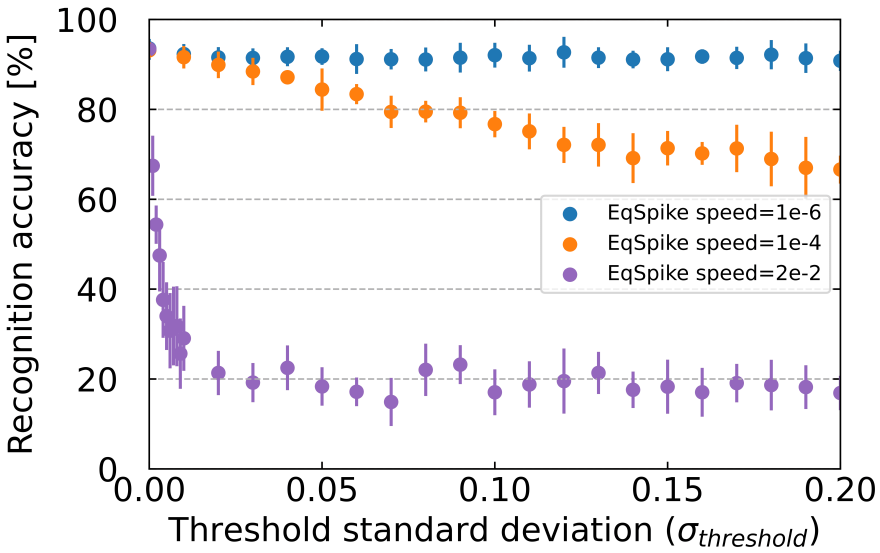


Figure 3.14 – Évolution de la précision avec des variations aléatoires du seuil pour différents modèles de memristor.

à augmenter le signal $\bar{\rho}$. Ainsi, l'utilisation d'un memristor plus lent augmente la consommation d'énergie, mais permet une plus grande résilience à la variation du seuil en diminuant le ratio signal sur bruit. La Figure 3.14 montre trois vitesses de memristor, avec des paramètres testés expérimentalement par [Wu et al., 2018] pour le plus rapide, [Woo et al., 2017] pour le plus lent et des paramètres artificiels entre les deux autres memristors pour l'intermédiaire. La Figure 3.14 montre que le memristor lent est moins sensible à la variation du seuil, mais la modification du poids est aussi plus gourmande en énergie. La moyenne mesurée des gradients d'erreur (partie gauche de l'Équation 3.15) de *EqSpike* est de l'ordre de 10^{-4} ce qui correspond à la *vitesse* du memristor intermédiaire sur la Figure 3.14.

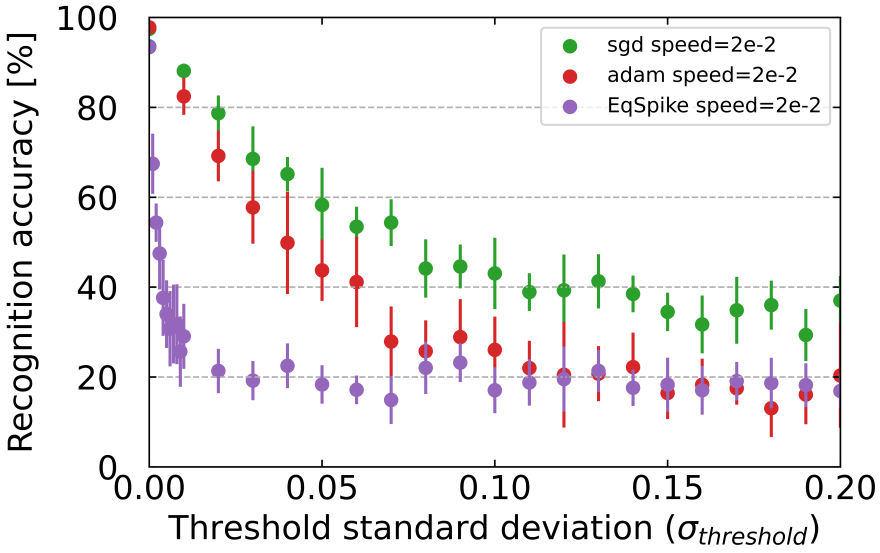


Figure 3.15 – Évolution de la précision avec des variations aléatoires du seuil pour différents algorithmes.

La Figure 3.15 montre que les algorithmes *BP-SGD* et *BP-ADAM* présentent également une perte de précision dans les simulations utilisant les memristors les plus rapides. Mais leur robustesse reste meilleure, la perte de précision arrivant avec un $\sigma_{threshold}$ de plusieurs ordres de grandeur supérieur à *EqSpike*. Cette robustesse peut s'expliquer par le fait que les memristors seront toujours modifiés dans la direction du gradient, car l'impulsion est toujours du signe du gradient pour *BP*, contrairement à *EqSpike* où les impulsions de programmation sont positives puis négatives, ce qui peut changer le signe de la modification du memristor. Ainsi *BP* permet d'obtenir de meilleurs résultats surtout avec les memristors les plus lents et ceux de vitesse intermédiaire.

Après le bruit d'écriture de la conductance, le seuil du memristor est le deuxième paramètre qui affecte le plus la précision cette étude. Ce résultat n'est pas surprenant, car notre implémentation d'*EqSpike* est basée sur la superposition des deux impulsions des neurones connectés pour obtenir une impulsion avec l'estimation de la dérivée de la fréquence en dessous du seuil de modification de la synapse. Il est donc normal que si la tension requise pour écrire le memristor ne s'annule pas avec la tension de programmation, des erreurs de programmation faisant chuter la précision se produisent, en particulier sur les memristors les plus sensibles à ces erreurs (les plus rapides). Toutefois, choisir des memristors lents permet de maintenir des taux de reconnaissance élevés.

3.9 Conclusion

Dans ce chapitre, nous avons proposé un circuit pour implémenter la règle d'apprentissage *EqSpike* avec des synapses memristives et des neurones *LIF*. Avec un nombre de neurones inférieur à celui des synapses, la complexité rajoutée sur les neurones permet un gain de place plus important que si elle était présente au niveau des synapses. Le gradient est calculé localement et implicitement par la dynamique des neurones. Les synapses sont modifiées intrinsèquement par les impulsions du neurone.

Avec des memristors parfaits, il n'y a pas de perte de précision de classification par rapport à une implémentation *EqSpike*. Nous avons proposé des hyperparamètres qui permettent de réduire la consommation d'énergie et d'augmenter le pas de mise à jour des synapses pour mieux s'adapter à une faible précision des memristors, pour une faible perte de précision de classification. La réduction du nombre d'impulsions de programmation, grâce à leur génération stochastique, a permis d'augmenter le taux d'apprentissage d'un ordre de grandeur et ainsi d'atteindre une tolérance au bruit de conductance lors de la programmation de l'ordre de 10^{-4}. Cela correspond à un incrément d'environ 10Ω pour un memristor couvrant une plage de $10k\Omega$ à $100k\Omega$ contre un incrément $> 1\Omega$ sans l'augmentation du taux d'apprentissage et la diminution de nombre de modifications des memristors. Des stratégies supplémentaires devront être mises en œuvre pour atteindre des tolérances de 10^{-2} correspondant aux bruits expérimentaux, par exemple via la binarisation du réseau.

Enfin, l'impact de différentes non-idéalités de composant à composant a été étudié. La résilience aux variations intrinsèques des memristors a été comparée entre *EqSpike* et *BP*. *EqSpike* est moins robuste que *BP* aux variations à cause du nombre d'écritures plus important pour *EqSpike*. À l'exception de la variabilité du seuil du memristor dans le cas des memristor rapides, et de variations asymétriques de la pente autour de sa valeur moyenne, notre implémentation est résiliente à la plupart des imperfections du memristor, pour des déviations proches des memristors les plus avancés en réduction de la variation inter-composants $(5\% - 10\%)$.

Pour la variance des neurones, on observe que la perte de précision est non significative pour $\sigma_{neuron} = 0.1$. Avec une variance 10% sur les pentes, le réseau perd environ 10% de précision et avec une variance 20% sur les pentes, le réseau perd environ 18% de précision, ce qui en fait le deuxième paramètre le moins robuste à la variabilité après le bruit. L'implémentation d'*EqSpike* avec des memristors est basée sur la superposition d'un signal de programmation égal au seuil d'écriture du memristor et d'un signal proportionnel à la dérivée de la fréquence du neurone. Cette implémentation rend le réseau sensible aux variations intercomposants du seuil d'écriture des memristors. Avec des memristors *rapides* (la conductance qui change rapidement pour une faible tension à ses bornes), le réseau est incapable d'apprendre pour des variations avec un écart-type de plus de 0.01%. En revanche, si le memristor est suffisamment *lent*, alors l'algorithme peut être robuste à cette variation avec presque aucune perte de précision. Le réseau est également robuste à la non-linéarité avec une perte de moins de 6% pour une très forte non-linéarité et moins de 2% pour un facteur de non-linéarité inférieur à 1. Les memristors avec le moins de non-linéarité possèdent un facteur de non-linéarité inférieur à 1 [Gao et al., 2015]. La perte de précision due à des asymétries n'a pas été quantifiée, mais les essais effectué monte une incapacité d'obtenir une bonne précision, ce qui est en accord avec la littérature.

Contrairement à *BP-ADAM* et *BP-SGD*, *EqSpike* présente un apprentissage local et intrinsèque. Mais *EqSpike* est moins résilient que l'optimiseur *BP-ADAM* sur trois des axes de variation testés : le seuil, le bruit de mise à jour et la pente. Nous n'avons pas trouvé dans la littérature un memristor qui correspondent à tous ces critères en même temps. Une des raisons est que retrouver l'ensemble des valeurs des paramètres souhaités dans un article est souvent difficile. Développer une méthodologie pour obtenir les paramètres importants

pour *EqSpike* est donc nécessaire pour trouver les memristors adéquats. De plus, avec l'amélioration des composants dans le futur, la variabilité va continuer à s'améliorer pour atteindre une perte de précision avec *EqSpike* encore plus faible. Notre étude sur la variabilité permet de choisir les memristors les mieux adaptés pour une implémentation physique d'*EqSpike*, en particulier des memristors *lents* avec une variabilité faible au niveau de la pente.

Plusieurs papiers dans la littérature essaient de mitiger le problème de variation intercomposants, voire de l'exploiter. Par exemple, en rendant les poids binaires [Truong et al., 2014, Qin et al., 2020] ou ternaires [Boquet et al., 2021], la variation sur les états ON et OFF et la précision de la conductance deviennent moins impactants et la précision d'un changement de conductance aussi. Mais une évolution de *EqSpike* avec ces algorithmes ou vers des mécanismes d'auto-adaptation permettrait une meilleure robustesse au défaut des composants. Un premier pas en ce sens a été constitué par la binarisation d'*EqProp* [Laydevant et al., 2021], le défi pour *EqSpike* étant de réaliser une binarisation compatible avec les contraintes des réseaux à impulsions.

Chapitre 4

A-EqSpike : Utilisation de la superposition des impulsions pour calculer la règle d'apprentissage de *EqSpike*

Dans les chapitres précédents, l'algorithme *EqSpike* a été présenté, puis une implémentation de l'algorithme avec des memristors comme synapses, a été proposée et simulée. Pour programmer le memristor, des impulsions de programmation ont été utilisées. En effet, elles permettent de passer le seuil interne du memristor pour modifier sa conductance. Nous avons montré que les modifications de *EqSpike* peuvent être similaires à celle obtenues par la *STDP*. De plus, [Bengio et al., 2017] montrent que la *STDP* pourrait implémenter *Eq-prop*. Enfin, la section 1.4.3 montre comment utiliser la forme des impulsions pour programmer les memristors avec la *STDP*. Dans ce chapitre, nous nous appuyons sur ces fortes similarités entre *EqSpike*, *Eq-prop* et la *STDP* pour modifier les memristors sans le bloc $\overline{\dot{\rho}}$, présenté précédemment, mais grâce aux formes des impulsions des neurones. Pour cela, nous proposons une variante où chaque neurone envoie une impulsion avec une forme spécifique. La superposition des impulsions permet de calculer la dérivée de la fréquence et de modifier le memristor, implémentant ainsi la loi d'apprentissage de *EqSpike* directement.

4.1 Introduction

Nous avons proposé un algorithme avec un apprentissage intrinsèque, *EqSpike*, capable d'effectuer la règle d'apprentissage localement grâce à un bloc $\overline{\dot{\rho}}$ attaché aux neurones. Nous avons proposé un schéma permettant de mettre en œuvre ce bloc $\overline{\dot{\rho}}$ sur un circuit neuromorphique à base de memristors pour les synapses. Un inconvénient de cette méthode est que le bloc $\overline{\dot{\rho}}$ est constitué de différents éléments qui prennent de la place et consomment de l'énergie.

L'algorithme de la *STDP* utilise les impulsions pour calculer la règle d'apprentissage. Il a été montré qu'il est possible, avec la bonne forme d'impulsion des neurones et grâce à leur superposition, d'avoir le même comportement que la *STDP* sur des memristors comme expliqué dans la section 1.4.3. Dans la *STDP*, les synapses sont en général unidirectionnelles. En effet, dans [Linares-Barranco and Serrano-Gotarredona, 2009, Querlioz et al., 2011], les deux impulsions des neurones implémentent la *STDP* entre le neurone postsynaptique et présynaptique, mais cette implémentation fonctionnerait difficilement pour des synapses bidirectionnelles et il faudrait des impulsions bidirectionnelles. Malheureusement, dans *EqSpike* la notion de pré et postsynaptique est ambiguë à cause de la bidirectionnalité du réseau, ce qui empêche d'utiliser les mêmes impulsions que les papiers précédemment cités. Pour implémenter *EqSpike* avec des impulsions qui calculent la règle d'apprentissage localement et modifient le memristor en conséquence, il faut réussir à trouver une forme d'impulsion unique qui permet la bidirectionnalité des synapses.

Dans ce chapitre, nous allons regarder comment il est possible d'implémenter la règle d'apprentissage d'*EqSpike* grâce à la superposition et l'accumulation d'impulsions. Avec des impulsions de forme adéquate et de durée plus élevée que le temps de réfraction des neurones, les impulsions vont s'accumuler, ce qui permet de calculer la dérivée de la fréquence du neurone et ainsi d'implémenter la règle d'apprentissage. Ensuite, nous regarderons le comportement des différentes variantes de l'algorithme avec des complexités d'implémentation physique de différents niveaux sur un problème de classification de chiffres manuscrits, pour comparer leur performance de reconnaissance d'images ainsi que le comportement de l'algorithme en fonction de différents paramètres des formes des impulsions.

4.2 Accumulation d'impulsions avec une forme adéquate pour modifier les memristors

Pour appliquer la loi d'apprentissage d'*EqSpike*, il faut être capable de calculer la dérivée de la fréquence d'un neurone et d'appliquer une modification proportionnelle à cette dérivée sur la conductance du memristor à chaque impulsion de l'autre neurone connecté. Nous avons proposé dans le chapitre 2 une implémentation permettant de calculer la dérivée de la fréquence du neurone grâce au bloc $\overline{\dot{\rho}}$. La mise en œuvre de ce bloc augmente l'espace nécessaire sur une puce implémentant *EqSpike*.

Dans ce chapitre, nous proposons une méthode alternative pour calculer cette dérivée grâce à la superposition et l'accumulation des impulsions du même neurone. Avec une impulsion dont la durée est plus longue que le temps réfractaire du neurone, les différentes impulsions émises par le neurone se superposeront. Alors avec une forme d'impulsion adéquate, la superposition et l'accumulation des impulsions peut avoir le comportement d'un accumulateur. Avec une forme carrée d'amplitude $\gamma_{amp} = 1$ et de durée γ_{len}, la valeur cumulée des impulsions à un temps t correspond au nombre d'impulsions émises dans une fenêtre temporelle de durée γ_{len} précédant le temps t.

Pour obtenir la dérivée de la fréquence, il faut calculer la différence du nombre d'impulsions entre deux instants. Pour effectuer le calcul de la dérivée, nous proposons une forme d'impulsion où la moitié de l'impulsion ($\frac{\gamma_{len}}{2}$) est égale à 1 et l'autre moitié à -1, comme présenté dans la Figure 4.1a. Alors la superposition d'impulsions implémente deux accumulateurs. La partie positive compte le nombre d'impulsions sur une fenêtre $[t - \frac{\gamma_{len}}{2}, t - \gamma_{len}]$ et la partie négative compte le nombre d'impulsions sur une fenêtre $[t, t - \frac{\gamma_{len}}{2}]$. La valeur totale accumulée à un temps t est alors :

$$V_{impulsion}(t) = \sum_{i=t-\frac{\gamma_{len}}{2}}^{t} \delta(i) - \sum_{i=t-\gamma_{len}}^{t-\frac{\gamma_{len}}{2}} \delta(i), \tag{4.1}$$

avec γ_{len} la durée d'une impulsion et $\delta(i) = 1$ respectivement 0 selon la présence ou l'absence d'une impulsion au temps i. Ainsi, lors d'une accélération, l'accumulation des impulsions donnera un signal positif, et inversement lors d'une décélération.

Dans la Figure 4.1b, les événements en rouge sont les moments d'émission des impulsions et la courbe bleue correspond à l'accumulation des impulsions. Le train d'impulsions est équivalent à une forte accélération de la fréquence suivie d'une baisse de la fréquence. Lors de l'accélération de la fréquence (début du train d'impulsion), la valeur des impulsions cumulées augmente avec l'arrivée des impulsions. Quand la fréquence commence à se stabiliser, la valeur de l'accumulation décroît vers 0. L'arrêt des impulsions correspond ensuite à une diminution de fréquence, ce qui donne une valeur négative à l'accumulation des impulsions avant de retourner à 0. Le décalage temporel du signal comparé au train d'impulsions est dû à la durée de l'impulsion. La Figure 4.1b correspond à une petite série d'impulsions, en revanche pour regarder le comportement plus global de l'accumulation d'impulsions, nous avons fait évoluer la fréquence d'un neurone avec une accélération et une décélération réalistes, représentées par la Figure 4.1c. La Figure 4.1d représente l'accumulation des poids au cours du temps. En jaune la valeur oscille autour de 0, car la fréquence est constante, en rouge la moyenne est supérieure à 0 à cause de l'accélération de la fréquence, et en vert la moyenne est inférieure à 0 à cause de la décélération de la fréquence, ce qui correspond au comportement souhaité.

L'accumulation des impulsions d'un neurone permet donc de calculer une estimation de la dérivée de sa fréquence. Cependant, avec une forme carrée, le début et la fin d'une nouvelle impulsion présentent une discontinuité, ce qui conduit à un changement abrupt dans l'estimation de la dérivée. Pour palier à ce problème, une solution possible est d'utiliser une impulsion linéaire par morceaux avec une amplitude maximale, γ_{amp}, égale à $0.1V_{th}$ sur la Figure 4.2a. Les pentes encodent alors la distance temporelle au début de l'impulsion en fonction de son amplitude.

Pour modifier le poids d'une synapse connectant deux neurones i et j, en accord avec la loi d'apprentissage de *EqSpike* :

$$\Delta w_{ij} = (\rho_j \dot{\rho}_i) + (\rho_i \dot{\rho}_j),$$

il faut écrire le memristor de la valeur de l'accumulation des impulsions pour le neurone i à chaque fois que le neurone j émet une impulsion, et réciproquement.

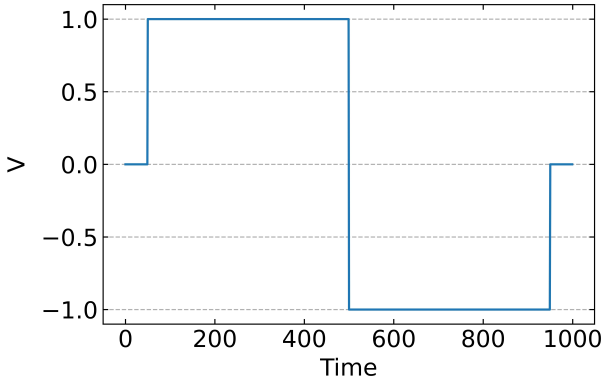
Nous proposons pour cela que le neurone j émette une impulsion de programmation (Figure 4.2b) qui aura une tension suffisamment élevée pour permettre d'écrire le memristor. En effet, pour modifier un memristor, il faut franchir son seuil V_{th}, comme représenté dans l'équation 3.2. Si le neurone j envoie une impulsion avec une partie positive et une partie négative avec une amplitude V_{th} (Figure 4.2b avec $V_{th} = 1$), et que l'autre neurone connecté envoie le signal cumulé de ses impulsions, alors la superposition de ces deux impulsions aux bornes du memristor met à jour sa conductance par une valeur proportionnelle à l'approximation de la dérivée de la fréquence. En effet, le changement de conductance G pour un signal positif, conjointement avec une impulsion de programmation est :

$$\Delta G(t) \propto (V_{mem}(t) - V_{th}) \tag{4.2}$$

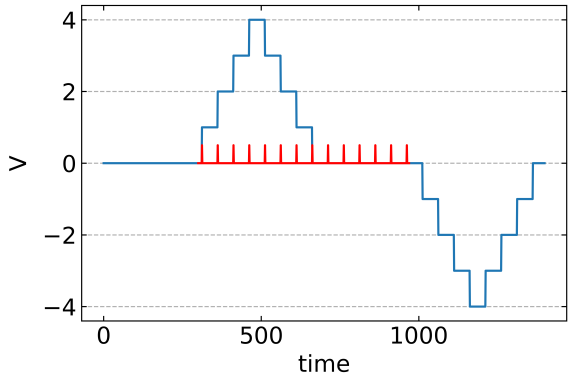
avec $V_{mem}(t) = V_{prog}(t) - \sum_{i=0}^{t} V^i_{queue}(t)$, avec V_{prog} la valeur de l'impulsion de programmation (égale à V_{th}) et $\sum_{i=0}^{t} V^i_{queue}(t)$ la somme des queues d'impulsion $V^i_{queue}(t)$ émises au temps i pour le temps t. Donc, quand un neurone émet une impulsion de programmation, la valeur des queues des impulsions cumulées de l'autre neurone est soustraite aux bornes du memristor, et réciproquement. Pour éviter de modifier le memristor en dehors de la règle d'apprentissage, la durée de l'impulsion de programmation doit être inférieure ou égale à $T_{refract}$. Sans cela les impulsions de programmation d'un même neurone pourraient s'annuler, ou se superposer et dépasser le seuil V_{th}.

La règle d'apprentissage d'*EqSpike* est bidirectionnelle. Pour l'appliquer, il faut à la fois estimer la dérivée de la fréquence de deux neurones connectés et envoyer l'impulsion de programmation. En fusionnant les deux formes 4.2b et 4.2a, on obtient la forme présentée Figure 4.3. Le neurone envoie alors une impulsion avec une partie courte et une amplitude élevée, et avant et après une impulsion avec à la fois un temps long et une faible amplitude. Dans la suite, nous appellerons :

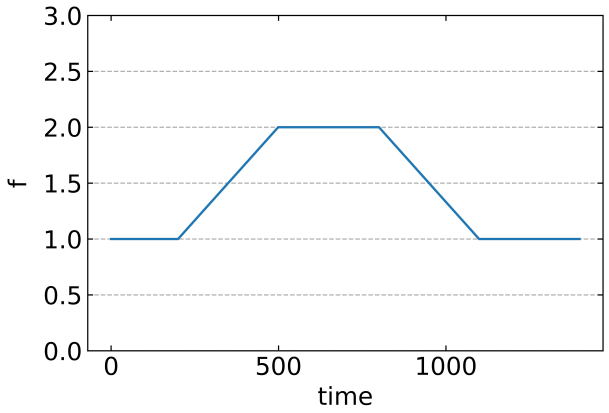
— **impulsion de programmation**, la courte partie avec l'amplitude élevée au centre,
— **queue d'impulsion**, la partie avec la faible intensité qui sert à estimer la dérivée.



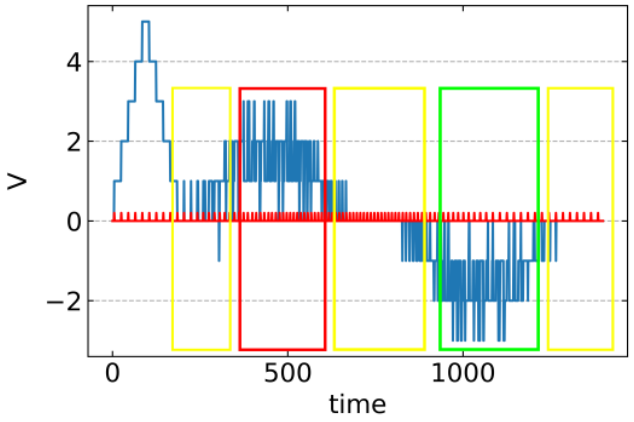
(a) Illustration d'une impulsion carrée de durée $\gamma_{len} = 900$.



(b) Illustration d'un train d'impulsion d'un neurone avec une forme carré d'impulsion. Les événements en rouge sont les moments d'émission des impulsions et la courbe bleue correspond à l'accumulation des impulsions.

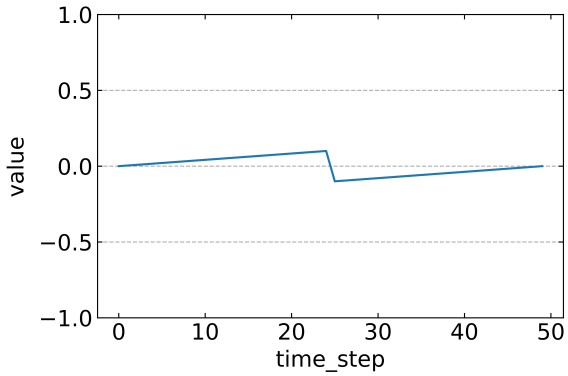


(c) Évolution de la fréquence d'un neurone pour illustrer l'accumulation des impulsions de la figure (d).

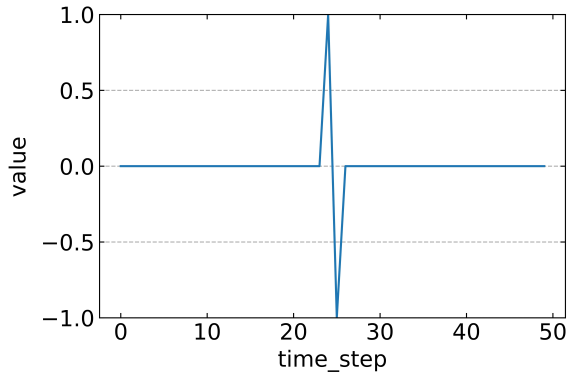


(d) Illustration d'un train d'impulsions d'un neurone. La fréquence du neurone est celle de la figure (c). En jaune la valeur oscille autour de o, car la fréquence est constante, en rouge la moyenne est supérieure à o à cause de l'accélération de la fréquence, et en vert la moyenne est inférieure à o à cause de la décélération de la fréquence.

Figure 4.1 – Illustration de l'accumulation d'impulsions comme accumulateur pour calculer la dérivée de la fréquence.



(a) Forme d'impulsion en scie, avec une amplitude faible (0.1) pour calculer la dérivée de la fréquence.



(b) Forme d'impulsion courte avec une amplitude élevée pour activer le seuil du memristor, ici égale à 1.

Figure 4.2 – Différentes formes pour les impulsions de programmation.

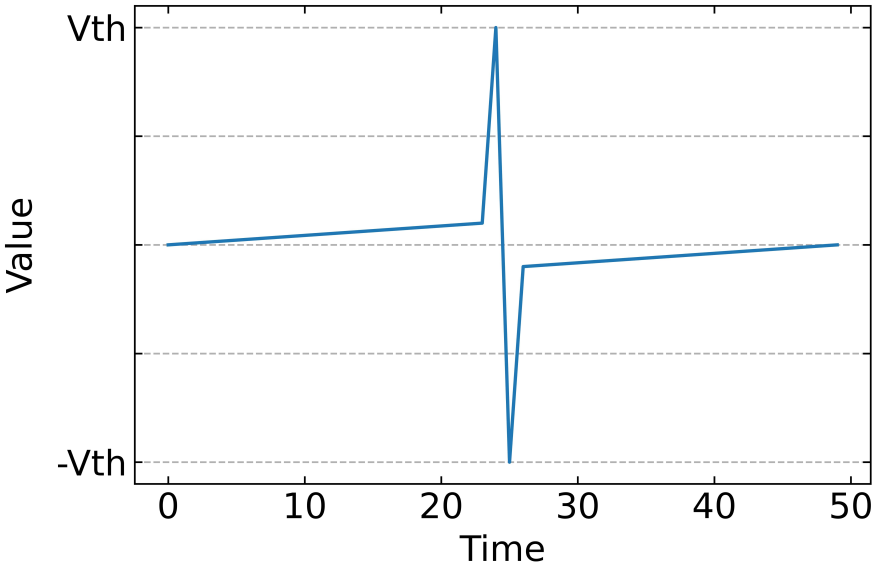


Figure 4.3 – Forme d'impulsion pour un réseau bidirectionnel

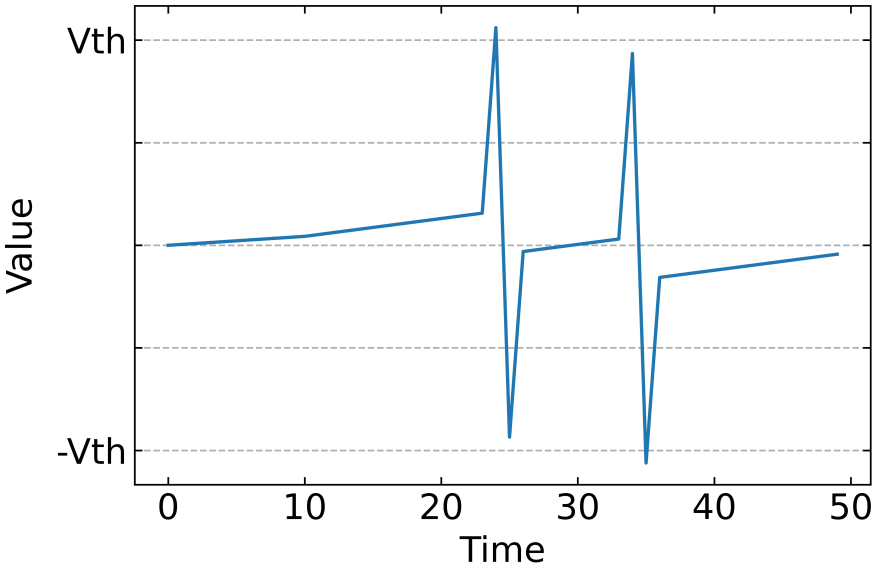


Figure 4.4 – Le signal superposé des deux impulsions ; on observe le dépassement de V_{th} avec l'accumulation des impulsions.

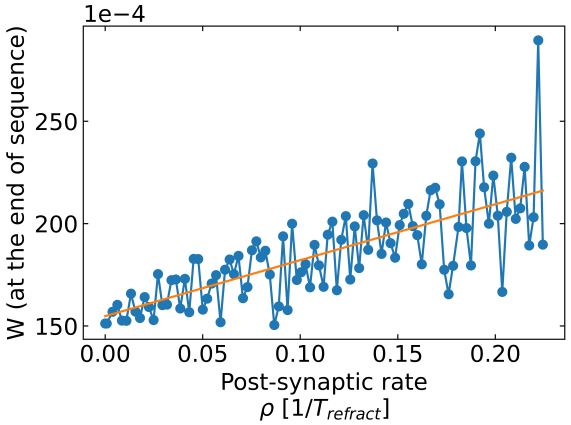
Grâce à cette nouvelle forme, un neurone peut envoyer un signal qui dépasse le seuil du memristor pour permettre sa modification et calculer la dérivée de sa fréquence. Nous appelons cet algorithme *A-EqSpike* pour *EqSpike* avec accumulation d'impulsions.

Pour vérifier que la superposition des impulsions est capable d'approximer la loi d'apprentissage d'*A-EqSpike*, nous considérons, comme dans les chapitres précédents, le cas de deux neurones connectés par une synapse et le même protocole de simulation qu'au chapitre 2. Pour ce faire, les fréquences et respectivement les accélérations de chaque neurone sont fixées alternativement. Le poids synaptique est mis à jour grâce à la superposition des impulsions et de l'équation du memristor idéal.

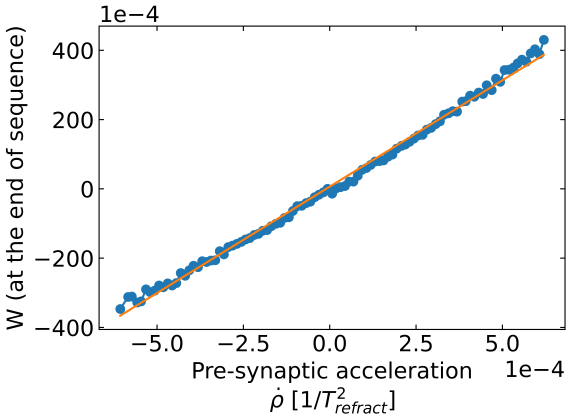
Les Figures 4.5a et 4.5b montrent l'évolution du poids, ΔW, à la fin de la simulation en fonction de la fréquence et de la dérivée de la fréquence pour une forme de durée $100dt$ (pas de temps). La valeur de modification du poids W est plus faible que dans les chapitres précédents, car la modification du poids dépend de l'amplitude des impulsions, de leur durée et du taux d'apprentissage, qui ne sont pas les mêmes. Ces figures ne montrent pas une dépendance linéaire à la fréquence. En effet pour une fréquence de 0, le memristor est quand même modifié. Le problème avec cette méthode est qu'un neurone seul peut écrire le memristor d'une valeur proportionnelle à la dérivée de sa fréquence. La Figure 4.4 illustre ce problème avec deux impulsions d'un neurone. Avec l'accumulation des deux impulsions, V_{th} est dépassé positivement au temps 25 et négativement au temps 36, ce qui modifie la valeur du memristor. Or, pour respecter la règle de *EqSpike*, un neurone ne doit modifier la valeur du poids synaptique qu'en fonction de la dérivée de la fréquence d'un autre neurone. Comme un neurone peut écrire sa propre dérivée sur le memristor à chaque impulsion $(\rho\dot{\rho})$, la règle d'apprentissage est en réalité :

$$\Delta w_{ij} \approx (\rho_i\dot{\rho}_i) + (\rho_j\dot{\rho}_j) + (\rho_j\dot{\rho}_i) + (\rho_i\dot{\rho}_j) \tag{4.3}$$

L'application de cette loi d'apprentissage se remarque avec la Figure 4.5b qui montre la dépendance à la fréquence. En effet, même pour une fréquence du neurone postsynaptique nulle, la modification du poids n'est pas égale à 0, justement, car le neurone présynaptique a modifié seul la valeur de la synapse.



(a) Dépendance de la modification du poids synaptique à la fréquence avec une durée d'impulsion de $100dt$.



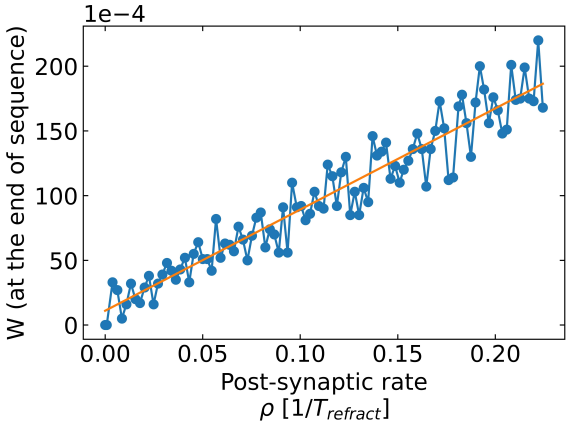
(b) Dépendance de la modification du poids synaptique à l'accélération de la fréquence avec une durée d'impulsion de $100dt$.

Figure 4.5 – Vérification de la dépendance de la modification du poids synaptique à la fréquence(a) et à l'accélération(b) des neurones pour la loi d'apprentissage *EqSpike* avec des impulsions d'une durée de $100dt$

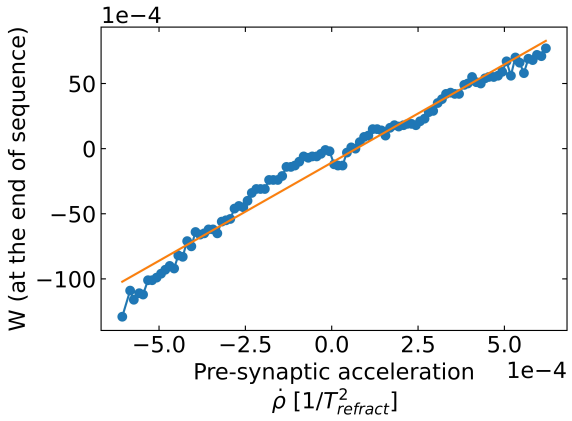
Pour régler une partie des problèmes qui rendent inexploitable cette implémentation, il est possible de limiter la valeur du signal émis par un neurone lors de l'accumulation des impulsions sous la valeur V_{th}, ce qui empêche un neurone de modifier le memristor de lui-même. Cette variante sera appelée **desynch**. Avec cette limitation de tension, si les deux neurones envoient un signal simultanément, alors le memristor ne sera pas modifié, car en bornant le signal d'un neurone à une valeur V_{th}, l'accumulation des queues d'impulsion disparaît pendant l'impulsion de programmation (pic central de l'impulsion).

Les Figures 4.6a et 4.6b montrent l'évolution du poids, ΔW, à la fin de la simulation, en fonction de la fréquence et de la dérivée de la fréquence pour la variante **desynch**. La régression linéaire de ces mesures

est représentée par la ligne orange. La loi d'apprentissage est bruitée pour la dépendance à la fréquence, ce qui est dû aux problèmes de superposition des impulsions de programmation.



(a) Dépendance de la modification du poids synaptique à la fréquence avec une durée d'impulsion d'une durée de $100dt$.



(b) Dépendance de la modification du poids synaptique à l'accélération de la fréquence avec une durée d'impulsion d'une durée de $100dt$.

Figure 4.6 – Vérification de la dépendance linéaire de l'évolution des poids synaptiques à la fréquence(a) et à l'accélération(b) des neurones pour la loi d'apprentissage *EqSpike* avec des impulsions d'une durée de $100dt$ pour la variante **desynch**.

En effet, lorsque deux neurones émettent une impulsion en même temps, le signal reçu par le memristor, soit l'addition des deux impulsions, est équivalent à $2V_{th}+\epsilon$, avec ϵ la somme des queues d'impulsion (dans le cas avec des signaux bornés, le signal est $\leq 2V_{th}$). Ce signal va modifier d'une valeur proportionnelle à $V_{th}+\epsilon$ la conductance du memristor, puis, dû à la partie négative des impulsions, le memristor sera ensuite modifié d'une valeur proportionnelle à $-V_{th}+\epsilon$. La Figure 4.7 montre deux impulsions simultanées de deux neurones avec $\epsilon = 0$. La partie rouge représente la valeur supérieure à V_{th} qui sera écrite sur le poids synaptique. Si le temps d'impulsion est très court comparé à la durée d'une impulsion de communication, alors le changement brusque sera peu perceptible, en revanche si les deux temps sont du même ordre de grandeur, il est possible que ce changement affecte la dynamique du réseau. Ce phénomène n'est pas problématique dans un cas où les signaux ne sont pas bornés et où les poids ne sont pas modifiés dynamiquement pendant l'exécution, car ϵ sera écrit sur le memristor et il n'y aura pas de changement brusque des poids qui modifierait la dynamique du réseau. Malheureusement, ce n'est pas le cas pour *A-EqSpike* qui modifie les poids synaptiques pendant l'exécution et a besoin de signaux bornés pour l'estimation de sa loi d'apprentissage.

De plus, la superposition des impulsions de programmation peut provenir des choix d'implémentation du simulateur. Dans le cas d'une implémentation où les neurones sont désynchronisés, la probabilité que deux impulsions soient émises au même moment, événement que nous appellerons *collision*, est faible. En simulation, la valeur temporelle la plus petite exprimable par le simulateur est le pas de temps dt, ce qui conditionne le risque de collision des impulsions, alors qu'il est possible que le temps d'impulsion de programmation voulu soit inférieur à dt. Si une impulsion est émise à un temps t et une autre à un temps $t + \epsilon_t$ avec $\epsilon_t < dt$ alors il y aura une collision dans la simulation qui ne devrait pas exister en pratique. Une solution serait de réduire au maximum la durée d'un pas de temps pour représenter plus fidèlement le comportement, mais cela augmenterait le temps, déjà long, de la simulation. Nous proposons par la suite deux méthodes pour réduire le risque de superposition des impulsions de programmation.

Pour limiter la superposition des impulsions de programmation, nous proposons une variante, appelée **desynch_proba**. L'impulsion de programmation pour activer le seuil du memristor sera générée aléatoirement, avec une probabilité p. La Figure 4.8 montre l'accumulation d'impulsions de deux neurones avec une génération de l'impulsion de programmation $p = 0$ à gauche et $p = 0.5$ à droite. La partie rouge correspond au moment où une synapse est écrite avec une superposition de l'impulsion de programmation des deux neurones, ce que l'on cherche à éviter. La probabilité de superposition des impulsions de programmation, si les neurones émettent en même temps, devient alors p^2 mais la probabilité de ne pas écrire le memristor est p. Cette solution réduit seulement l'impact du problème de superposition des impulsions de

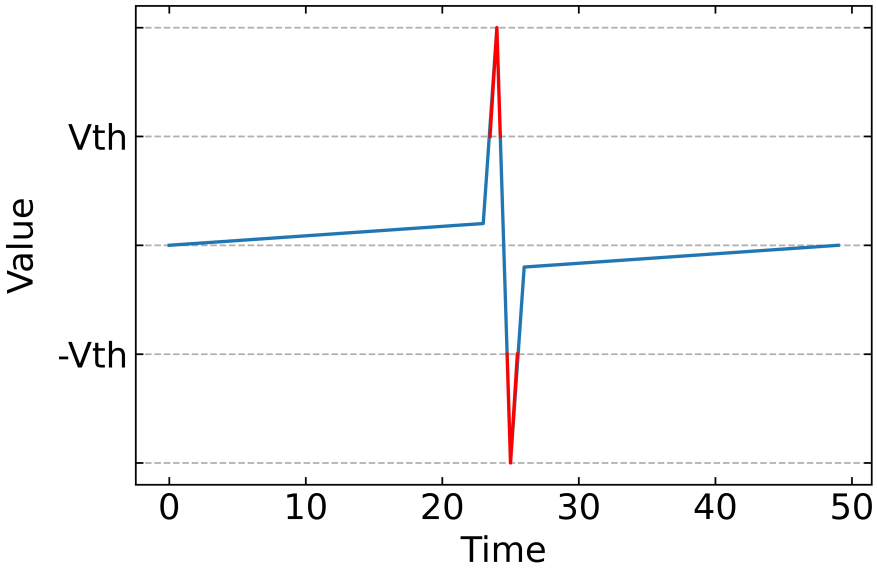


Figure 4.7 – Superposition d'une impulsion présynaptique et postsynaptique simultanées. La partie rouge correspond à une tension qui permet de modifier la conductance du memristor.

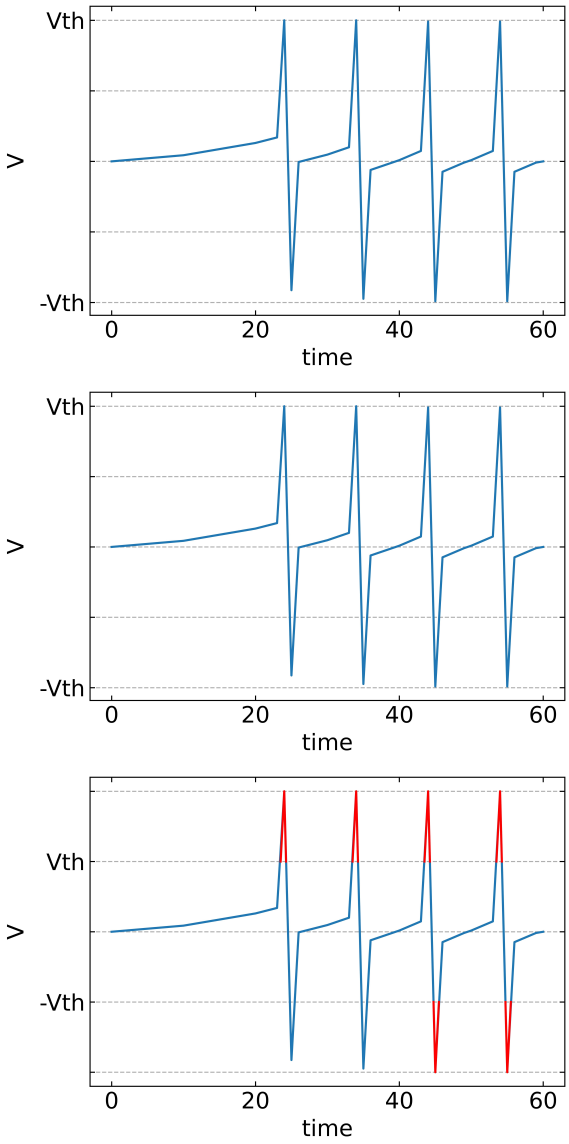
programmation d'un facteur p^2 tout en rajoutant du bruit lors de l'application de la règle d'apprentissage au memristor. Le problème de cette méthode est la nécessité de pouvoir générer aléatoirement les impulsions de programmation.

Nous avons considéré deux probabilités différentes, $p = 0.3$ et $p = 0.5$. La Figure 4.9 montre l'évolution du poids, ΔW, à la fin de la simulation en fonction de la fréquence et de la dérivée de la fréquence, avec l'apparition d'une impulsion de programmation aléatoire. Les ensembles de graphes 4.9 représentent la dépendance des paramètres à la loi d'apprentissage pour deux probabilités p différentes. Avec une probabilité de génération d'impulsion de programmation de 50%, la dépendance à l'accélération est plus bruitée qu'avec une probabilité de 30%. En revanche, la dépendance à la fréquence est plus bruitée avec une probabilité faible de génération d'impulsion de programmation. De plus, il est à noter qu'avec la réduction du nombre de modifications des synapses due à la réduction du nombre d'impulsions de programmation, il faut modifier le taux d'apprentissage pour avoir un ΔW équivalent aux variantes sans probabilités.

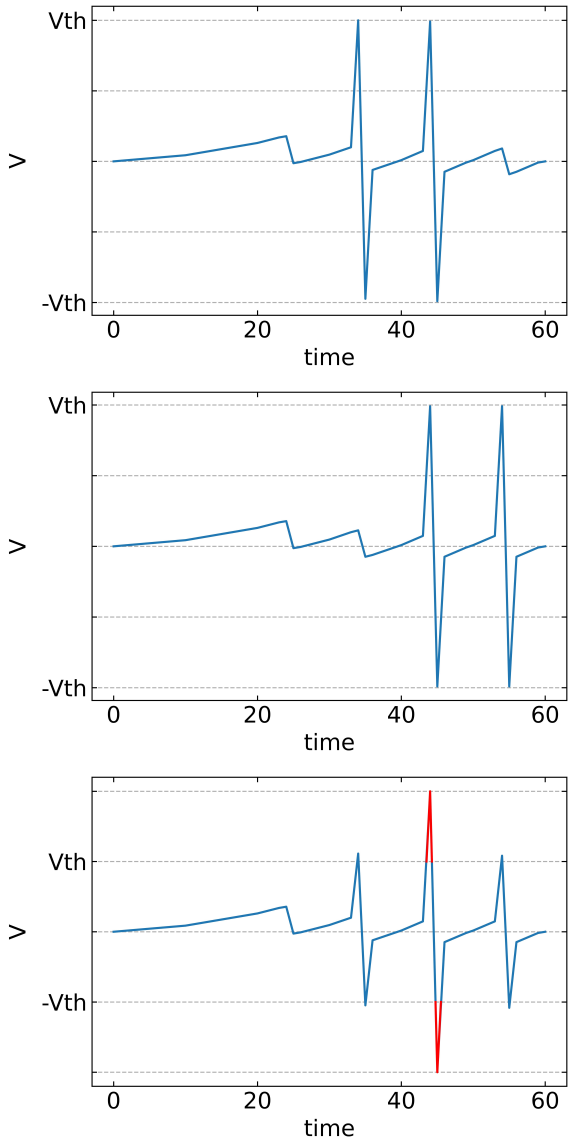
Pour éviter de générer aléatoirement les impulsions de programmation, ce qui demande une implémentation spécifique, nous nous plaçons dans le cas théorique où le temps de l'impulsion de programmation, de durée τ_{imp}, est inférieur à dt, il est alors possible de simuler approximativement le risque de *collision* des impulsions de programmation. Nous proposons une variante, appelée **desynch_collision**, qui consiste à superposer les impulsions de programmation des impulsions simultanées avec une probabilité $p_{collision} = \frac{dt}{\tau_{imp}}$. La superposition d'impulsions peut être simulée avec un tirage aléatoire de probabilité $p_{collision}$ lors d'une impulsion de programmation en partant de l'hypothèse que le début de l'impulsion des neurones est aléatoirement réparti dans un pas de temps. Cette solution permet ainsi de gagner en temps d'exécution au prix d'une approximation des possibilités de collision et une hypothèse sur τ_{imp} comparé à dt.

Les Figures 4.10a et 4.10b montrent l'évolution du poids, ΔW, à la fin de la simulation en fonction de la fréquence et de la dérivée de la fréquence pour des neurones désynchronisés avec un risque de collision de $\frac{1}{10}$ pour la variante **desynch_collision**. On remarque que dans ce scénario, la loi d'apprentissage dépend en effet linéairement de la fréquence et de l'accélération conformément à l'algorithme *EqSpike*. Cette variante est donc valide pour implémenter *EqSpike* avec la superposition des impulsions si ces hypothèses sont applicables.

Les variantes d'*A-EqSpike* présentées précédemment sont applicables pour des réseaux de neurones désynchronisés. Il est possible de mieux approximer la règle d'apprentissage en utilisant des réseaux de neurones où les neurones sont synchronisés, ce qui rajoute en revanche des contraintes pour l'implémentation physique. Nous proposons une variante de *A-EqSpike*, appelée **synch**, où les impulsions sont synchronisées et discrétisées. Dans les simulations, les impulsions continues sont représentées discrétisées du fait des contraintes temporelles de la simulation, ce qui n'est pas forcément le cas dans une implémentation physique.

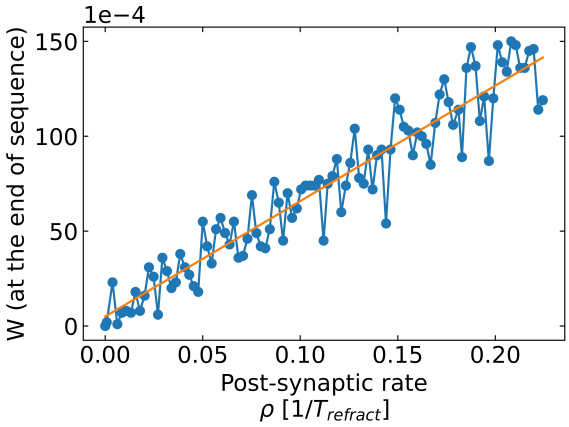


(a) Superposition du signal de deux neurones qui émettent des impulsions au même instant, le signal du neurone i est en haut, le signal du neurone j au milieu, et en bas la somme des deux signaux perçus par la synapse.

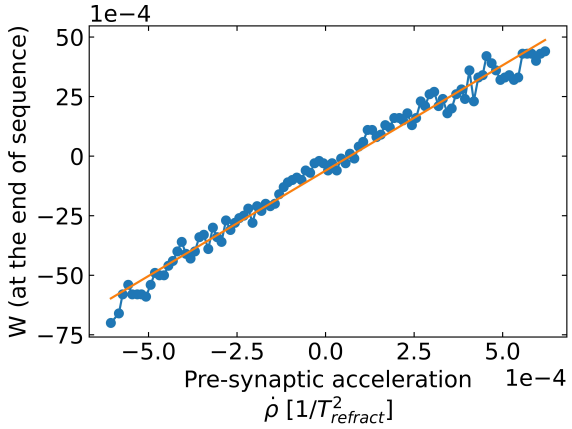


(b) Superposition du signal de deux neurones qui émettent des impulsions au même instant, le signal du neurone i est en haut, le signal du neurone j au milieu, et en bas la somme des deux signaux perçus par la synapse. La génération de l'impulsion de programmation est avec $p = 0.5$.

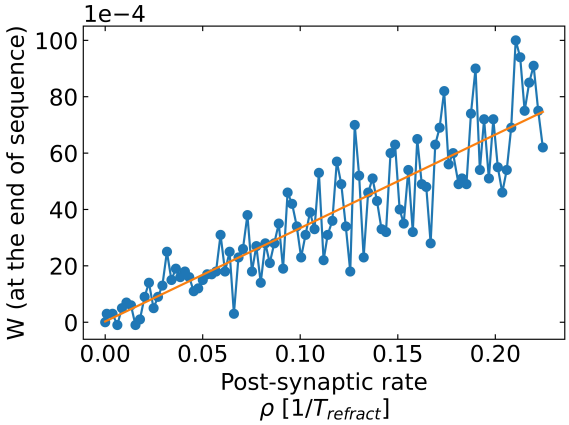
Figure 4.8 – Illustration de l'accumulation d'impulsions de deux neurones avec une génération de l'impulsion de programmation $p = 0$ à gauche et $p = 0.5$ à droite. La partie rouge correspond au moment où une synapse est écrite avec une superposition de l'impulsion de programmation des deux neurones.



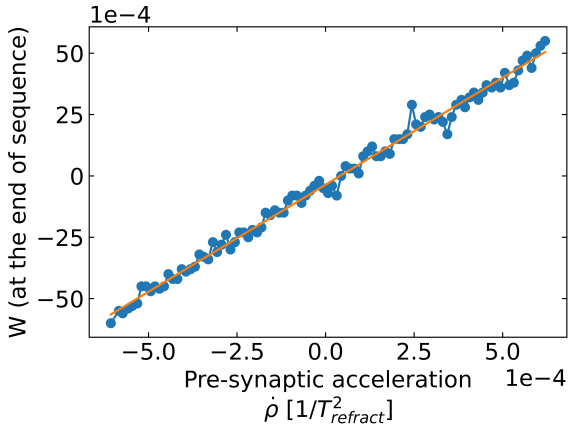
(a) Dépendance de la modification du poids synaptique à la fréquence avec une durée d'impulsion d'une durée de $100dt$ et pour une probabilité de génération de l'impulsion de programmation $p = 0.5$.



(b) Dépendance de la modification du poids synaptique à l'accélération de la fréquence avec une durée d'impulsion d'une durée de $100dt$ et pour une probabilité de génération de l'impulsion de programmation $p = 0.5$.

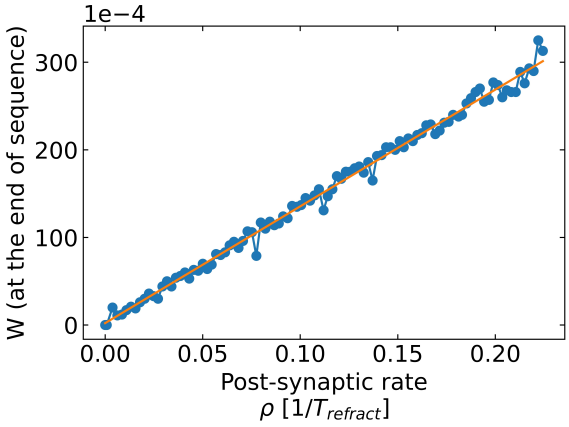


(c) Dépendance de la modification du poids synaptique à la fréquence avec une durée d'impulsion d'une durée de $100dt$ et pour une probabilité de génération de l'impulsion de programmation $p = 0.3$.

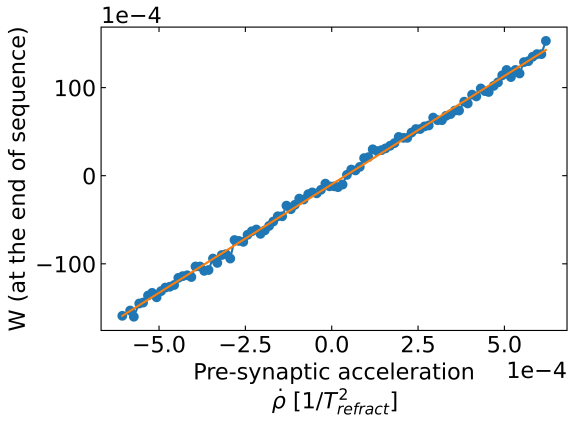


(d) Dépendance de la modification du poids synaptique à l'accélération de la fréquence avec une durée d'impulsion d'une durée de $100dt$ et pour une probabilité de génération de l'impulsion de programmation $p = 0.3$.

Figure 4.9 – Vérification de la dépendance linéaire de l'évolution des poids synaptiques à la fréquence(a, c) et à l'accélération(b, d) des neurones pour la loi d'apprentissage *EqSpike* avec des impulsions d'une durée de $100dt$ pour la variante **desynch_proba** avec une probabilité de générer la phase de transitoire de 50% et 30%, respectivement.



(a) Dépendance de la modification du poids synaptique à la fréquence avec une durée d'impulsion d'une durée de $100dt$.



(b) Dépendance de la modification du poids synaptique à l'accélération de la fréquence avec une durée d'impulsion d'une durée de $100dt$.

Figure 4.10 – Vérification de la dépendance linéaire de l'évolution des poids synaptiques à la fréquence(a) et à l'accélération(b) des neurones pour la loi d'apprentissage *EqSpike* avec des impulsions d'une durée de $100dt$ pour la variante **desynch_collision** avec un risque de collisions de 10%.

Nous considérons des impulsions discrétisées donc qui alternent entre 0 et leur valeur finie de l'impulsion comme montré sur la Figure 4.11. En synchronisant les neurones pour envoyer alternativement la valeur des impulsions de chaque neurone, et en synchronisant les impulsions de programmation avec l'autre neurone connecté, alors il devient impossible qu'un memristor soit modifié grâce à un seul neurone et que les impulsions de programmation se superposent. La Figure 4.11 illustre la nouvelle impulsion de chaque neurone et leur alternance dans le temps. Avec cela, les queues d'impulsion d'un neurone ne peuvent pas être additionnées avec une impulsion de programmation du même neurone, et les impulsions de programmation des deux neurones ne peuvent pas se superposer. Cette implémentation nécessite d'avoir tous les neurones synchronisés, mais permet d'être plus proche de la règle d'apprentissage au détriment de la complexité d'implémentation.

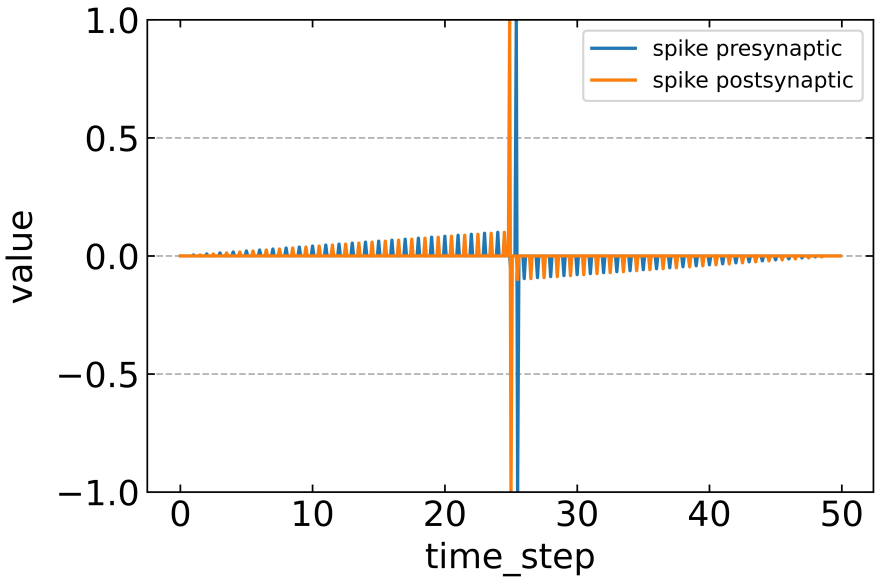
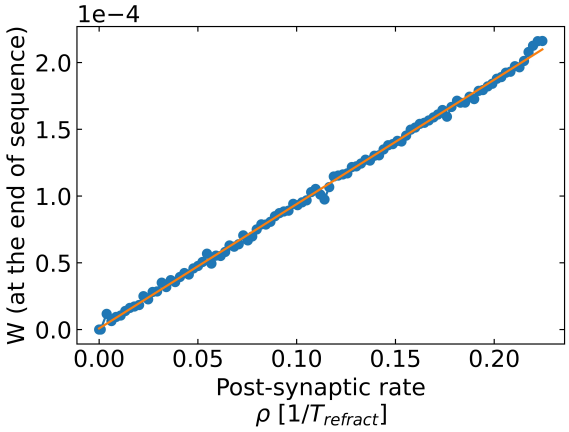


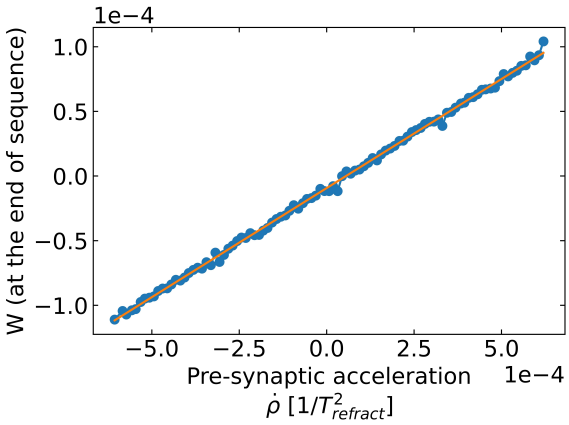
Figure 4.11 – Forme d'impulsion pour un réseau bidirectionnel pour écrire les memristors et calculer la dérivée de la fréquence, avec des impulsions synchronisées.

Les Figures 4.12b et 4.12a montrent l'évolution du poids, ΔW, à la fin de la simulation en fonction de la fréquence et de la dérivée de la fréquence pour une forme de durée $100dt$ (pas de temps). La loi

d'apprentissage est bien approximée, avec la variante **synch**, pour les dépendances linéaires à la fréquence et à son accélération.



(a) Dépendance de la modification du poids synaptique à la fréquence avec une durée d'impulsion de $100dt$.



(b) Dépendance de la modification du poids synaptique à l'accélération de la fréquence avec une durée d'impulsion de $100dt$.

Figure 4.12 – Vérification de la dépendance de la modification du poids synaptique à la fréquence (a) et à l'accélération (b) des neurones pour la loi d'apprentissage *EqSpike* avec des impulsions d'une durée de $100dt$ pour la variante **synch**.

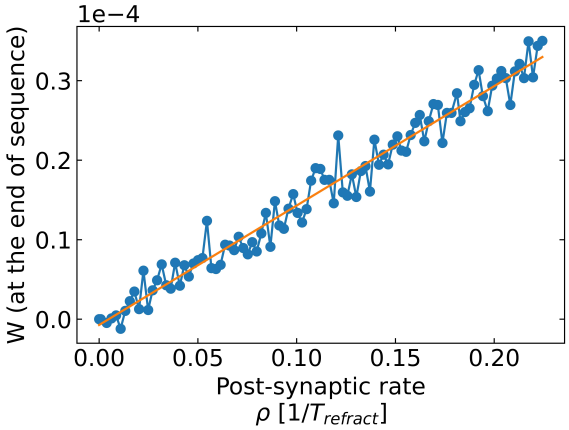
Il est important de noter que la durée de l'impulsion correspond à la taille de la fenêtre temporelle de l'accumulation, ce qui conditionne l'accélération et la fréquence minimale observable avec précision par cette méthode. En effet, sur l'Équation 4.1 rappelée ici :

$$V_{impulsion}(t) = \sum_{i=t-\frac{\gamma_{len}}{2}}^{t} \delta(i) - \sum_{i=t-\gamma_{len}}^{t-\frac{\gamma_{len}}{2}} \delta(i),$$

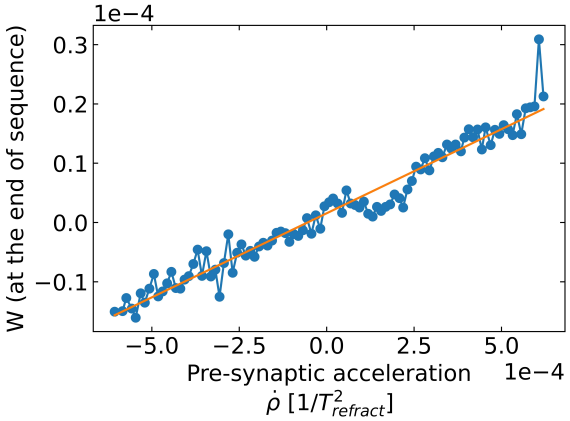
l'accumulation des impulsions ne se fait que sur une fenêtre de temps $\frac{\gamma_{len}}{2}$. En revanche, dans le but d'une implémentation neuromorphique à basse consommation, il est important de noter que des impulsions longues consomment plus d'énergie que des impulsions courtes (pour une amplitude similaire), un compromis précision/énergie est donc nécessaire pour une implémentation physique.

Afin d'estimer l'impact de la durée de l'impulsion sur la précision de reconnaissance, nous avons regardé la dépendance à la règle d'apprentissage pour une durée d'impulsion plus courte de $40dt$. Les Figures 4.13a et 4.13b sont obtenues avec une durée d'impulsion de $40dt$. La mise à jour des poids est bruitée comparée à la dépendance linéaire théorique pour des temps d'impulsion plus longs. Ceci illustre bien le compromis entre la précision et le gain temps-énergie. Mais même avec ce bruit, la règle d'apprentissage reste bien approximée. Le bruit peut être expliqué par des transitions plus abruptes entre chaque point de la queue d'impulsion quand l'impulsion est plus courte en temps pour la même amplitude maximum.

Nous avons montré que la loi d'apprentissage de *EqSpike* est bien approximée avec la nouvelle implémentation proposée pour les variantes étudiées. Pour une meilleure approximation, il est préférable d'avoir des impulsions de longue durée, mais un compromis doit être fait entre la durée d'impulsion et le temps de calcul et par conséquent l'énergie dépensée. Nous avons donc choisi de rester sur des signaux d'environ $40dt$ pour la suite.



(a) Dépendance de la modification du poids synaptique à la fréquence avec une durée d'impulsion de $40dt$.



(b) Dépendance de la modification du poids synaptique à l'accélération de la fréquence avec une durée d'impulsion de $40dt$.

Figure 4.13 – Vérification de la dépendance de la modification du poids synaptique à la fréquence(a) et a l'accélération(b) des neurones pour la loi d'apprentissage *EqSpike* avec des impulsions de $40dt$ avec la variante **synch**.

4.3 Classification de chiffres manuscrits avec *A-EqSpike*

Dans la section précédente, nous avons montré que la loi d'apprentissage d'*A-EqSpike* peut être approximée avec des formes d'impulsion bien choisies qui s'accumulent dans le temps. Dans cette partie, nous allons dans un premier temps comparer la précision de classification des variantes d'*A-EqSpike* entre eux sur *Digits*, puis nous regarderons la précision de classification sur *MNIST* pour la variante la plus performante que nous comparerons à *EqSpike*, *Eq-prop* et *BPTT*. Enfin, nous analyserons l'impact des formes de l'impulsion sur le problème de classification d'image *Digits* avec la variante **synch**.

4.3.1 Résultats obtenus sur le jeu de données *Digits*

Nous allons comparer les différentes variantes de *A-EqSpike* sur le jeu de données de chiffres manuscrits *Digits*, pour observer la précision obtenue des variantes en fonction de leur complexité. La précision des variantes est résumée dans le Tableau 4.3. La précision de classification a été obtenue comme la précision moyenne de 10 réseaux de 64 neurones d'entrée, 32 neurones cachés, et 10 neurones de sortie, initialisés aléatoirement et 30 époques d'apprentissage pour chaque réseau. Dans le Tableau 4.1, nous présentons les hyperparamètres utilisés pour obtenir la meilleure précision **synch** sur le jeu de données *Digits*, soit une précision de $97.1\% \pm 0.62\%$. Les paramètres utilisés pour les autres variantes sont représentés dans le Tableau 4.2

Pas de temps, dt	γ_{LIF}	γ_{LI}	Amplitude de l'impulsion, γ_{amp}
0.5	0.01	0.1	0.0001
Longueur impulsion, γ_{len} $(T_{refract})$	T_{free} $(T_{refract})$	T_{nudge} $(T_{refract})$	η_{lr}
20	100	62.5	1

Table 4.1 – Hyperparamètres utilisés pour la classification d'images de la base *Digits* avec *A-EqSpike* pour **synch**.

La précision de classification sur le jeu de donnée *Digits* pour la variante **desynch**, qui est la variante la moins complexe, est de $94.75\% \pm 0.93\%$, soit 3% moins que l'état de l'art. La variante **desynch_proba** augmente ce résultat, mais obtient une précision moins bonne que la variante **desynch_collision** qui obtient

Pas de temps, dt	γ_{LIF}	γ_{LI}	Amplitude de l'impulsion, γ_{amp}
0.5	0.01	0.1	0.1
Longueur impulsion, γ_{len} $(T_{refract})$	T_{free} $(T_{refract})$	T_{nudge} $(T_{refract})$	η_{lr}
20	100	62.5	0.001

Table 4.2 – Hyperparamètres utilisés pour la classification d'images de la base *Digits* avec *A-EqSpike* pour les algorithmes désynchronisés.

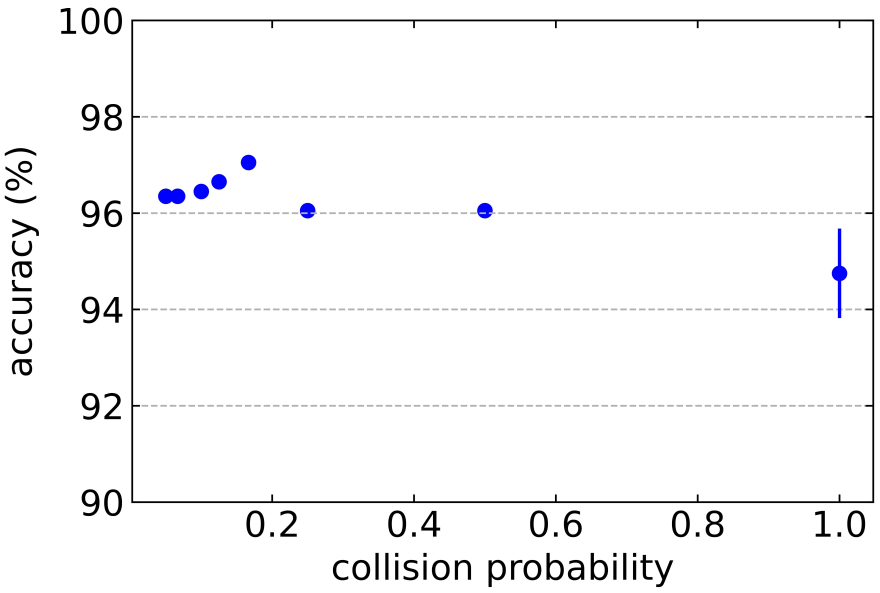


Figure 4.14 – Précision de classification (%) sur la base de données *Digits* avec la variante **desynch_collision**.

des résultats similaires à la variante synchronisée **synch** montrant que cette variante évite bien les problèmes mentionnés sur l'algorithme *A-EqSpike* naïf.

desynch	**desynch_collision**	**desynch_proba**	**synch**
$94.75\% \pm 0.93\%$	$97.05\% \pm 0.35\%$	$95.35\% \pm 0.83\%$	$97.1\% \pm 0.62\%$

Table 4.3 – Précision de classification d'images sur la base *Digits* pour les différentes variantes d'*A-EqSpike*.

La Figure 4.14 montre la précision de classification sur le jeu de données *Digits* pour la variante **desynch_collision** avec collision, pour différentes probabilités de collision. La performance maximale est de $97.05\% \pm 0.35\%$ pour une probabilité de collision de $\frac{1}{6}$ (en prenant en compte la variance des résultats, une probabilité de collision inférieure donne des résultats similaires) soit équivalente à l'état de l'art et supérieure à la variante **desynch**. Cette variante montre que le bruit présent lors de la vérification de la dépendance à la fréquence et à l'accélération des neurones ne perturbe pas de manière significative la précision de classification.

La Figure 4.15 montre la précision de classification sur le jeu de données *Digits* pour la variante **desynch_proba** avec une probabilité de génération de l'impulsion de programmation, pour différentes probabilités. La performance maximale est de $95.35\% \pm 0.83\%$ pour une probabilité de génération de 50% soit légèrement moins que l'état de l'art, mais légèrement supérieure à la variante **desynch**. Pour compenser le nombre de modifications du memristors en moins dû à la probabilité de génération de l'impulsion de programmation, le taux d'apprentissage est inversement proportionnel à p. Cette variante possède les mêmes avantages que la variante d'*EqSpike* qui modifie le poids synaptique de manière stochastique. De plus, il y

a une légère augmentation de précision de classification comparé à la version de base **desynch**.

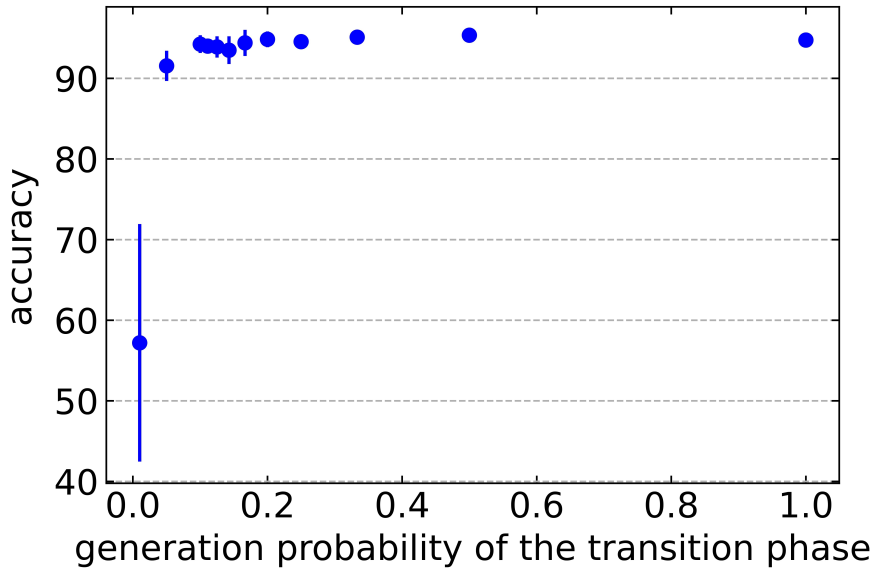


Figure 4.15 – Précision de classification (%) sur la base de données *Digits* avec la variante **desynch_proba**.

4.3.2 Résultats obtenus sur le jeu de données *MNIST*

Pour calibrer l'algorithme, la même méthode que pour *EqSpike* a été utilisée. Nous avons choisi le jeu d'hyperparamètres donnant la meilleure précision de classification pour la base d'images *Digits*, puis nous avons transféré les paramètres pour les images *MNIST* en augmentant la taille du réseau et en réduisant le taux d'apprentissage.

L'implémentation avec la variante **synch** obtient des performances de classification similaires à l'état de l'art sur *Digits*, soit les meilleures performances de classification des différentes variantes d'*A-EqSpike*. Nous vérifions les performances obtenues sur la base standardisée *MNIST*. Les paramètres utilisés sur *MNIST*, simulé avec un réseau de neurones 784x300x10, sont indiqués dans le Tableau 4.4.

Pas de temps, dt	γ_{LIF}	γ_{LI}	Amplitude de l'impulsion, γ_{amp}
0.5	0.01	0.1	1×10^{-6}
Longueur impulsion, γ_{len} $(T_{refract})$	T_{free} $(T_{refract})$	T_{nudge} $(T_{refract})$	η_{lr}
50	100	300	1

Table 4.4 – Hyperparamètres pour la variante **synch** pour *MNIST*.

La table 4.5 montre la précision des différents algorithmes, la précision de la variante de **synch**, après 100 époques, est de $96,71\% \pm 0.26\%$. L'approximation de la loi d'apprentissage conduit à une précision moindre par rapport à *EqSpike* ou *BPTT*. La Figure 4.12b suggère qu'une plus grande précision serait possible avec une impulsion de programmation plus longue et d'autres paramètres ajustés en conséquence. Cependant, ce serait au prix d'une augmentation du temps et du coût énergétique de l'exécution sur la puce matérielle.

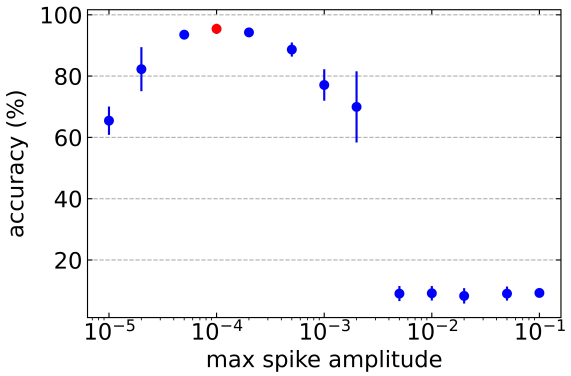
Malgré une légère baisse de précision, la variante **synch** de *A-EqSpike* obtient de bons résultats sur la base de données *MNIST*, ce qui fait de *A-EqSpike* une variante valide pour l'apprentissage intrinsèque des réseaux de neurones avec des synapses memristives.

Algorithme	*A-EqSpike* ***synch*** 300 784-300-10	*EqSpike* 300 784-300-10
MNIST	Test : $96.71\% \pm 0.26\%$ Train : $99.1\% \pm 0.17\%$	Test : $97.59\% \pm 0.1\%$ Train : $98.91\% \pm 0.03\%$

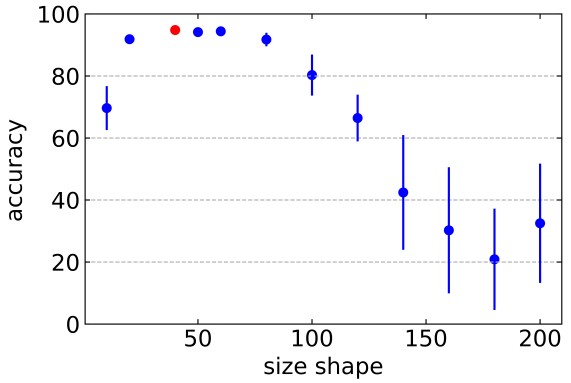
Table 4.5 – Comparaison des résultats entre *BPTT*, *Continual Equilibrium Propagation*, *EqSpike* et *A-EqSpike* ***synch*** avec la même procédure d'initialisation.

4.3.3 Étude de l'impact de l'amplitude et de la durée de la forme d'impulsion sur la classification d'image

La dépense en énergie de l'impulsion est la dépense en énergie principale des neurones. Dans une optique d'optimisation énergétique, il faut chercher un compromis entre la durée/amplitude de l'impulsion et la performance de classification. Dans cette section, nous allons regarder la précision du réseau avec différentes amplitudes et durées d'impulsion avec la variante d'*A-EqSpike* qui obtient la meilleure précision sur *Digits*, **synch**. Les hyperparamètres ont été optimisés à la main, ce qui a donné les valeurs présentées au tableau 4.2, et explique que les meilleures performances dans les figures qui suivent sont proches de ces valeurs.



(a) Précision de classification (%) sur 10 *runs* sur *Digits* avec différentes amplitudes d'impulsion γ_{amp}.



(b) Précision de classification(%) sur 10 *runs* sur *Digits* avec différentes durées d'impulsion γ_{len}.

Figure 4.16 – Précision de classification (%) sur 10 *runs* sur *Digits* avec différents paramètres sur les formes d'impulsion des neurones.

La Figure 4.16a donne la précision de classification (en %) sur le jeu de données *Digits* en fonction de l'amplitude maximum des queues d'impulsion. Les performances se dégradent si l'amplitude a plusieurs ordres de grandeur de différence avec la meilleure amplitude testée de $\gamma_{amp} = 0.0001 V_{th}$. Augmenter l'amplitude augmente la surface positive de l'impulsion, ce qui augmente l'amplitude de la modification de la conductance du memristor conformément à l'équation de modification de sa conductance. La valeur de la modification des poids synaptique est équivalente à l'accumulation des queues d'impulsion, une amplitude plus élevées des queues d'impulsions signifie donc une accumulation de valeurs plus importante lors d'une augmentation de la fréquence. Il faudrait donc ajuster le taux d'apprentissage, qui est constant dans chaque *run*, pour compenser l'augmentation ou la diminution de l'amplitude.

La Figure 4.16b nous montre la précision de l'algorithme sur le jeu de données *Digits* en fonction de la durée de l'impulsion. Pour une comparaison juste, la durée de la phase d'apprentissage est fixée à trois fois la durée d'une impulsion de façon à ce que les impulsions durant la stabilisation des fréquences des neurones au début de la phase d'apprentissage puissent se terminer avant la fin de la phase d'apprentissage. La précision obtenue avec $\gamma_{len} = 100dt$ est inférieure à la précision pour $40dt$ contrairement à ce qui pouvait être attendu avec les résultats observés lors de la validation de la règle d'apprentissage, où les impulsions longues diminuaient le bruit lors de l'estimation de la loi d'apprentissage. Cette observation s'explique car les hyperparamètres ont été optimisés pour une durée de $40dt$, en particulier le taux d'apprentissage. En effet, la durée des queues d'impulsion influe sur la modification des memristors en permettant à plus de queues

d'impulsion de se superposer, comme montré sur l'Équation 4.2. En augmentant γ_{len}, les modifications des poids sont devenues trop importantes à chaque itération pour que le réseau converge vers un jeu de poids avec de bonnes performances de reconnaissance. Pour modifier la durée des queues d'impulsion sans changer le taux d'apprentissage, il faut modifier l'amplitude maximale de celle-ci en conséquence.

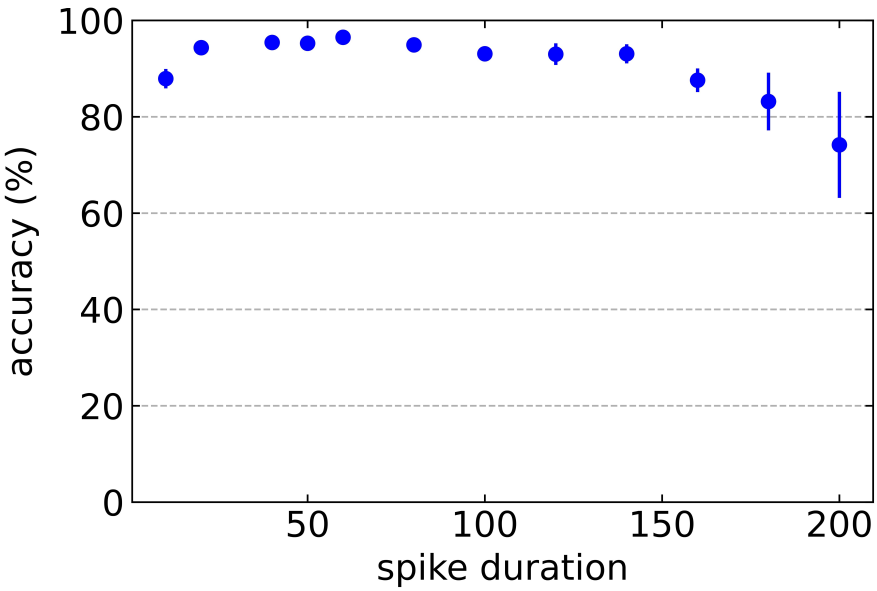


Figure 4.17 – Précision de classification sur 10 *runs* sur digits avec différentes longueurs d'impulsion et l'amplitude adaptée pour toujours avoir $\frac{\gamma_{amp}}{\gamma_{len}} = 0.000025$.

La Figure 4.17 montre la précision de l'algorithme sur le jeu de données *Digits* en fonction de la durée γ_{len} avec l'intégrale de la partie positive de la forme de l'impulsion identique pour chaque expérience. C'est-à-dire avec un ratio $\frac{\gamma_{amp}}{\gamma_{len}}$ constant, tel que :

$$\gamma_{imp} = \frac{\gamma_{amp}}{\gamma_{len}} = \frac{0.0001}{40} \tag{4.4}$$

À γ_{imp} constant, la précision du réseau est plus robuste au changement de durée d'impulsion que si l'amplitude reste la même (Figure 4.16b). Ce résultat n'est pas totalement surprenant, car les memristors sont modifiés grâce à la somme des amplitudes des queues d'impulsion. En gardant un ratio amplitude/taille constant, on garde une approximation du gradient d'erreur dans les mêmes ordres de grandeur.

4.4 Lien avec la *STDP*

Comme pour *EqSpike*, nous explorons par la suite l'émergence de la loi STDP dans l'évolution des poids de *A-EqSpike*. Avec cet algorithme, la modification de la synapse se fait grâce à la superposition des impulsions présynaptiques et postsynaptiques, de façon similaire aux méthodes montrées section 1.4.3. Cette implémentation est plus biologiquement plausible et plus adaptée pour des réseaux avec des memristors.

La dépendance de la synchronisation entre deux neurones de la première couche est calculée par la méthode de mesure utilisée dans le chapitre 2 et l'algorithme 3 présenté dans le même chapitre, pour 300 images. Cette première couche a des synapses unidirectionnelles, ce qui permet de définir des neurones présynaptiques et des neurones postsynaptiques pour la variante *synch*. À chaque émission d'une impulsion postsynaptique, l'algorithme calcule le temps moyen des impulsions présynaptiques dans une fenêtre temporelle autour de l'impulsion postsynaptique. Ensuite l'algorithme enregistre une paire $t_{post} - t_{pre}$ et δ_W, chaque point rouge de la Figure 4.18 correspond à une de ces paires.

Une modification par rapport à $t_{post} - t_{pre}$ de l'algorithme *STDP* est visible sur la Figure 4.19, qui est une moyenne des paires calculées précédemment avec $t_{post} - t_{pre}$ arrondie à un entier. Il y a une forte concentration de points proches de zéro sur la Figure 4.18 ce qui explique les faibles modifications en moyenne sur la Figure 4.19. Plus $t_{post} - t_{pre}$ s'éloigne de 0, plus les points sont corrélés à la loi d'apprentissage de la

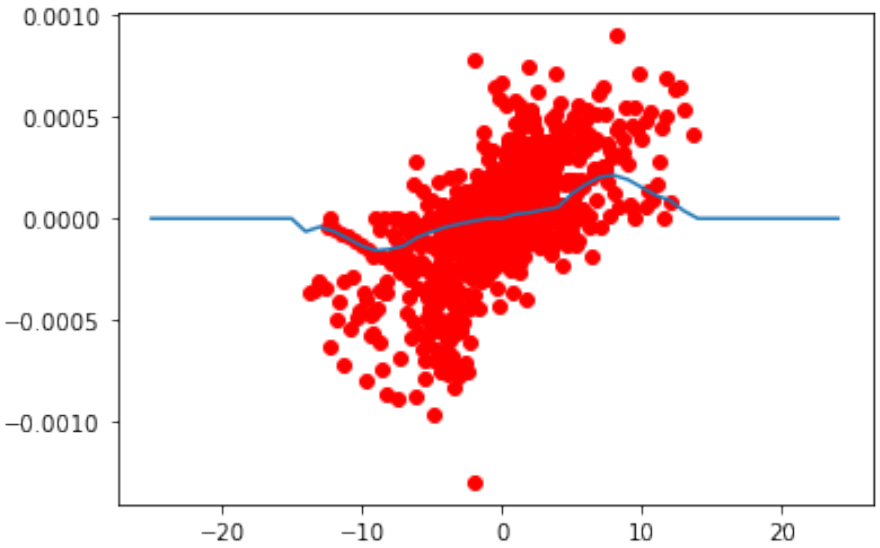


Figure 4.18 – Lien entre la *STDP* et *A-EqSpike* avec des impulsions superposées. Les points rouges représentent une modification du poids synaptique.

STDP. Les impulsions des neurones suffisent à modifier la valeur du poids synaptique selon leurs différences temporelles en accord avec la règle d'apprentissage tout en obtenant une corrélation avec une approximation de la loi d'apprentissage de la *STDP* (1.11).

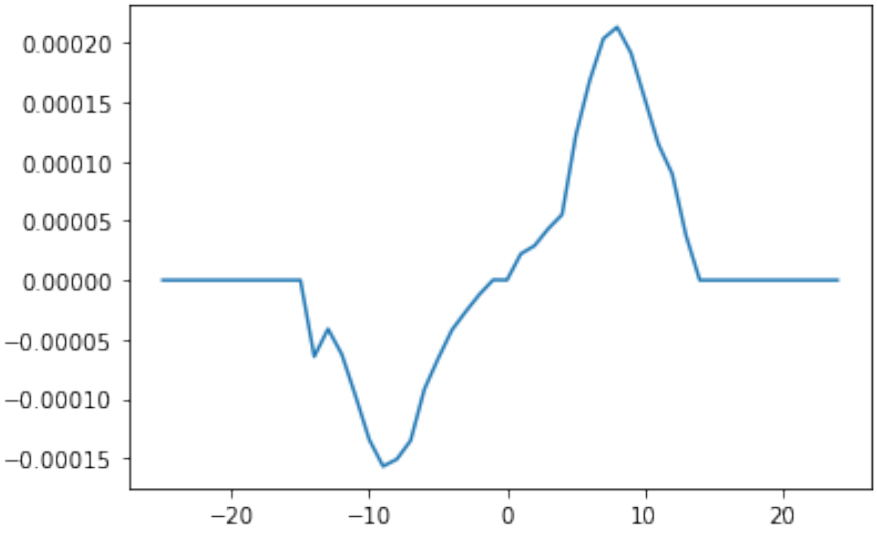


Figure 4.19 – Lien entre la *STDP* et *A-EqSpike* sur 300 images.

Cette section a montré le lien entre *STDP* et notre algorithme, ce qui renforce la bio-plausibilité de notre implémentation d'*EqSpike* avec une superposition d'impulsions.

4.5 Conclusion

Nous avons présenté une nouvelle méthode pour implémenter *EqSpike* à travers la superposition des impulsions des neurones dans un réseau de neurones avec des memristors comme synapses. Avec une forme adéquate d'impulsion, la loi d'apprentissage est bien approximée. Nous avons proposé différentes variantes d'*A-EqSpike* pour l'implémentation de l'algorithme pour essayer de résoudre les problèmes de superposition des impulsions de programmation ainsi que les problèmes dûs à la bidirectionnalité. Nous avons étudié l'impact de la forme des impulsions sur la précision de classification.

Parmi les 4 variantes, la variante synchronisée **synch** et la variante avec une probabilité de collision **desynch_collision** ont obtenu des performances au niveau de l'état de l'art ($\approx 97\%$) sur la base de

données *Digits*. En revanche, la précision de classification de la variante probabiliste **desynch_proba** atteint seulement 95.35%. La variante **synch** a obtenu des résultats 1 à 2% en dessous de l'état de l'art pour des réseaux de même taille sur la base de données *MNIST*, ceci étant en partie dû à un temps de simulation long qui a empêché un meilleur échantillonnage temporel.

A-EqSpike est une approche différente de *EqSpike* qui exploite une forme particulière d'impulsions pour implémenter directement la loi d'apprentissage via leur superposition. Elle s'affranchit du bloc nécessaire au calcul de la dérivée dans *EqSpike*, mais nécessite des formes d'impulsion plus compliquées à mettre en œuvre électroniquement en raison de leurs formes particulières et du besoin de les garder en mémoire pour les superposer. De futures études seront nécessaires pour déterminer les meilleures implémentations physiques possibles. En particulier, contrairement à *EqSpike*, il n'est pas possible de placer un filtre à la sortie du neurone pour lisser les modifications afin d'avoir une dépendance à la fréquence moins bruitée. En effet, les modifications des poids synaptique avec *A-EqSpike* sont plus bruitées à cause du changement abrupt de signe lors de l'impulsion de programmation, mais insérer un filtre passe-bas est compliqué, car cela filtrerait aussi les impulsions de programmation, ce qui empêcherait une modification des memristors. Il serait intéressant pour le futur de cette approche, de réussir à implémenter l'équivalent d'un filtre passe-bas pour réduire le bruit et ainsi augmenter les performances de classification.

Chapitre 5

Conclusion et perspectives

5.1 Conclusion

Les réseaux de neurones artificiels avec des architectures telles que les *GPUs* ou les *TPUs* consomment dix à cent fois plus d'énergie par carte qu'un cerveau biologique pour des performances à résoudre des taches complexes limitées en comparaison. Une des raisons de cette surconsommation est due à l'architecture *Von Neumann* des puces *CMOS*. En effet, un accès mémoire consomme plusieurs centaines, voire plusieurs milliers de fois l'énergie d'une opération comme une addition [Sze et al., 2017]. Le développement de puces neuromorphiques, sans architecture *Von Neumann* et avec des mémoires non volatiles pour les synapses, comme des memristors, est prometteur pour une réduction d'énergie et de temps de calcul des réseaux de neurones en intégrant la mémoire au calcul. En revanche, l'apprentissage sur puce reste compliqué, car l'implémentation de *BP* est associée à plusieurs défis tels que la nécessité d'embarquer sur la puce des mémoires et des circuits externes au réseau. Remplacer *BP* par un autre algorithme capable d'obtenir les mêmes performances d'apprentissage en ayant un apprentissage local pour éviter ces circuits périphériques permettrait de réduire la consommation d'énergie de plusieurs ordres de grandeur et de réduire la surface des puces neuromorphiques.

Eq-prop [Scellier and Bengio, 2016] est un algorithme dynamique qui permet de calculer localement à la synapse un gradient d'erreur similaire à *BPTT* grâce à la dynamique du réseau. C'est donc un bon candidat pour un apprentissage local physique [Kendall et al., 2020] malgré la nécessité d'un circuit externe pour garder en mémoire les gradients et les appliquer à la synapse. Pour aller plus loin, il faut un apprentissage intrinsèque, c'est-à-dire un apprentissage local où les poids synaptiques sont modifiés par le comportement des neurones connectés.

Le cerveau réalise un tel apprentissage intrinsèque en modifiant les synapses directement par les impulsions neurales. Malheureusement, les algorithmes non-supervisés et intrinsèques inspirés de l'apprentissage effectué par le cerveau comme la *STDP* n'obtiennent pas des performances de classification comparables à *BP* sur des problèmes supervisés.

Le but de la thèse était de proposer des algorithmes capables d'apprentissage supervisé intrinsèque avec une précision de classification proche de l'état de l'art avec des nanocomposants émergents pour synapses, comme des memristors, dans une optique d'apprentissage sur circuit et de réduction énergétique de l'inférence et de l'apprentissage des *ANNs/SNNs*. Pour cela, j'ai développé un algorithme basé sur *Eq-prop* pour des réseaux de neurones à impulsions où les impulsions des neurones sont utilisées pour calculer et pour modifier la valeur des poids synaptiques.

Dans le chapitre 2, nous avons dans un premier temps modifié *Eq-prop* pour les réseaux à impulsions puis nous avons proposé une méthode pour calculer et mémoriser localement aux neurones les paramètres nécessaires à la règle d'apprentissage : $\Delta w_{ij} = \dot{\rho}_i \rho_j + \dot{\rho}_j \rho_i$, en particulier la dérivée de la fréquence $\dot{\rho}$. En effet, pour le surcoût d'un bloc supplémentaire, $\bar{\dot{\rho}}$, par neurone, il est possible de calculer localement la dérivée de la fréquence du neurone, ce qui est nécessaire au calcul de la règle d'apprentissage. Ce nouvel algorithme, avec un apprentissage intrinsèque, est appelé *EqSpike*, et atteint des performances proches de l'état de l'art pour la classification d'image (96.87% de précision sur *MNIST*, contre 97.11% pour *BPTT*

avec un réseau de taille $784 \times 100 \times 10$). Cet algorithme permet un apprentissage intrinsèque grâce à l'évolution de la dynamique du réseau et la modification des poids synaptiques effectuée par les impulsions des neurones qui appliquent la valeur calculée par $\overline{\rho}$ au poids synaptique. Nous avons proposé une variante à l'algorithme ne réalisant pas les phases d'apprentissage lorsque l'erreur est trop faible, afin de réduire l'énergie consommée d'environ un facteur 2. De plus, l'estimation énergétique théorique avec une implémentation entièrement *CMOS* est de plusieurs ordres de grandeur en dessous des *GPUs*, deux ordres de grandeur pour l'entrainement ($42J$ pour 50 époques sur *MNIST*) et trois ordres de grandeur pour l'inférence ($15mJ$ pour 10000images). Aussi, l'utilisation de neurones à impulsions permet de connaître le résultat de l'inférence dès les premières impulsions des neurones de la couche de sortie, pour une perte de précision de moins de 2.4% comparé à la précision en fin de phase d'inférence, mais une réduction du temps de l'inférence d'environ un ordre de grandeur. Pour finir, nous avons montré expérimentalement le lien entre la *STDP* et *EqSpike*, déjà suggéré entre la *STDP* et *Eq-prop* [Bengio et al., 2017, Scellier and Bengio, 2016], ouvrant des voies vers des implémentations bio plausibles de *EqSpike*.

Dans le chapitre 3, une implémentation de *EqSpike* adaptée pour les memristors est proposée. Les memristors sont des résistances nanométriques programmables souvent utilisées comme synapses dans des circuits neuromorphiques dans une structure en réseau matriciel afin d'effectuer les sommes pondérées grâce à la conductance des memristors. Nous avons proposé un schéma permettant d'implémenter *EqSpike* avec ce réseau matriciel grâce à la superposition des impulsions des neurones qui a mené à un dépôt de brevet *Thales* [*FR2101311 + PCT/EP2022/053026*]. Nous avons montré que la précision de cette implémentation est équivalente à celle d'*EqSpike* pour des memristors parfaits. De plus, nous avons proposé une variante à l'algorithme avec une mise à jour aléatoire des poids synaptiques permettant de réduire d'un ordre de grandeur le nombre d'impulsions de programmation, les plus coûteuses en énergie. Combiné avec la sélection des hyperparamètres pour augmenter au maximum le taux d'apprentissage et réduire le temps de la phase d'apprentissage et le nombre d'époques, le nombre d'impulsions de programmation est réduit de 2 ordres de grandeur pour une perte de précision de classification de 3% comparé à la solution optimale. La réduction du nombre d'impulsions de programmation, grâce à leur génération stochastique, a permis d'augmenter le taux d'apprentissage d'un ordre de grandeur et ainsi d'avoir une tolérance au bruit de conductance lors de l'écriture de l'ordre de 10^{-4}. Cela correspond à une incrémentation d'environ 10Ω pour un memristor allant de $10k\Omega$ à $100k\Omega$ contre un incrément $> 1\Omega$ sans l'augmentation du taux d'apprentissage et la diminution de nombre de modifications des memristors.

En complément, dans le chapitre 3, l'impact de différentes non-idéalités des composants et entre composants a été étudié. La résilience aux variations intrinsèques des composants a été comparée à celle obtenue avec *BP*, avec différents algorithmes de descente de gradient stochastique comme référence. Nous avons observé qu'*EqSpike* est de manière générale moins robuste que *BP* aux variations à cause du nombre d'écritures plus important pour *EqSpike* qui accumule les erreurs au cours de l'exécution. Nous avons regardé la robustesse à la variation des neurones en appliquant à chaque entrée de neurone un facteur multiplicatif α aléatoire, de moyenne 1 et de déviation σ_{neuron}. La perte de précision de classification est de 17% pour $\sigma_{neuron} = 1$ mais est non significative pour $\sigma_{neuron} = 0.1$. *EqSpike* calcule implicitement (in-materio) la dérivée des fonctions d'activation, il est donc raisonnable de penser que la robustesse à ce type de variations dans les réseaux profonds sera meilleure que pour *BP* qui accumule une erreur à chaque couche. La modification de la conductance des memristors étant bruitée, nous avons regardé l'impact de ce bruit sur le réseau pour un bruit blanc avec une déviation de 10^{-7} à 1 (normalisé sur la plage de conductance possible). *EqSpike* est moins résistant que *BP* pour ce type de variation, car les poids sont plus souvent mis à jour ce qui entraîne une accumulation des erreurs plus importante au cours du temps. Mais avec un bruit inférieur à 10^{-3} les performances de cet algorithme ne changent pas significativement. Malheureusement, il est rare pour un memristor d'avoir plus de 1000 états, où un changement d'état soudain correspond donc à un bruit de 10^{-3}. Les memristors étant non linéaires, un facteur de non-linéarité symétrique allant de 0 à 2 a été appliqué. Le réseau perd environ 2% de précision pour un facteur égal à 1 et environ 6% de précision pour un facteur égal à 2. Les memristors avec le moins de non-linéarité possèdent un facteur de non-linéarité inférieur à 1 [Gao et al., 2015]. Nous avons ensuite fait varier aléatoirement le seuil d'écriture ou les pentes du memristor d'une déviation avec un écart-type de 20%. Le réseau perd environ 10% de précision pour une variation asymétrique des pentes autour de leur valeur moyenne de 10% et perd environ 18% de précision pour 20% de variation des pentes. En revanche, si les pentes d'un memristor varient symétriquement il n'y a pas de perte de précision pour une variation avec un écart-type d'au moins 100%. L'implémentation d'*EqSpike* avec des memristors est basée sur la superposition d'un signal de programmation égal au seuil

d'écriture du memristor et d'un signal proportionnel à la dérivée de la fréquence du neurone. Cette implémentation rend le réseau sensible aux variations intercomposants du seuil d'écriture des memristors. Avec des memristors *rapides* (la conductance qui change rapidement pour une faible tension à ses bornes), le réseau est incapable d'apprendre pour des variations avec un écart-type de plus de 0.01%. En revanche, si le memristor est suffisamment *lent*, alors l'algorithme peut être robuste à la variabilité du seuil sans perte significative de précision pour les memristors les plus *lents*. Notre étude sur la variabilité permet de choisir les memristors les mieux adaptés à une implémentation physique de *EqSpike*, en particulier ceux présentant une forte robustesse sur le seuil et les pentes, dans une moindre mesure linéaire, avec un bruit lors de l'écriture permettant suffisamment d'états pour résoudre le problème visé ($\leq 10^{-4}$ pour *Digits* avec nos paramètres). *EqSpike* s'adapte à des memristors qui ont des paramètres non symétriques à condition que l'asymétrie ne soit pas renforcée par la variance intercomposants. Nous n'avons pas trouvé dans la littérature un memristor qui correspondent à tous ces critères en même temps. Une des raisons est que retrouver les valeurs des paramètres souhaité dans un article est souvent difficile. Développer une méthodologie pour obtenir les paramètres importants pour *EqSpike* est donc nécessaire pour trouver les memristors adéquats. En revanche, avec l'amélioration des composants dans le futur, il est raisonnable de penser que la variabilité va continuer à diminuer pour atteindre des niveaux de variation permettant encore une plus faible perte de précision avec *EqSpike*.

Dans le chapitre 4, nous proposons une implémentation d'*EqSpike* plus proche de la biologie appelée *A-EqSpike* qui ne nécessite pas un bloc $\overline{\dot{\rho}}$ de composants CMOS pour chaque neurone afin de mesurer et calculer la dérivée de la fréquence du neurone. Il est possible d'implémenter un comportement similaire à la *STDP* utilisée pour les memristors grâce à la superposition d'impulsions de formes adéquates à ses bornes [Linares-Barranco and Serrano-Gotarredona, 2009]. En utilisant cette idée, il est possible de générer des formes d'impulsion permettant de calculer la règle d'apprentissage et de modifier le poids synaptique sans avoir à implémenter le bloc $\overline{\dot{\rho}}$. Les neurones émettent une impulsion en deux parties. La première partie est une longue queue d'impulsion qui, quand elle se superpose à d'autres queues d'impulsion, se comporte comme un accumulateur capable de calculer la dérivée des fréquences. La deuxième partie est une impulsion de programmation similaire à celle utilisée pour *EqSpike*. Nous avons proposé différentes variations de l'algorithme avec des complexités d'implémentation différentes, comme des probabilités de génération d'impulsions pour des variantes asynchrones ou des variantes synchronisées des neurones. Les deux variantes avec des neurones asynchrones montrant les meilleurs résultats, nécessitent soit une génération aléatoire des impulsions de programmation (95.35% sur *Digits*), soit une impulsion de programmation plus faible que le temps de réfraction des neurones (97.05% sur *Digits*) de plusieurs ordres de grandeur. La variante synchronisée est la variante de *A-EqSpike* qui obtient les meilleures performances avec une précision de classification de 97.1% sur *Digits* et de 96.71% sur *MNIST* contre 97.6% pour *EqSpike*.

5.2 Perspectives

Même si un schéma de circuit a été proposé dans le chapitre 3, aucune implémentation physique n'a été effectuée. Une suite logique à ces travaux serait d'implémenter une preuve de concept, sur de petits réseaux à une couche, d'*EqSpike* avec des memristors, puis d'augmenter la complexité du réseau. L'étude sur la robustesse à la variabilité des composants permet de choisir les composants les plus adaptées pour cette implémentation physique.

Dans cette thèse, nous nous sommes concentrés sur les synapses avec des memristors car leur nombre est en général bien plus important que celui des neurones. Les neurones seraient implémentés avec des circuits *CMOS*, mais les memristors, avec un circuit adapté, peuvent se comporter comme des neurones à impulsions. Il est donc possible de se servir de différents memristors pour implémenter des synapses et aussi des neurones physiques, ces derniers étant appelés *neuristors* [Pickett et al., 2013]. Cette technologie permettrait une meilleure économie d'énergie ainsi qu'une augmentation de la densité du circuit. En revanche, les neurones de *EqSpike* ont un comportement spécifique et il faudrait adapter les neuristors existants (ou adapter l'algorithme) pour faire fonctionner l'ensemble de manière optimale.

Au-delà de la preuve de concept physique nécessaire, plusieurs voies peuvent être explorées. En effet, *EqSpike* a été développé avec l'idée d'une faible consommation pour calculer des réseaux comportant des milliards de paramètres ainsi que pour s'approcher de réseaux embarquables. De plus, un rapprochement avec la biologie a été effectué et peut être encore développé.

Le plus gros réseau entraîné dans cette thèse possédait une couche cachée de 300 neurones pour s'entraîner sur le problème *MNIST* qui, même s'il s'agit d'un problème de référence pour les réseaux à impulsions, manque de complexité comparé aux problèmes les plus ardus où *BP* est performant. *EqSpike* peut être parallélisé en partie pour accélérer son temps de simulation et ainsi être capable d'entrainer des réseaux convolutifs pour des problèmes comme *CIFAR-10* ou *ImageNet* dans des délais raisonnables pour vérifier ses performances comparé à l'état de l'art.

De plus, *EqSpike*, ne fonctionne pas avec des processus algorithmiques comme l'apprentissage par paquet (*batch*, en anglais), *dropout* [Hinton et al., 2012b] ou *batch normalization* [Ioffe and Szegedy, 2015], pourtant la règle d'apprentissage calcule une descente de gradient [Ernoult et al., 2019b]. Il serait intéressant d'adapter ces techniques à l'algorithme pour permettre un apprentissage plus rapide et plus robuste. De même, une version de *Equilibrium propagation* binaire a été récemment développée [Laydevant et al., 2021]. Cette méthode nécessite un poids synaptique réel pendant l'apprentissage, mais des poids binaires pendant l'inférence, ce qui rend compliqué l'apprentissage intrinsèque tel que proposé par *EqSpike*. Toutefois, les poids binaires rendent cette méthode robuste aux variations des composants l'algorithme. Des études permettraient de rendre possible l'interface entre les deux algorithmes, et ainsi de profiter de la résistance aux bruits de conductance des réseaux binaires.

Les réseaux de neurones sont généralement sur-paramétrés pour l'entraînement [Zhou et al., 2020]. Ces poids supplémentaires peuvent être supprimé après l'apprentissage pour réduire le nombre de calculs nécessaire et ainsi réduire l'énergie nécessaire à l'exécution du réseau pour l'inférence [LeCun et al., 1990]. Certaines méthodes proposent d'*élaguer* (*prunnig*, en anglais) le réseau pendant l'entraînement afin de gagner en performance énergétique avec peu ou pas de perte de précision de classification [Gale et al., 2019, Blalock et al., 2020, Hacene et al., 2021]. L'élagage nécessite une architecture du réseau modifiable, ce qui demande une adaptation du matériel ainsi que d'*EqSpike*.

Dans cette thèse, nous ne nous sommes pas concentrés sur la façon de transmettre un signal en entrée ou comment le traiter en sortie. Nous ne nous sommes pas non plus concentrés sur l'embarquabilité de *EqSpike*. Disposer d'un circuit capable de traiter une information à la sortie du capteur peut être utile pour des raisons de vitesse d'exécution, mais aussi pour traiter l'information avant de la stocker ou de la transmettre ailleurs. En effet, les objets connectés sont de plus en plus nombreux et il est courant de transmettre beaucoup d'informations entre ces appareils embarqués et un serveur ou un autre appareil. Par exemple, les images prises par un drone ou une caméra de sécurité envoyée à un poste de surveillance consomment beaucoup de bande passante. Analyser ces images pour envoyer les images jugées utiles par un réseau ou comprimer l'image grâce à un *ANNs* pour réduire sa taille, sont des applications intéressantes pour l'industrie. Cette thèse propose des pistes pour permettre d'embarquer un apprentissage sur puce, cependant des améliorations sont encore nécessaires pour atteindre cet objectif.

Par exemple, avec l'arrivée de nouvelles données dans le système embarqué qui pourraient potentiellement être apprises, il serait nécessaire d'entraîner le réseau de neurones avec toutes les anciennes données pour éviter l'oubli catastrophique [McCloskey and Cohen, 1989]. Malheureusement, il est peu avantageux de stocker tout le jeu de données sur un système embarqué pour des raisons de mémoire et d'énergie. Pour éviter cela, des algorithmes d'apprentissage continu (*online*, en anglais) doivent être développés. Un apprentissage continu nécessite d'étiqueter les données, pour aller plus loin *EqSpike* doit être adapté à de l'apprentissage non supervisé pour disposer d'un système embarqué plus autonome.

Pour les expériences effectuées dans cette thèse, des jeux de données avec des valeurs réelles ont été utilisés, obligeant à pré-traiter les données en les convertissant en trains d'impulsions. Des capteurs qui envoient des impulsions existent et permettraient d'éviter de convertir le signal en impulsions, réduisant d'autant plus l'énergie dépensée. Malheureusement ces signaux sont épisodiques, ce qui n'est pas compatible avec des entrées constantes comme dans *EqSpike*. En effet, la dynamique du réseau sert à calculer l'erreur du réseau pendant la phase d'apprentissage et ne peut pas être exploitée comme un *RNN* pour avoir une mémoire temporelle. Pour résoudre ce problème, il serait possible d'exprimer la dynamique avec une trajectoire plutôt qu'un hamiltonien et de minimiser l'action de la trajectoire (*action functional*, en anglais) qui est une fonction lagrangienne [Scellier, 2021], ce qui transforme un *energy-based model* en un *Lagrangian-based model*. L'encodage de l'information en trains d'impulsions est aussi très coûteux en énergie. Pour être compétitif aux optimisations matérielles les plus performantes énergétiquement, il faut le minimum possible d'impulsions par neurone [Davidson and Furber, 2021]. Dans le chapitre 2, environ 98% de l'énergie était

consommée par la première couche de neurones à cause des trains d'impulsions des entrées, car nous n'avons pas optimisé l'encodage du signal d'entrée. Il serait intéressant de modifier l'encodage pour réduire significativement la consommation d'énergie. De plus, cette couche n'étant pas bidirectionnelle, il serait par exemple possible de diviser la fréquence par deux, mais d'augmenter les poids synaptiques par deux, ainsi que le taux d'apprentissage. Cela devrait normalement être équivalent, en moyenne, à l'implémentation déjà proposée. D'autres techniques d'encodage devraient aussi pouvoir être explorées.

Par exemple, il est plus avantageux pour l'énergie de tirer parti du temps dans un *SNN* et d'encoder l'information avec le délai du moment de l'émission de l'impulsion (le moment d'émission de l'impulsion encode l'information plutôt que la fréquence du neurone) [Kheradpisheh and Masquelier, 2020]. Cela réduit fortement le nombre d'impulsions, mais rend impossible la méthode de calcul de la règle d'apprentissage présentée dans cette thèse. En effet, *EqSpike* doit écrire le memristor à chaque impulsion pour être proportionnel à la fréquence du neurone. Avec une communication par délai, le nombre d'impulsions ne sera pas proportionnel à la valeur de la fonction d'activation du neurone comme demandé par *EqSpike*. Pour cet encodage, il faut donc modifier la façon de calculer et d'appliquer la fréquence ρ.

Une solution pour que la règle d'apprentissage ne dépende pas de la fréquence des neurones, mais uniquement de leur dérivée, est proposée par [Richards and Lillicrap, 2019] qui s'inspire des dendrites biologiques et ignore en plus les deux phases distinctes de l'apprentissage de *Eq-prop* pour n'en mettre en œuvre qu'une seule. De cette façon, cette méthode permettrait plus facilement d'encoder une information grâce au timing des impulsions, tout en rajoutant un lien entre la biologie et l'algorithme avec les dendrites.

Avec l'algorithme *A-EqSpike*, les synapses sont modifiées uniquement grâce aux impulsions des neurones. Cette méthode ne permet pas de filtrer les modifications comme dans *EqSpike* ce qui conduit à des modifications bruitées. Travailler sur la forme des impulsions ou trouver un équivalent au filtrage permettrait des modifications moins bruitées et donc de meilleures performances pour *A-EqSpike*.

Les algorithmes, *EqSpike* et *A-EqSpike* développés dans la thèse permettraient une économie d'énergie de plusieurs ordres de grandeur comparé aux architectures *Von Neumann* sans perte de précision comparé à l'état de l'art et d'embarquer l'apprentissage, une fois déployés sur des puces neuromorphiques avec les memristors adéquats.

Bibliographie

[Amdahl, 1967] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485.

[An, 1996] An, G. (1996). The effects of adding noise during backpropagation training on a generalization performance. *Neural computation*, 8(3) :643–674.

[Auli et al., 2013] Auli, M., Galley, M., Quirk, C., and Zweig, G. (2013). Joint Language and Translation Modeling with Recurrent Neural Networks.

[Bellec et al., 2020] Bellec, G., Scherr, F., Subramoney, A., Hajek, E., Salaj, D., Legenstein, R., and Maass, W. (2020). A solution to the learning dilemma for recurrent networks of spiking neurons. *Nature Communications*, 11(1) :3625.

[Bengio, 2009] Bengio, Y. (2009). Learning Deep Architectures for AI. *Foundations and Trends® in Machine Learning*, 2(1) :1–127.

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

[Benjamin et al., 2014] Benjamin, B. V., Gao, P., McQuinn, E., Choudhary, S., Chandrasekaran, A. R., Bussat, J.-M., Alvarez-Icaza, R., Arthur, J. V., Merolla, P. A., and Boahen, K. (2014). Neurogrid : A Mixed-Analog-Digital Multichip System for Large-Scale Neural Simulations. *Proceedings of the IEEE*, 102(5) :699–716.

[Bi and Poo, 2001] Bi, G.-q. and Poo, M.-m. (2001). Synaptic Modification by Correlated Activity : Hebb's Postulate Revisited. *Annual Review of Neuroscience*, 24(1) :139–166.

[Blalock et al., 2020] Blalock, D., Gonzalez Ortiz, J. J., Frankle, J., and Guttag, J. (2020). What is the State of Neural Network Pruning ? *Proceedings of Machine Learning and Systems*, 2 :129–146.

[Boquet et al., 2021] Boquet, G., Macias, E., Morell, A., Serrano, J., Miranda, E., and Vicario, J. L. (2021). Offline Training for Memristor-based Neural Networks. In *2020 28th European Signal Processing Conference (EUSIPCO)*, pages 1547–1551.

[Brader et al., 2007] Brader, J. M., Senn, W., and Fusi, S. (2007). Learning Real-World Stimuli in a Neural Network with Spike-Driven Synaptic Dynamics. *Neural Computation*, 19(11) :2881–2912.

[Brown et al., 2020] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language Models are Few-Shot Learners. *arXiv :2005.14165 [cs]*. arXiv : 2005.14165.

[Burr et al., 2017] Burr, G. W., Shelby, R. M., Sebastian, A., Kim, S., Kim, S., Sidler, S., Virwani, K., Ishii, M., Narayanan, P., Fumarola, A., Sanches, L. L., Boybat, I., Gallo, M. L., Moon, K., Woo, J., Hwang, H., and Leblebici, Y. (2017). Neuromorphic computing using non-volatile memory. *Advances in Physics : X*, 2(1) :89–124.

[Burr et al., 2015] Burr, G. W., Shelby, R. M., Sidler, S., di Nolfo, C., Jang, J., Boybat, I., Shenoy, R. S., Narayanan, P., Virwani, K., Giacometti, E. U., Kurdi, B. N., and Hwang, H. (2015). 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.

[Chang et al., 2018] Chang, C.-C., Chen, P.-C., Chou, T., Wang, I.-T., Hudec, B., Chang, C.-C., Tsai, C.-M., Chang, T.-S., and Hou, T.-H. (2018). Mitigating Asymmetric Nonlinear Weight Update Effects in

Hardware Neural Network based on Analog Resistive Synapse. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 8(1) :116–124.

[Chanthbouala et al., 2012] Chanthbouala, A., Garcia, V., Cherifi, R. O., Bouzehouane, K., Fusil, S., Moya, X., Xavier, S., Yamada, H., Deranlot, C., Mathur, N. D., Bibes, M., Barthélémy, A., and Grollier, J. (2012). A ferroelectric memristor. *Nature Materials*, 11(10) :860–864.

[Chen et al., 2016] Chen, W., Fang, R., Balaban, M. B., Yu, W., Gonzalez-Velo, Y., Barnaby, H. J., and Kozicki, M. N. (2016). A CMOS-compatible electronic synapse device based on Cu/SiO $_2$$$ /W programmable metallization cells. *Nanotechnology*, 27(25) :255202.

[Cho et al., 2014] Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *arXiv :1406.1078 [cs, stat]*.

[Comsa et al., 2020] Comsa, I. M., Potempa, K., Versari, L., Fischbacher, T., Gesmundo, A., and Alakuijala, J. (2020). Temporal Coding in Spiking Neural Networks with Alpha Synaptic Function : Learning with Backpropagation. arXiv :1907.13223 [cs, q-bio].

[Davidson and Furber, 2021] Davidson, S. and Furber, S. B. (2021). Comparison of Artificial and Spiking Neural Networks on Digital Hardware. *Frontiers in Neuroscience*, 15.

[Davies et al., 2018] Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S. H., Dimou, G., Joshi, P., Imam, N., Jain, S., Liao, Y., Lin, C.-K., Lines, A., Liu, R., Mathaikutty, D., McCoy, S., Paul, A., Tse, J., Venkataramanan, G., Weng, Y.-H., Wild, A., Yang, Y., and Wang, H. (2018). Loihi : A Neuromorphic Manycore Processor with On-Chip Learning. *IEEE Micro*, 38(1) :82–99.

[Detorakis et al., 2018] Detorakis, G., Sheik, S., Augustine, C., Paul, S., Pedroni, B. U., Dutt, N., Krichmar, J., Cauwenberghs, G., and Neftci, E. (2018). Neural and Synaptic Array Transceiver : A Brain-Inspired Computing Framework for Embedded Learning. *Frontiers in Neuroscience*, 0.

[Dillavou et al., 2022] Dillavou, S., Stern, M., Liu, A. J., and Durian, D. J. (2022). Demonstration of Decentralized, Physics-Driven Learning. *arXiv :2108.00275 [cond-mat]*. arXiv : 2108.00275.

[Duchi et al., 2011] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7).

[Ernoult et al., 2019a] Ernoult, M., Grollier, J., and Querlioz, D. (2019a). Using memristors for robust local learning of hardware restricted Boltzmann machines. *Scientific reports*, 9(1) :1–15.

[Ernoult et al., 2019b] Ernoult, M., Grollier, J., Querlioz, D., Bengio, Y., and Scellier, B. (2019b). Updates of Equilibrium Prop Match Gradients of Backprop Through Time in an RNN with Static Input. *arXiv :1905.13633 [cs, stat]*.

[Ernoult et al., 2020] Ernoult, M., Grollier, J., Querlioz, D., Bengio, Y., and Scellier, B. (2020). Equilibrium Propagation with Continual Weight Updates. *arXiv :2005.04168 [cs, stat]*.

[Eryilmaz et al., 2013] Eryilmaz, S. B., Kuzum, D., Jeyasingh, R. G. D., Kim, S., BrightSky, M., Lam, C., and Wong, H.-S. P. (2013). Experimental demonstration of array-level learning with phase change synaptic devices. In *2013 IEEE International Electron Devices Meeting*, pages 25.5.1–25.5.4, Washington, DC, USA. IEEE.

[Falez et al., 2019] Falez, P., Tirilly, P., Bilasco, I. M., Devienne, P., and Boulet, P. (2019). Unsupervised visual feature learning with spike-timing-dependent plasticity : How far are we from traditional feature learning approaches ? *Pattern Recognition*, 93 :418–429.

[Fouda et al., 2019a] Fouda, M. E., Kurdahi, F., Eltawil, A., and Neftci, E. (2019a). Spiking Neural Networks for Inference and Learning : A Memristor-based Design Perspective. *arXiv :1909.01771 [cs]*.

[Fouda et al., 2019b] Fouda, M. E., Neftci, E., Eltawil, A., and Kurdahi, F. (2019b). Effect of Asymmetric Nonlinearity Dynamics in RRAMs on Spiking Neural Network Performance. In *2019 53rd Asilomar Conference on Signals, Systems, and Computers*, pages 495–499.

[Frenkel et al., 2021] Frenkel, C., Bol, D., and Indiveri, G. (2021). Bottom-Up and Top-Down Neural Processing Systems Design : Neuromorphic Intelligence as the Convergence of Natural and Artificial Intelligence. *arXiv :2106.01288 [cs]*.

[Frenkel et al., 2019a] Frenkel, C., Lefebvre, M., Legat, J.-D., and Bol, D. (2019a). A 0.086-mm^2 12.7-pJ/SOP 64k-Synapse 256-Neuron Online-Learning Digital Spiking Neuromorphic Processor in 28-nm CMOS. *IEEE Transactions on Biomedical Circuits and Systems*, 13(1) :145–158.

[Frenkel et al., 2019b] Frenkel, C., Legat, J.-D., and Bol, D. (2019b). MorphIC : A 65-nm 738k-Synapse/mm^2 Quad-Core Binary-Weight Digital Neuromorphic Processor With Stochastic Spike-Driven Online Learning. *IEEE Transactions on Biomedical Circuits and Systems*, 13(5) :999–1010. Conference Name : IEEE Transactions on Biomedical Circuits and Systems.

[Gale et al., 2019] Gale, T., Elsen, E., and Hooker, S. (2019). The State of Sparsity in Deep Neural Networks. *arXiv :1902.09574 [cs, stat]*. arXiv : 1902.09574.

[Galluppi et al., 2014] Galluppi, F., Temple, S., Plana, L. A., and others (2014). The SpiNNaker Project. *Proceedings of the IEEE*, 102 :652–665.

[Gao et al., 2015] Gao, L., Wang, I. T., Chen, P. Y., Vrudhula, S., Seo, J.-s., Cao, Y., Hou, T. H., and Yu, S. (2015). Fully parallel write/read in resistive synaptic array for accelerating on-chip learning. *Nanotechnology*, 26(45).

[Gerstner, 2008] Gerstner, W. (2008). Spike-response model. *Scholarpedia*, 3(12) :1343.

[Gerstner and Brette, 2009] Gerstner, W. and Brette, R. (2009). Adaptive exponential integrate-and-fire model. *Scholarpedia*, 4(6) :8427.

[Gerstner et al., 2014] Gerstner, W., Kistler, W. M., Naud, R., and Paninski, L. (2014). *Neuronal dynamics : From single neurons to networks and models of cognition*. Cambridge University Press.

[Gerstner et al., 2018] Gerstner, W., Lehmann, M., Liakoni, V., Corneil, D., and Brea, J. (2018). Eligibility Traces and Plasticity on Behavioral Time Scales : Experimental Support of NeoHebbian Three-Factor Learning Rules. *Frontiers in Neural Circuits*, 0.

[Glorot and Bengio, 2010] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Teh, Y. W. and Titterington, M., editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy. PMLR.

[Hacene et al., 2021] Hacene, G. B., Lassance, C., Gripon, V., Courbariaux, M., and Bengio, Y. (2021). Attention Based Pruning for Shift Networks. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 4054–4061. ISSN : 1051-4651.

[Hebb, 1949] Hebb, D. (1949). The organization of behavior ; a neuropsychological theory. *John Wiley & Sons, Inc.*

[Hinton et al., 2012a] Hinton, G., Srivastava, N., and Swersky, K. (2012a). Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14(8) :2.

[Hinton, 2002] Hinton, G. E. (2002). Training Products of Experts by Minimizing Contrastive Divergence. *Neural Computation*, 14(8) :1771–1800. Conference Name : Neural Computation.

[Hinton et al., 2012b] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012b). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv :1207.0580*.

[Hirtzlin et al., 2019] Hirtzlin, T., Bocquet, M., Ernoult, M., Klein, J. O., Nowak, E., Vianello, E., Portal, J. M., and Querlioz, D. (2019). Hybrid Analog-Digital Learning with Differential RRAM Synapses. In *2019 IEEE International Electron Devices Meeting (IEDM)*, pages 22.6.1–22.6.4.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8) :1735–1780.

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

[Hornik et al., 1989] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5) :359–366.

[Ielmini and Ambrogio, 2020] Ielmini, D. and Ambrogio, S. (2020). Emerging neuromorphic devices. *Nanotechnology*, 31(9) :092001.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization : Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR.

[Ishii et al., 2019] Ishii, M., Kim, S., Lewis, S., Okazaki, A., Okazawa, J., Ito, M., Rasch, M., Kim, W., Nomura, A., Shin, U., Hosokawa, K., BrightSky, M., and Haensch, W. (2019). On-Chip Trainable 1.4M 6T2R PCM Synaptic Array with 1.6K Stochastic LIF Neurons for Spiking RBM. In *2019 IEEE International Electron Devices Meeting (IEDM)*, pages 14.2.1–14.2.4.

[Jaderberg et al., 2017] Jaderberg, M., Czarnecki, W. M., Osindero, S., Vinyals, O., Graves, A., Silver, D., and Kavukcuoglu, K. (2017). Decoupled Neural Interfaces using Synthetic Gradients. *arXiv :1608.05343 [cs]*.

[Jerry et al., 2017] Jerry, M., Chen, P.-Y., Zhang, J., Sharma, P., Ni, K., Yu, S., and Datta, S. (2017). Ferroelectric FET analog synapse for acceleration of deep neural network training. *2017 IEEE International Electron Devices Meeting (IEDM)*.

[Jo et al., 2010] Jo, S. H., Chang, T., Ebong, I., Bhadviya, B. B., Mazumder, P., and Lu, W. (2010). Nanoscale Memristor Device as Synapse in Neuromorphic Systems. *Nano Letters*, 10(4) :1297–1301.

[Joseph and Nagarajan, 2020] Joseph, V. and Nagarajan, C. (2020). MADONNA : A Framework for Energy Measurements and Assistance in Designing Low Power Deep Neural Networks. page 7.

[Jouppi et al., 2017] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. (2017). In-Datacenter Performance Analysis of a Tensor Processing UnitTM. *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12.

[Jung et al., 2022] Jung, S., Lee, H., Myung, S., Kim, H., Yoon, S. K., Kwon, S.-W., Ju, Y., Kim, M., Yi, W., Han, S., Kwon, B., Seo, B., Lee, K., Koh, G.-H., Lee, K., Song, Y., Choi, C., Ham, D., and Kim, S. J. (2022). A crossbar array of magnetoresistive memory devices for in-memory computing. *Nature*, 601(7892) :211–216.

[Kaiser et al., 2020] Kaiser, J., Mostafa, H., and Neftci, E. (2020). Synaptic Plasticity Dynamics for Deep Continuous Local Learning (DECOLLE). *Frontiers in Neuroscience*, 0.

[Kataeva et al., 2015] Kataeva, I., Merrikh-Bayat, F., Zamanidoost, E., and Strukov, D. (2015). Efficient training algorithms for neural networks based on memristive crossbar circuits. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8.

[Kendall et al., 2020] Kendall, J., Pantone, R., Manickavasagam, K., Bengio, Y., and Scellier, B. (2020). Training End-to-End Analog Neural Networks with Equilibrium Propagation. *arXiv :2006.01981 [cs]*.

[Kheradpisheh and Masquelier, 2020] Kheradpisheh, S. R. and Masquelier, T. (2020). Temporal Backpropagation for Spiking Neural Networks with One Spike per Neuron. *International Journal of Neural Systems*, 30(06) :2050027.

[Kingma and Ba, 2017] Kingma, D. P. and Ba, J. (2017). Adam : A Method for Stochastic Optimization. *arXiv :1412.6980 [cs]*.

[Kocić et al., 2019] Kocić, J., Jovičić, N., and Drndarević, V. (2019). An End-to-End Deep Neural Network for Autonomous Driving Designed for Embedded Automotive Platforms. *Sensors (Basel, Switzerland)*, 19(9) :2064.

[Krizhevsky et al., 2009] Krizhevsky, A., Hinton, G., and others (2009). Learning multiple layers of features from tiny images. Publisher : Citeseer.

[Laborieux et al., 2021] Laborieux, A., Ernoult, M., Scellier, B., Bengio, Y., Grollier, J., and Querlioz, D. (2021). Scaling Equilibrium Propagation to Deep ConvNets by Drastically Reducing Its Gradient Estimator Bias. *Frontiers in Neuroscience*, 0.

[Laydevant et al., 2021] Laydevant, J., Ernoult, M., Querlioz, D., and Grollier, J. (2021). Training Dynamical Binary Neural Networks With Equilibrium Propagation. pages 4640–4649.

[LeCun et al., 2015] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553) :436–444.

[LeCun et al., 2010] LeCun, Y., Cortes, C., and Burges, C. (2010). MNIST handwritten digit database. *ATT Labs [Online]. Available : http ://yann.lecun.com/exdb/mnist*, 2.

[LeCun et al., 1990] LeCun, Y., Denker, J. S., and Solla, S. A. (1990). Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605.

[Lee et al., 2016] Lee, J. H., Delbruck, T., and Pfeiffer, M. (2016). Training Deep Spiking Neural Networks Using Backpropagation. *Frontiers in Neuroscience*, 0.

[Lequeux et al., 2016] Lequeux, S., Sampaio, J., Cros, V., Yakushiji, K., Fukushima, A., Matsumoto, R., Kubota, H., Yuasa, S., and Grollier, J. (2016). A magnetic synapse : multilevel spin-torque memristor with perpendicular anisotropy. *Scientific Reports*, 6(1) :31510. Number : 1 Publisher : Nature Publishing Group.

[Leroux et al., 2021] Leroux, N., Marković, D., Martin, E., Petrisor, T., Querlioz, D., Mizrahi, A., and Grollier, J. (2021). Radio-Frequency Multiply-And-Accumulate Operations with Spintronic Synapses. *Physical Review Applied*, 15(3) :034067. arXiv : 2011.07885.

[Li et al., 2018] Li, C., Belkin, D., Li, Y., Yan, P., Hu, M., Ge, N., Jiang, H., Montgomery, E., Lin, P., Wang, Z., Song, W., Strachan, J. P., Barnell, M., Wu, Q., Williams, R. S., Yang, J. J., and Xia, Q. (2018). Efficient and self-adaptive in-situ learning in multilayer memristor neural networks. *Nature Communications*, 9(1) :2385.

[Li et al., 2015] Li, S., Liu, X., Nandi, S. K., Venkatachalam, D. K., and Elliman, R. G. (2015). High-endurance megahertz electrical self-oscillation in Ti/NbOx bilayer structures. *Applied Physics Letters*, 106(21) :212902.

[Lillicrap et al., 2016] Lillicrap, T. P., Cownden, D., Tweed, D. B., and Akerman, C. J. (2016). Random synaptic feedback weights support error backpropagation for deep learning. *Nature Communications*, 7(1) :13276.

[Lim et al., 2019] Lim, S., Bae, J.-H., Eum, J.-H., Lee, S., Kim, C.-H., Kwon, D., Park, B.-G., and Lee, J.-H. (2019). Adaptive learning rule for hardware-based deep neural networks using electronic synapse devices. *Neural Computing and Applications*, 31(11) :8101–8116.

[Linares-Barranco and Serrano-Gotarredona, 2009] Linares-Barranco, B. and Serrano-Gotarredona, T. (2009). Memristance can explain Spike-Time-Dependent-Plasticity in Neural Synapses. *Nature Precedings*, pages 1–1.

[Liu et al., 2019] Liu, T., Wen, W., Jiang, L., Wang, Y., Yang, C., and Quan, G. (2019). A fault-tolerant neural network architecture. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE.

[Maass, 1997] Maass, W. (1997). Networks of spiking neurons : The third generation of neural network models. *Neural Networks*, 10(9) :1659–1671.

[Maass et al., 1991] Maass, W., Schnitger, G., and Sontag, E. D. (1991). On the computational power of sigmoid versus boolean threshold circuits. In *FOCS*, pages 767–776.

[Marković et al., 2020] Marković, D., Mizrahi, A., Querlioz, D., and Grollier, J. (2020). Physics for neuromorphic computing. *Nature Reviews Physics*, 2(9) :499–510.

[Martin et al., 2021] Martin, E., Ernoult, M., Laydevant, J., Li, S., Querlioz, D., Petrisor, T., and Grollier, J. (2021). EqSpike : Spike-driven Equilibrium Propagation for Neuromorphic Implementations. *arXiv :2010.07859 [cs]*.

[McCloskey and Cohen, 1989] McCloskey, M. and Cohen, N. J. (1989). Catastrophic Interference in Connectionist Networks : The Sequential Learning Problem. In Bower, G. H., editor, *Psychology of Learning and Motivation*, volume 24, pages 109–165. Academic Press.

[Mead and Ismail, 2012] Mead, C. and Ismail, M. (2012). *Analog VLSI implementation of neural systems*, volume 80. Springer Science & Business Media.

[Merolla et al., 2014] Merolla, P. A., Arthur, J. V., Alvarez-Icaza, R., Cassidy, A. S., Sawada, J., Akopyan, F., Jackson, B. L., Imam, N., Guo, C., Nakamura, Y., Brezzo, B., Vo, I., Esser, S. K., Appuswamy, R., Taba, B., Amir, A., Flickner, M. D., Risk, W. P., Manohar, R., and Modha, D. S. (2014). A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197) :668–673.

[Mesnard et al., 2016] Mesnard, T., Gerstner, W., and Brea, J. (2016). Towards deep learning with spiking neurons in energy based models with contrastive Hebbian plasticity. *arXiv :1612.03214 [cs, q-bio]*.

[Milo et al., 2020] Milo, V., Malavena, G., Monzio Compagnoni, C., and Ielmini, D. (2020). Memristive and CMOS Devices for Neuromorphic Computing. *Materials*, 13(1) :166.

[Milo et al., 2016] Milo, V., Pedretti, G., Carboni, R., Calderoni, A., Ramaswamy, N., Ambrogio, S., and Ielmini, D. (2016). Demonstration of hybrid CMOS/RRAM neural networks with spike time/rate-dependent plasticity. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pages 16.8.1–16.8.4.

[Moore, 1998] Moore, G. E. (1998). Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1) :82–85.

[Moradi et al., 2018] Moradi, S., Qiao, N., Stefanini, F., and Indiveri, G. (2018). A scalable multi-core architecture with heterogeneous memory structures for Dynamic Neuromorphic Asynchronous Processors (DYNAPs). *IEEE Transactions on Biomedical Circuits and Systems*, 12(1) :106–122.

[Moskovitz et al., 2018] Moskovitz, T. H., Litwin-Kumar, A., and Abbott, L. F. (2018). Feedback alignment in deep convolutional networks. *arXiv :1812.06488 [cs, stat]*.

[Mostafa, 2018] Mostafa, H. (2018). Supervised Learning Based on Temporal Coding in Spiking Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, 29(7) :3227–3235.

[Mostafa et al., 2018] Mostafa, H., Ramesh, V., and Cauwenberghs, G. (2018). Deep Supervised Learning Using Local Errors. *Frontiers in Neuroscience*, 0.

[Movellan, 1991] Movellan, J. R. (1991). Contrastive Hebbian Learning in the Continuous Hopfield Model. In Touretzky, D. S., Elman, J. L., Sejnowski, T. J., and Hinton, G. E., editors, *Connectionist Models*, pages 10–17. Morgan Kaufmann.

[Nandakumar S. et al., 2017] Nandakumar S., R., Le Gallo, M., Boybat, I., Rajendran, B., Sebastian, A., and Eleftheriou, E. (2017). Mixed-precision training of deep neural networks using computational memory. *arXiv e-prints*, 1712 :arXiv :1712.01192.

[Navarro et al., 2020] Navarro, M. A., Salari, A., Lin, J. L., Cowan, L. M., Penington, N. J., Milescu, M., and Milescu, L. S. (2020). Sodium channels implement a molecular leaky integrator that detects action potentials and regulates neuronal firing. *eLife*, 9 :e54940.

[Neftci et al., 2017] Neftci, E. O., Augustine, C., Paul, S., and Detorakis, G. (2017). Event-Driven Random Back-Propagation : Enabling Neuromorphic Deep Learning Machines. *Frontiers in Neuroscience*, 0.

[Nøkland, 2016] Nøkland, A. (2016). Direct Feedback Alignment Provides Learning in Deep Neural Networks. *arXiv :1609.01596 [cs, stat]*.

[Nøkland and Eidnes, 2019] Nøkland, A. and Eidnes, L. H. (2019). Training Neural Networks with Local Error Signals. *arXiv :1901.06656 [cs, stat]*.

[O'Connor and Welling, 2016] O'Connor, P. and Welling, M. (2016). Deep Spiking Networks. *arXiv :1602.08323 [cs]*.

[O'Connor et al., 2019] O'Connor, P., Gavves, E., and Welling, M. (2019). Training a Spiking Neural Network with Equilibrium Propagation. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1516–1523.

[Park et al., 2016] Park, J., Kwak, M., Moon, K., Woo, J., Lee, D., and Hwang, H. (2016). TiOx-Based RRAM Synapse With 64-Levels of Conductance and Symmetric Conductance Change by Adopting a Hybrid Pulse Scheme for Neuromorphic Computing. *IEEE Electron Device Letters*, 37(12) :1559–1562.

[Park et al., 2020] Park, J., Lee, J., and Jeon, D. (2020). A 65-nm Neuromorphic Image Classification Processor With Energy-Efficient Training Through Direct Spike-Only Feedback. *IEEE Journal of Solid-State Circuits*, 55(1) :108–119.

[Parmar and Suri, 2018] Parmar, V. and Suri, M. (2018). Design Exploration of Hybrid CMOS-OxRAM Deep Generative Architectures. *arXiv :1801.02003 [cs]*.

[Payvand et al., 2020] Payvand, M., Fouda, M. E., Kurdahi, F., Eltawil, A., and Neftci, E. O. (2020). Error-triggered Three-Factor Learning Dynamics for Crossbar Arrays. In *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 218–222.

[Pfeiffer and Pfeil, 2018] Pfeiffer, M. and Pfeil, T. (2018). Deep Learning With Spiking Neurons : Opportunities and Challenges. *Frontiers in Neuroscience*, 0.

[Pickett et al., 2013] Pickett, M. D., Medeiros-Ribeiro, G., and Williams, R. S. (2013). A scalable neuristor built with Mott memristors. *Nature Materials*, 12(2) :114–117. Number : 2 Publisher : Nature Publishing Group.

[Polyak and Juditsky, 1992] Polyak, B. T. and Juditsky, A. B. (1992). Acceleration of Stochastic Approximation by Averaging. *SIAM Journal on Control and Optimization*, 30(4) :838–855.

[Prezioso et al., 2018] Prezioso, M., Mahmoodi, M. R., Bayat, F. M., Nili, H., Kim, H., Vincent, A., and Strukov, D. B. (2018). Spike-timing-dependent plasticity learning of coincidence detection with passively integrated memristive circuits. *Nature Communications*, 9(1) :5311.

[Prezioso et al., 2015] Prezioso, M., Merrikh-Bayat, F., Hoskins, B. D., Adam, G. C., Likharev, K. K., and Strukov, D. B. (2015). Training and operation of an integrated neuromorphic network based on metal-oxide memristors. *Nature*, 521(7550) :61–64.

[Qiao et al., 2015] Qiao, N., Mostafa, H., Corradi, F., Osswald, M., Stefanini, F., Sumislawska, D., and Indiveri, G. (2015). A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128K synapses. *Frontiers in Neuroscience*, 0.

[Qin et al., 2020] Qin, Y.-F., Kuang, R., Huang, X.-D., Li, Y., Chen, J., and Miao, X.-S. (2020). Design of High Robustness BNN Inference Accelerator Based on Binary Memristors. *IEEE Transactions on Electron Devices*, 67(8) :3435–3441.

[Querlioz et al., 2013] Querlioz, D., Bichler, O., Dollfus, P., and Gamrat, C. (2013). Immunity to device variations in a spiking neural network with memristive nanodevices. *IEEE Transactions on Nanotechnology*, 12(3) :288–295.

[Querlioz et al., 2011] Querlioz, D., Dollfus, P., Bichler, O., and Gamrat, C. (2011). Learning with memristive devices : How should we model their behavior ? In *2011 IEEE/ACM International Symposium on Nanoscale Architectures*, pages 150–156.

[Raoux et al., 2008] Raoux, S., Burr, G. W., Breitwisch, M. J., Rettner, C. T., Chen, Y.-C., Shelby, R. M., Salinga, M., Krebs, D., Chen, S.-H., Lung, H.-L., and Lam, C. H. (2008). Phase-change random access memory : A scalable technology. *IBM Journal of Research and Development*, 52(4.5) :465–479.

[Richards and Lillicrap, 2019] Richards, B. A. and Lillicrap, T. P. (2019). Dendritic solutions to the credit assignment problem. *Current Opinion in Neurobiology*, 54 :28–36.

[Robbins and Monro, 1951] Robbins, H. and Monro, S. (1951). A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3) :400–407.

[Rosenblatt, 1958] Rosenblatt, F. (1958). The perceptron : a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6 :386–408.

[Ruder, 2016] Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv :1609.04747*.

[Rueckauer et al., 2017] Rueckauer, B., Lungu, I.-A., Hu, Y., Pfeiffer, M., and Liu, S.-C. (2017). Conversion of Continuous-Valued Deep Networks to Efficient Event-Driven Networks for Image Classification. *Frontiers in Neuroscience*, 11.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088) :533–536.

[Russakovsky et al., 2015] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3) :211–252.

[Sakemi et al., 2020] Sakemi, Y., Morino, K., Morie, T., and Aihara, K. (2020). A Supervised Learning Algorithm for Multilayer Spiking Neural Networks Based on Temporal Coding Toward Energy-Efficient VLSI Processor Design. arXiv :2001.05348 [cs, stat].

[Scellier, 2021] Scellier, B. (2021). A deep learning theory for neural networks grounded in physics. *arXiv :2103.09985 [cs]*. arXiv : 2103.09985.

[Scellier and Bengio, 2016] Scellier, B. and Bengio, Y. (2016). Equilibrium Propagation : Bridging the Gap Between Energy-Based Models and Backpropagation. *arXiv :1602.05179 [cs]*.

[Scellier et al., 2018] Scellier, B., Goyal, A., Binas, J., Mesnard, T., and Bengio, Y. (2018). Generalization of Equilibrium Propagation to Vector Field Dynamics. *arXiv :1808.04873 [cs, stat]*.

[Schemmel et al., 2010] Schemmel, J., Brüderle, D., Grübl, A., Hock, M., Meier, K., and Millner, S. (2010). A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1947–1950.

[Sharad et al., 2012] Sharad, M., Augustine, C., Panagopoulos, G., and Roy, K. (2012). Spin-Based Neuron Model With Domain-Wall Magnets as Synapse. *IEEE Transactions on Nanotechnology*, 11.

[Silver et al., 2016] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587) :484–489. Number : 7587 Publisher : Nature Publishing Group.

[Simonyan and Zisserman, 2014] Simonyan, K. and Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv :1409.1556*.

[Song et al., 2018] Song, Y. J., Lee, J. H., Han, S. H., Shin, H. C., Lee, K. H., Suh, K., Jeong, D. E., Koh, G. H., Oh, S. C., Park, J. H., Park, S. O., Bae, B. J., Kwon, O. I., Hwang, K. H., Seo, B., Lee, Y., Hwang, S. H., Lee, D. S., Ji, Y., Park, K., Jeong, G. T., Hong, H. S., Lee, K. P., Kang, H. K., and Jung, E. S. (2018). Demonstration of Highly Manufacturable STT-MRAM Embedded in 28nm Logic. In *2018 IEEE International Electron Devices Meeting (IEDM)*, pages 18.2.1–18.2.4, San Francisco, CA. IEEE.

[Strubell et al., 2019] Strubell, E., Ganesh, A., and McCallum, A. (2019). Energy and Policy Considerations for Deep Learning in NLP. *arXiv :1906.02243 [cs]*. arXiv : 1906.02243.

[Strukov et al., 2008] Strukov, D. B., Snider, G. S., Stewart, D. R., and Williams, R. S. (2008). The missing memristor found. *Nature*, 453(7191) :80–83.

[Suri et al., 2015] Suri, M., Parmar, V., Kumar, A., Querlioz, D., and Alibart, F. (2015). Neuromorphic hybrid RRAM-CMOS RBM architecture. In *2015 15th Non-Volatile Memory Technology Symposium (NVMTS)*, pages 1–6.

[Sze et al., 2017] Sze, V., Chen, Y.-H., Emer, J., Suleiman, A., and Zhang, Z. (2017). Hardware for machine learning : Challenges and opportunities. In *2017 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–8. IEEE.

[Szegedy et al., 2015] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2015). Rethinking the Inception Architecture for Computer Vision. *arXiv :1512.00567 [cs]*.

[Tavanaei et al., 2019] Tavanaei, A., Ghodrati, M., Kheradpisheh, S. R., Masquelier, T., and Maida, A. S. (2019). Deep Learning in Spiking Neural Networks. *Neural Networks*, 111 :47–63. arXiv : 1804.08150.

[Thorpe and Imbert, 1989] Thorpe, S. J. and Imbert, M. (1989). Biological constraints on connectionist modelling. *Connectionism in perspective*, pages 63–92.

[Truong et al., 2014] Truong, S. N., Ham, S.-J., and Min, K.-S. (2014). Neuromorphic crossbar circuit with nanoscale filamentary-switching binary memristors for speech recognition. *Nanoscale research letters*, 9(1) :1–9.

[Wan et al., 2020] Wan, W., Kubendran, R., Eryilmaz, S. B., Zhang, W., Liao, Y., Wu, D., Deiss, S., Gao, B., Raina, P., Joshi, S., Wu, H., Cauwenberghs, G., and Wong, H.-S. P. (2020). 33.1 A 74 TMACS/W CMOS-RRAM Neurosynaptic Core with Dynamically Reconfigurable Dataflow and In-situ Transposable Weights for Probabilistic Graphical Models. In *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 498–500.

[Werbos, 1990] Werbos, P. (1990). Backpropagation through time : what it does and how to do it. *Proceedings of the IEEE*, 78(10) :1550–1560.

[Weston et al., 2008] Weston, J., Ratle, F., and Collobert, R. (2008). Deep Learning via Semi-Supervised Embedding. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 1168–1175, New York, NY, USA. Association for Computing Machinery.

[Woo et al., 2017] Woo, J., Padovani, A., Moon, K., Kwak, M., Larcher, L., and Hwang, H. (2017). Linking conductive filament properties and evolution to synaptic behavior of RRAM devices for neuromorphic applications. *IEEE Electron Device Letters*, 38(9) :1220–1223.

[Wu et al., 2018] Wu, W., Wu, H., Gao, B., Yao, P., Zhang, X., Peng, X., Yu, S., and Qian, H. (2018). A methodology to improve linearity of analog RRAM for neuromorphic computing. In *2018 IEEE Symposium on VLSI Technology*, pages 103–104. IEEE.

[Xi et al., 2021] Xi, Y., Gao, B., Tang, J., Chen, A., Chang, M.-F., Hu, X. S., Spiegel, J. V. D., Qian, H., and Wu, H. (2021). In-memory Learning with Analog Resistive Switching Memory : A Review and Perspective. *Proceedings of the IEEE*, 109(1) :14–42.

[Yao et al., 2020] Yao, P., Wu, H., Gao, B., Tang, J., Zhang, Q., Zhang, W., Yang, J. J., and Qian, H. (2020). Fully hardware-implemented memristor convolutional neural network. *Nature*, 577(7792) :641–646.

[Zenke and Ganguli, 2018] Zenke, F. and Ganguli, S. (2018). SuperSpike : Supervised Learning in Multilayer Spiking Neural Networks. *Neural Computation*, 30(6) :1514–1541.

[Zhang et al., 2020a] Zhang, M., Wang, J., Amornpaisannon, B., Zhang, Z., Miriyala, V. P. K., Belatreche, A., Qu, H., Wu, J., Chua, Y., Carlson, T. E., and Li, H. (2020a). Rectified Linear Postsynaptic Potential Function for Backpropagation in Deep Spiking Neural Networks. arXiv :2003.11837 [cs].

[Zhang et al., 2020b] Zhang, W., Gao, B., Tang, J., Yao, P., Yu, S., Chang, M.-F., Yoo, H.-J., Qian, H., and Wu, H. (2020b). Neuro-inspired computing chips. *Nature Electronics*, 3(7) :371–382.

[Zhang et al., 2021] Zhang, X., Cai, W., Wang, M., Pan, B., Cao, K., Guo, M., Zhang, T., Cheng, H., Li, S., Zhu, D., Wang, L., Shi, F., Du, J., and Zhao, W. (2021). Spin-Torque Memristors Based on Perpendicular Magnetic Tunnel Junctions for Neuromorphic Computing. *Advanced Science*, 8(10) :2004645. _eprint : https ://onlinelibrary.wiley.com/doi/pdf/10.1002/advs.202004645.

[Zhang et al., 2016] Zhang, Y., Lee, K., and Lee, H. (2016). Augmenting supervised neural networks with unsupervised objectives for large-scale image classification. In *International conference on machine learning*, pages 612–621. PMLR.

[Zhou et al., 2014] Zhou, B., Lapedriza, A., Xiao, J., Torralba, A., and Oliva, A. (2014). Learning Deep Features for Scene Recognition using Places Database. In *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc.

[Zhou et al., 2020] Zhou, H., Lan, J., Liu, R., and Yosinski, J. (2020). Deconstructing Lottery Tickets : Zeros, Signs, and the Supermask. *arXiv :1905.01067 [cs, stat]*.

[Zoppo et al., 2020] Zoppo, G., Marrone, F., and Corinto, F. (2020). Equilibrium Propagation for Memristor-Based Recurrent Neural Networks. *Frontiers in Neuroscience*, 0.