NeuroTorch: Une librairie Python dédiée à l'apprentissage automatique dans le domaine des neurosciences

Mémoire

Jérémie Gince

Maîtrise en informatique - avec mémoire
Maître ès sciences (M. Sc.)

Québec, Canada

NeuroTorch : Une librairie Python dédiée à l'apprentissage automatique dans le domaine des neurosciences

Mémoire

Jérémie Gince

Sous la direction de:

Simon Hardy, directeur de recherche
Patrick Desrosiers, co-directeur de recherche

Résumé

L'apprentissage automatique a considérablement progressé dans le domaine de la recherche en neurosciences, mais son application pose des défis en raison des différences entre les principes biologiques du cerveau et les méthodes traditionnelles d'apprentissage automatique. Dans ce contexte, le projet présenté propose NeuroTorch, un pipeline convivial d'apprentissage automatique spécialement conçu pour les neuroscientifiques, afin de relever ces défis. Les objectifs clés de ce projet sont de fournir une librairie d'apprentissage profond adaptée aux neurosciences computationnelles, d'implémenter l'algorithme *eligibility trace forward propagation* (e-prop) pour sa plausibilité biologique, de comparer les réseaux de neurones continus et à impulsions en termes de résilience, et d'intégrer un pipeline d'apprentissage par renforcement.

Le projet se divise en plusieurs parties. Tout d'abord, la théorie des dynamiques neuronales, des algorithmes d'optimisation et des fonctions de transformation d'espaces sera développée. Ensuite, l'attention sera portée sur la conception du pipeline NeuroTorch, incluant l'implémentation de l'algorithme e-prop. Les résultats de la prédiction de séries temporelles d'activité neuronale chez le poisson-zèbre seront présentés, ainsi que des observations sur la résilience à l'ablation des réseaux obtenus. Enfin, une section sera consacrée à l'exploration du pipeline d'apprentissage par renforcement de NeuroTorch et à la validation de son architecture dans l'environnement LunarLander de Gym.

En résumé, les modèles à impulsions de NeuroTorch ont atteint des précisions de 96,37%, 85,58% et 74,16% respectivement sur les ensembles de validation MNIST, Fashion-MNIST et Heidelberg. De plus, les dynamiques *leaky-integrate-and-fire with explicit synaptic current - low pass filter* (SpyLIF-LPF) et Wilson-Cowan ont été entraînées avec succès à l'aide de l'algorithme e-prop sur des données neuronales expérimentales du ventral habenula du poisson-zèbre, obtenant respectivement des valeurs de pVar de 0,97 et 0,96. Les résultats concernant la résilience indiquent que l'application de la loi de Dale améliore la robustesse des modèles en termes d'ablation hiérarchique. Enfin, grâce au pipeline d'apprentissage par renforcement de NeuroTorch, différents types d'agents inspirés des neurosciences ont atteint le critère de réussite dans l'environnement LunarLander de Gym. Ces résultats soulignent la pertinence et l'efficacité de NeuroTorch pour les applications en neurosciences computationnelles.

Abstract

Machine learning has made significant advancements in neuroscience research, but its application presents challenges due to the differences between the biological principles of the brain and traditional machine learning methods. In this context, the presented project proposes NeuroTorch, a comprehensive machine learning pipeline specifically designed for neuroscientists to address these challenges. The key objectives of this project are to provide a deep learning library tailored to computational neuroscience, implement the eligibility trace forward propagation (e-prop) algorithm for biological plausibility, compare continuous and spiking neural networks in terms of resilience, and integrate a reinforcement learning pipeline.

The project is divided into several parts. Firstly, the theory of neural dynamics, optimization algorithms, and space transformation functions will be developed. Next focus will be on the design of the NeuroTorch pipeline, including the implementation of the e-prop algorithm. Results of predicting a time series of neuronal activity in zebrafish will be presented, along with observations on the resilience to network ablations obtained. Finally, a section will be dedicated to exploring the NeuroTorch reinforcement learning pipeline and validating its architecture in the LunarLander environment of Gym.

In summary, NeuroTorch spiking models achieved accuracies of 96.37%, 85.58%, and 74.16% on the MNIST, Fashion-MNIST, and Heidelberg validation sets, respectively. Furthermore, the leaky-integrate-and-fire with explicit synaptic current - low pass filter (SpyLIF-LPF) and Wilson-Cowan dynamics were successfully trained using the e-prop algorithm on experimental neuronal data from the ventral habenula of zebrafish, achieving pVar values of 0.97 and 0.96, respectively. Results regarding resilience indicate that the application of the Dale law improves the robustness of models in terms of hierarchical ablation. Lastly, through the NeuroTorch reinforcement learning pipeline, different types of neuroscience-inspired agents successfully met the success criterion in the Gym's LunarLander environment. These results highlight the relevance and effectiveness of NeuroTorch for applications in computational neuroscience.

Table des matières

Liste des figures

Liste des tableaux

"To infinity, and beyond!"
- Buzz Lightyear

Remerciements

Je tiens à exprimer ma sincère gratitude envers mes directeurs, Simon Hardy et Patrick Desrosiers, de m'avoir donné l'opportunité de poursuivre une maîtrise en neurosciences computationnelles. Leur encadrement, leurs conseils judicieux et leur soutien indéfectible tout au long de ce parcours académique ont été inestimables pour mon développement professionnel et personnel. Je leur suis particulièrement reconnaissant pour la grande liberté qu'ils m'ont accordée dans la conception de mon projet, ce qui m'a permis d'aspirer à des objectifs ambitieux et de viser les étoiles.

Je tiens également à reconnaître la précieuse collaboration du laboratoire du professeur Paul de Koninck pour avoir généreusement partagé leurs données, ce qui a grandement enrichi mes travaux de recherche.

De plus, je suis profondément reconnaissant envers le centre de recherche UNIQUE pour la bourse d'excellence qui m'a été attribuée en début de parcours. Cette aide financière a été déterminante pour me permettre de me concentrer pleinement sur mes études et mes recherches.

Finalement, j'adresse humblement mes remerciements à toutes les personnes qui ont contribué, directement ou indirectement, à cette aventure académique. Leur encouragement, leur guidage et leur confiance en mes capacités ont été essentiels pour mener à bien ce projet.

Avant-propos

Dans le chapitre 2, le manuscrit intitulé «NeuroTorch : A Python library for neuroscience-oriented machine learning» sera présenté. Celui-ci sera soumis prochainement au journal *Scientific Reports - Nature*. Ce document concrétise l'implémentation de NeuroTorch ainsi que son application sur des données neuronales de poissons-zèbres provenant du laboratoire de Paul De Koninck au Centre de recherche CERVO de Québec. Je, Jérémie Gince, suis l'auteur principal de cet article, et les coauteurs sont les suivants :

Anthony Drouin, stagiaire au Département de physique, de génie physique et d'optique de l'Université Laval ;

Antoine Légaré, étudiant au doctorat affilié au Centre de recherche CERVO de Québec ainsi qu'au centre interdisciplinaire en modélisation mathématique de l'Université Laval ;

Paul De Koninck, professeur au Département de biochimie, de microbiologie et de bio-informatique de l'Université Laval et affilié au Centre de recherche CERVO de Québec ;

Patrick Desrosiers, professeur associé au Département de physique, de génie physique et d'optique de l'Université Laval, affilié au Centre de recherche CERVO de Québec ainsi qu'au centre interdisciplinaire en modélisation mathématique de l'Université Laval ;

Simon V. Hardy, professeur au Département d'informatique et de génie logiciel de l'Université Laval et au Département de biochimie, de microbiologie et de bio-informatique de l'Université Laval ainsi qu'affilié au Centre de recherche CERVO de Québec.

Dans cet article, J.G. a réalisé le développement et le codage de la bibliothèque, ainsi que les expériences numériques. A.D. a apporté son assistance à J.G. dans le travail computationnel. A.L. a été responsable de la collecte des données d'activité neuronale, tandis que P.D.K. a supervisé son travail expérimental. J.G. a rédigé la première version du manuscrit qui a ensuite été bonifié par P.D. et S.V.H. J.G. a rédigé la documentation de NeuroTorch et A.D. a contribué.

Introduction

La neuroscience est une discipline scientifique qui explore le système nerveux et son rôle dans le fonctionnement des organismes vivants. Elle vise à comprendre les mécanismes complexes sous-jacents aux processus mentaux et aux comportements observés dans le règne biologique. Dans cette quête, l'analyse des séries temporelles est essentielle due à sa proximité avec les données expérimentales et à sa richesse d'information. Par exemple, l'imagerie cérébrale [1], comme l'imagerie par résonance magnétique fonctionnelle (IRMf) [2], permet de cartographier l'activité cérébrale durant différentes tâches. Pour comprendre les interactions entre les différentes régions du système nerveux, les chercheurs étudient également les connexions neuronales. Elles permettent de tracer et de cartographier les voies de communication entre les neurones [3]. Un des objectifs majeurs de la neuroscience est de construire le connectome [4], une cartographie complète des connexions neuronales du cerveau. Connaître le connectome devrait contribuer à élucider plusieurs questions fondamentales en neurosciences [5]. Cependant, il est encore impossible d'extraire le connectome structurel et de dresser la liste des synapses sans recourir à des méthodes invasives [6]. Malgré les progrès réalisés, la complexité et la densité des connexions neuronales rendent cette tâche extrêmement difficile. De nouvelles approches non invasives tel que l'imagerie du cerveau du poisson-zèbre in vivo [7] offre une opportunité unique pour observer et analyser les circuits neuronaux dans un organisme vivant. L'intégration de l'apprentissage machine dans l'analyse de ces données complexes représente une avancée prometteuse pour la compréhension approfondie du cerveau et du comportement biologique, en contribuant à élucider le connectome [8, 9] et en ouvrant de nouvelles perspectives dans l'étude du fonctionnement neuronal.

En intégrant des principes biologiques tels que la plasticité synaptique et les propriétés électrophysiologiques des neurones dans la conception des réseaux neuronaux artificiels, on vise à créer des modèles qui reproduisent fidèlement le fonctionnement du cerveau vivant [10, 11]. Cette approche améliore notre compréhension des mécanismes cérébraux et renforce l'interprétabilité des modèles. L'adoption de dynamiques biologiquement plausibles permet une meilleure correspondance avec les processus neuronaux réels, permettant ainsi l'étude des processus biologiques par le biais de simulations numériques. En offrant une simulation réaliste, cette approche rend les résultats plus interprétables et plus utiles en recherche. Ce dialogue entre l'apprentissage profond et la neuroscience offre donc des opportunités de progrès mutuels

et favorise le développement de modèles plus performants et interprétables, ouvrant ainsi de nouvelles perspectives de recherche [12, 13].

Un des principes fondamentaux rapprochant l'apprentissage profond et la biologie vient des réseaux de neurones à impulsions (SNN) ou parfois nommés réseaux de neurones à décharges. Ce type de réseau permet une dynamique plus réaliste au niveau biologique que les réseaux artificiels classiques (ANN) [14, 15] dû à son activité en impulsions plutôt que continue. Également, elle offre dans quelques situations un pipeline asynchrone et permet de réduire l'énergie utilisée par son activité neuronale éparse. D'autre part, l'entraînement de ces réseaux par rétropropagation est remplacé par des processus plus plausibles au niveau biologique comme STDP [16] et e-prop [17] utilisant de l'information locale plutôt que globale comme observée dans le monde vivant [18, 19]. De par ces nouvelles pratiques, le pont entre l'apprentissage machine et la neuroscience se réduit considérablement au niveau théorique. Plus concrètement, les réseaux non récurrents à activation ReLu [20] utilisés en apprentissage machine sont remplacés par des réseaux récurrents avec des dynamiques provenant des neurosciences. En effet, les dynamiques *leaky integrate and fire* (LIF) et *adaptative leaky integrate and fire* (ALIF) [14, 15, 21] sont utilisées pour modéliser de l'activité neuronale relativement réaliste. L'émergence des processeurs neuromorphiques, tels que le processeur Loihi d'Intel [22] ou SpiNNaker [23], ont aussi joué un rôle clé dans la croissance de la popularité des réseaux de neurones à impulsions [24]. Ces processeurs permettent d'exploiter de manière optimale les caractéristiques spécifiques des SNN, notamment leur nature éparse et asynchrone. Grâce à ces avancées, l'utilisation efficace des réseaux à impulsions en termes de temps et d'énergie est désormais possible, ce qui a contribué à leur essor dans le domaine de la recherche en neurosciences computationnelles.

Au niveau pratique, la recherche en apprentissage profond est en général effectuée avec les modules Python [25] d'autodifférentiation tels que PyTorch [26] et TensorFlow [27]. Toutefois, ces modules ne proposent pas intrinsèquement de modèles ou de techniques d'entraînement dérivés des neurosciences. Pour combler cette lacune, des librairies spécialisées telles que Norse [28], SpyTorch [29], et snnTorch [30] ont été développées pour mettre en œuvre des architectures de réseaux neuronaux à impulsions, facilitant ainsi la recherche en informatique neuromorphique. Ces librairies fournissent un ensemble de couches d'apprentissage à impulsions qui peuvent être utilisées avec PyTorch, démontrant leur fonctionnalité dans des tâches telles que la classification de jeux de données populaires tels que MNIST [31, 32] et Heidelberg [33]. L'article [34] présente une liste plus exhaustive de librairies sur les réseaux neuronaux à impulsions (SNN). Cependant, bien que ces librairies proposent différentes règles d'apprentissage, elles n'incluent pas actuellement la règle d'apprentissage e-prop. Pour disposer d'un pipeline d'apprentissage complet, la récente librairie PyTorch Lightning [35] offre un pipeline d'entraînement complet et intégré compatible avec PyTorch, mais ne comprend pas d'algorithmes d'optimisation spécifiques aux neurosciences. De même, la bibliothèque Poutyne [36] propose également une

alternative pour la gestion de l'entraînement des modèles PyTorch, mais elle ne propose pas non plus d'algorithmes d'optimisation spécifiques aux neurosciences. Ces deux bibliothèques se concentrent principalement sur la simplification du processus d'entraînement et la fourniture d'outils supplémentaires tels que des métriques prédéfinies et des *callbacks* d'entraînement.

De son côté, l'apprentissage par renforcement (RL) révolutionne la résolution des problèmes considérés impossibles telle que la construction d'un agent ayant la capacité de jouer à un jeu vidéo utilisant seulement des images de son environnement afin de prendre ses décisions et accomplir une tâche complexe [37]. Ce type d'apprentissage inspiré des neurosciences et de la psychologie devient de plus en plus important en apprentissage profond grâce à ses performances dépassant maintenant l'humain pour certaines tâches comme les jeux Atari, Go et le Poker. En neurosciences, le RL prend de plus en plus d'importance [38, 39] grâce aux parallèles entre l'apprentissage du cerveau biologique et celui artificiel [40]. Pour l'optimisation de réseaux en apprentissage par renforcement, la librairie *stable baselines 3* [41] est certainement une des plus populaires. Elle fournit un pipeline d'apprentissage complet avec un grand ensemble d'algorithmes d'optimisation hautement utilisé en RL. Toutefois, elle ne fournit pas de modèle ou d'algorithme d'optimisation plausible biologiquement.

Les problématiques rencontrées dans le domaine des neurosciences computationnelles, telles que l'absence d'algorithmes d'apprentissage biologiquement plausibles spécifiquement adaptés à ce domaine et le manque de librairies proposant un pipeline d'apprentissage dédié aux neurosciences computationnelles, ont motivé l'objectif principal de ce projet. Ainsi, l'élaboration d'une librairie d'apprentissage profond avec PyTorch vise à combler ces lacunes en fournissant des algorithmes d'apprentissage biologiquement plausibles conçus spécifiquement pour les neurosciences computationnelles. Cette librairie offrira également un pipeline d'apprentissage complet et intégré, centré sur les besoins et les spécificités de la recherche en neurosciences computationnelles.

En plus de cet objectif principal, le projet comprend également des objectifs secondaires importants. Tout d'abord, il prévoit l'implémentation générale de l'algorithme *eligibility trace forward propagation* (e-prop) proposé par Bellec et collab. [17]. Cette technique d'apprentissage est réputée pour sa plausibilité biologique et son adaptation aux caractéristiques des réseaux neuronaux biologiques, ce qui en fait un choix particulièrement pertinent pour la librairie en développement.

Un deuxième objectif secondaire est la comparaison de la résilience entre les réseaux de neurones continus et les réseaux de neurones à impulsions. Cette comparaison permettra de mieux comprendre les avantages et les limites de chaque approche en termes de défaillance des neurones ou d'ablation hiérarchique. Ces connaissances contribueront à orienter les choix de dynamiques utilisées dans les applications en neurosciences computationnelles.

Enfin, un autre objectif consiste à intégrer un pipeline d'apprentissage par renforcement à la

librairie proposée. L'apprentissage par renforcement est une technique puissante pour l'entraînement des réseaux neuronaux, qui peut être particulièrement pertinent pour la modélisation des mécanismes de prise de décision et de contrôle moteur observés dans le cerveau. L'intégration de cette fonctionnalité permettra aux chercheurs en neurosciences computationnelles d'explorer et d'exploiter les principes de l'apprentissage par renforcement dans leurs études.

Pour accomplir ces objectifs, la librairie NeuroTorch [42] a été développée lors de ce projet. Elle s'appuie sur les dernières avancées en matière de librairies de réseaux de neurones à impulsions, et offre des fonctionnalités supplémentaires pour l'analyse des connectomes, la régularisation basée sur ces métriques, ainsi que des méthodes d'optimisation telles que la rétropropagation à travers le temps (BPTT), l'*eligibility trace forward propagation* (e-prop) [17], le *Recursive-least-square* (RLS) [3, 43-45], et le *Proximal Policy Optimization* (PPO) [46]. En outre, la librairie intègre une dynamique populaire en neurosciences, connue sous le nom de Wilson-Cowan [15, 47-50], ainsi que d'autres outils destinés à la recherche en informatique neuromorphique. Ainsi, NeuroTorch offre un large éventail de fonctionnalités spécifiquement conçues pour soutenir la recherche en neurosciences computationnelles et en informatique neuromorphique.

Dans le premier chapitre, la théorie sur les dynamiques neuronales, les algorithmes d'optimisations ainsi que sur des fonctions de transformation d'espaces seront élaborés. Afin de supporter le premier chapitre, une petite expérience sur la réduction dimensionnelle éparse est présentée en annexe A. Le second chapitre présentera l'article sur la conception du pipeline de NeuroTorch, l'implémentation de l'algorithme e-prop et des résultats sur la prédiction de séries temporelles d'activité neuronale d'un poisson-zèbre, ainsi qu'une observation sur la résilience des réseaux résultants. Finalement, le dernier chapitre portera sur le pipeline d'apprentissage par renforcement de NeuroTorch et des résultats de validation de l'architecture dans l'environnement LunarLander de Gym [51] offrant des environnements standardisés et des mesures de performance objectives.

L'intégralité de la librairie NeuroTorch, ainsi que ses codes sources, est disponible sur github.com/NeuroTorch. Ce répertoire contient toutes les informations nécessaires pour installer NeuroTorch et accéder aux données de référence. De plus, des tutoriels complets sont disponibles sur Google Colab pour une expérience pratique avec la librairie.

Chapitre 1

Théorie

Ce chapitre se déploiera en plusieurs étapes. Tout d'abord, la structure fondamentale d'un modèle d'apprentissage profond sera examinée. Ensuite, les dynamiques sous-tendant ces modèles ainsi que les algorithmes d'apprentissage responsables de leur optimisation seront explorés. Par la suite, une méthode de régularisation du réseau, permettant de contrôler un paramètre biologique spécifique, sera introduite. Enfin, le chapitre se conclura en détaillant un ensemble de méthodes conçues pour transformer les données de manière à les rendre pertinentes pour l'entraînement des modèles. Il est important de noter que les dynamiques telles que le modèle *Leaky-integrate-and-fire*, *Leaky-integrate with explicit synaptic current*, *Leaky-integrate-and-fire with explicit synaptic current* et *Wilson-Cowan*, ainsi que l'algorithme d'apprentissage e-prop, seront abordées dans le chapitre 2. Cela est dû à leur pertinence pour l'article présenté dans le même chapitre. Afin d'éviter toute redondance, ces algorithmes ne seront pas détaillés dans le présent chapitre.

Ces méthodes sont disponibles dans la librairie NeuroTorch [42] et peuvent être instanciées en tant que couches d'apprentissage, de *callbacks* ou autre. D'autres méthodes non présentées dans ce chapitre sont également disponibles. Veuillez vous référer à la librairie pour une liste plus complète des modules actuellement implémentés.

1.1 Structure des modèles

1.1.1 Le modèle

L'apprentissage profond, une sous-catégorie de l'intelligence artificielle, est le processus par lequel un système informatique apprend à partir des données disponibles pour effectuer des tâches spécifiques sans être explicitement programmé pour les accomplir. Ce système informatique est un modèle formé de neurones artificiels qui sont organisés de façon à être en mesure d'accomplir une tâche spécifique. Cette tâche peut être traduite comme une fonction prenant en entrée une image, un texte ou des évènements passés et résultant en une sortie sous une

forme quelconque. Le théorème d'approximation universelle [52] stipule qu'il est possible d'approximer toute fonction continue à l'aide d'un modèle adéquat. Ainsi, il est possible de créer un modèle qui est capable d'approximer cette fonction associée à une tâche spécifique. Il est donc maintenant nécessaire d'exprimer la structure générale d'un tel modèle dans l'objectif d'être en mesure de construire des modèles adaptés aux problèmes que l'on tente de résoudre.

Pour débuter, il est important de considérer l'unité de base d'un modèle, également désigné sous le nom de réseau de neurones artificiels, qui est la couche. Une couche est équipée de paramètres, les poids entre les neurones et leur biais (θ), ainsi que d'une dynamique spécifique ($h_\theta(\cdot)$) qui régit les interactions entre les neurones artificiels et la manière dont l'information est traitée à travers le réseau. Ici, le terme "dynamique" est employé dans un sens général, englobant tout calcul numérique pouvant être perçu comme un processus dynamique, où des opérations mathématiques sont réalisées de manière séquentielle. Ainsi, les données sont fournies à l'instant zéro, tandis que la sortie est obtenue un pas de temps ultérieur suite à l'application de la fonction $h_\theta(\cdot)$. Une couche peut donc être définie par la relation $\hat{y} = h_\theta(x)$, où x et $\hat{y}$ correspondent respectivement aux entrées et aux sorties de la couche en question. Pour mieux visualiser cette notion, la figure 1.1.1 illustre une couche qui prend en entrée des données, les traite à travers une fonction prédéterminée, et fournit le résultat en sortie.

De manière plus concrète, l'une des dynamiques les plus couramment utilisées est la couche linéaire, accompagnée d'une fonction d'activation $\mathrm{ReLu}(x) = \max(0, x)$ et de paramètres $\theta = \left[W^{\text{in}}, b\right]$. Cette configuration conduit à la dynamique $h_\theta(x) = \mathrm{ReLu}(W^{\text{in}} \cdot x + b)$, où le symbole $\cdot$ représente le produit intérieur. En effet, cette couche simple, souvent appelée *fully connected*, est largement préférée dans le domaine de l'apprentissage profond en raison de sa simplicité et de ses performances éprouvées à maintes reprises dans divers scénarios [20].

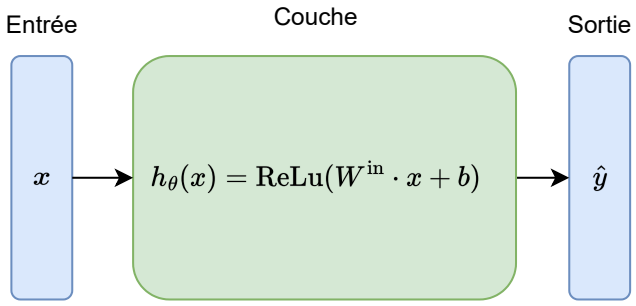

FIGURE 1.1.1 – Structure d'un modèle constitué d'une seule couche de neurones artificielles doté d'une dynamique linéaire à activation ReLu.

Ensuite, il devient possible d'assembler ces couches pour former un modèle plus complexe. Dans cette configuration, les sorties des couches précédentes deviennent les entrées des couches subséquentes, comme clairement démontré dans la figure 1.1.2. L'ajout de ces couches dans le

modèle permet d'accroître sa capacité d'expression. En d'autres termes, la capacité d'expression d'un modèle représente son potentiel à approximer des fonctions complexes. Il est bien établi dans le domaine de l'apprentissage profond que plus un modèle est profond, c'est-à-dire possédant plusieurs couches, plus sa capacité d'expression est élevée [53].

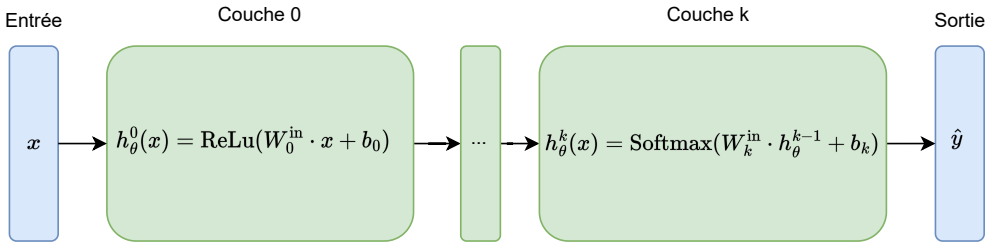

FIGURE 1.1.2 – Structure d'un modèle constitué de plusieurs couches où l'entrée d'une couche est la sortie de la précédente. Les dynamiques utilisée dans ces couches peuvent être différentes les unes des autres comme montré avec deux types d'activations différentes.

La structure multicouche présente un intérêt pour augmenter la complexité du modèle utilisé. Cependant, son adaptation aux données qui évoluent dans le temps, aussi appelées séries temporelles, est limitée, à l'exception des modèles de type *transformer* [54], qui ne seront pas abordés dans ce travail. Afin de traiter efficacement ce type de données, une architecture de modèle récurrent (RNN) est nécessaire, comme illustrée à la figure 1.1.3. L'avantage d'un RNN réside dans sa capacité à modéliser des dépendances temporelles et séquentielles, en permettant aux informations passées de se propager vers les sorties actuelles. De plus, il convient de noter que les RNN respectent le théorème d'approximation universelle. En effet, [55, 56] stipulent que tout système dynamique continu, et donc toute fonction, peut être approximé par un RNN.

Le fonctionnement de cette architecture est essentiellement similaire à celui de la structure présentée précédemment. Cependant, dans le cas spécifique d'un RNN, il existe une notion de temporalité. Il peut recevoir une série temporelle en entrée, notée par exemple $x = \left[x_0, \quad x_1, \quad \ldots, \quad x_{T-1}\right]$, et produire une série temporelle en sortie en intégrant séquentiellement les informations de x. Ainsi, il génère $\hat{y} = \left[h_\theta^0(x_0, h^{-1}), h_\theta^1(x_1, h^0), \ldots, h_\theta^{T-1}(x_{T-1}, h^{T-2})\right]$, où les h_θ^t représentent l'application du modèle complet aux données d'entrée à chaque instant de temps. Dans cette expression, chaque h_θ^t dépend des réponses précédentes, ce qui permet de capturer les dépendances temporelles dans la prédiction.

Les réseaux récurrents peuvent également être utilisés pour effectuer des prédictions à plus long terme. Par exemple, ils peuvent recevoir en entrée une série temporelle, l'intégrer, puis continuer à prédire des données pour les τ instants de temps suivants. Ainsi, la sortie du modèle devient $\hat{y} = \left[h_\theta^0(x_0), \ldots, h_\theta^{T-1}(x_{T-1}, h^{T-2}), \ldots, h_\theta^{\tau-1}(h^{\tau-2})\right]$. Cette capacité à modéliser des dépendances temporelles complexes en fait un outil puissant pour traiter et analyser les

données séquentielles, telles que les séries temporelles.

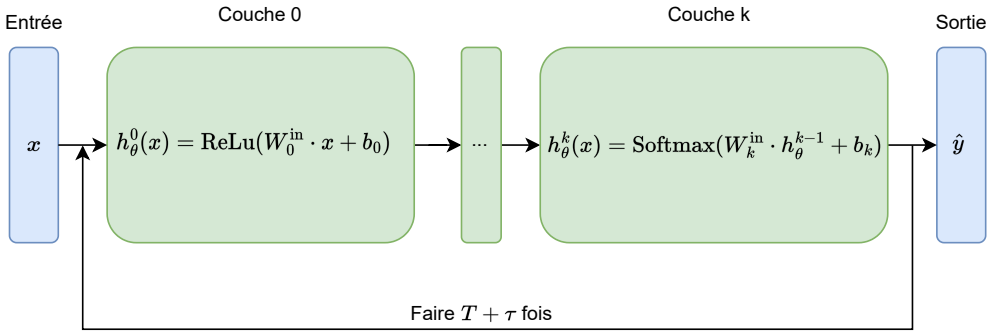

FIGURE 1.1.3 – Structure d'un modèle constitué de plusieurs couches où l'entrée d'une couche est la sortie de la précédente. Ce type de modèle étant récurrent, l'entrée du modèle est la sortie de celui-ci au pas de temps précédent. Dans cette illustration, T représente le nombre de pas de temps de x et τ représente le nombre de pas de prédiction.

Les problèmes en général s'accompagnent souvent d'un ensemble de contraintes, comme des limitations sur l'espace des entrées ou des sorties, des restrictions de complexité du modèle, voire même des contraintes sur la structure du modèle lui-même. Par exemple, dans certains cas, l'objectif n'est pas seulement de créer un modèle capable de prédire l'activité neuronale d'un poisson-zèbre, mais de produire un modèle biologiquement réaliste. Cela signifie que la structure et la dynamique du réseau doivent présenter des similitudes avec le fonctionnement du monde animal. Dans ce contexte, il serait préférable d'opter pour une structure récurrente et des dynamiques reconnues comme biologiquement réalistes, telles que *leaky integrate and fire*, *adaptative leaky integrate and fire* ou *Wilson-Cowan*, plutôt que d'utiliser des dynamiques plus simples comme celles d'une couche linéaire.

De plus, dans certains problèmes, les contraintes s'étendent à l'espace de sortie. Par exemple, dans une tâche de classification, un modèle doit prédire une distribution de probabilité sous forme d'un vecteur avec une entrée pour chaque classe possible. Dans ce cas, la dernière dynamique doit être équipée d'une activation appropriée, comme la fonction $\text{Softmax}(x_i) = \frac{\exp\{x_i\}}{\sum_j \exp\{x_j\}}$ où i et j sont des indices représentant des neurones. Dans le contexte de la prédiction d'images, le modèle doit renvoyer une matrice de taille correspondant à celle de l'image souhaitée. C'est pour répondre à ce type de contraintes que certaines dynamiques ou structures sont plus avantageuses que d'autres pour un problème donné.

1.1.2 L'optimisation du modèle

Une fois que la structure du modèle est solidement établie, ses paramètres initiaux sont généralement choisis de manière aléatoire. Cette approche découle de la difficulté à déterminer les valeurs optimales pour permettre au modèle de remplir efficacement sa tâche. Par conséquent,

il est impératif d'entreprendre une recherche des valeurs de paramètres visant à optimiser une fonction objectif. Cette fonction, souvent désignée par le terme de "fonction de perte" et notée $\mathcal{L}$, représente le critère à minimiser au cours du processus d'optimisation. Dans la plupart des cas, cette fonction de perte est différentiable, ce qui permet d'appliquer une descente de gradient pour mettre à jour les paramètres du modèle. L'algorithme d'apprentissage se charge ainsi de trouver les paramètres optimaux qui minimisent la fonction de perte, dans ce que l'on appelle le processus d'entraînement du modèle.

Le processus d'entraînement se déroule généralement en plusieurs étapes : le modèle effectue une série de prédictions sur les données d'entraînement qui lui sont dédiées, puis évalue ces prédictions à l'aide de la fonction de perte. Ensuite, les paramètres du modèle sont mis à jour en conséquence. Ces étapes se répètent jusqu'à ce qu'une convergence soit atteinte, c'est-à-dire jusqu'à ce que les performances du modèle deviennent suffisamment élevées pour la tâche en question. L'ensemble de cet algorithme est illustré dans la figure 1.1.4.

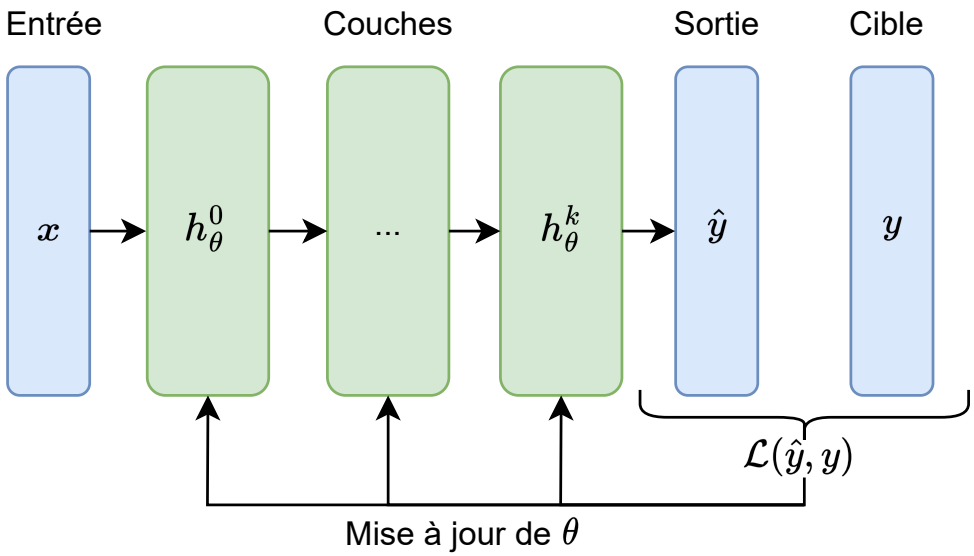

FIGURE 1.1.4 – Structure d'un modèle multicouche optimisé par un algorithme non défini utilisant la sortie attendue (la cible) y et la fonction de perte $\mathcal{L}(\cdot)$ pour mettre à jour les paramètres θ du modèle.

Une large variété d'algorithmes d'apprentissage est disponible, tout comme pour les dynamiques. Certains affichent une complexité de calcul élevée, d'autres se rapprochent davantage des mécanismes biologiques, certains sont rapides tandis que d'autres requièrent une quantité substantielle de mémoire. Le choix de l'algorithme d'apprentissage dépend du problème à résoudre ainsi que des contraintes spécifiques au contexte. En fonction de la situation donnée, certains algorithmes d'apprentissage se révèlent plus appropriés que d'autres.

Un exemple courant d'algorithme d'apprentissage est la rétropropagation [20], largement uti-

lisée pour l'entraînement des réseaux de neurones. Dans cette méthode, les dérivées partielles des sorties de chaque couche par rapport à ses entrées sont calculées. Ces dérivées en chaîne permettent de mesurer l'influence de chaque couche sur la fonction de perte. L'objectif est de minimiser la fonction de perte en ajustant les poids du réseau dans le sens contraire de son gradient par rapport aux paramètres du modèle. En d'autres termes, les poids θ sont mis à jour selon $\theta \mapsto \theta + \Delta\theta$, où $\Delta\theta$ est proportionnel au gradient négatif de la fonction de perte par rapport aux poids : $\Delta\theta \propto -\nabla_\theta \mathcal{L}(\hat{y}, y)$.

D'un autre côté, pour illustrer un autre type d'algorithme, prenons l'exemple d'une optimisation par algorithme génétique [57] basé sur la théorie de l'évolution de Darwin [58]. Dans ce cas, au lieu de se baser sur des gradients, l'algorithme génétique explore un espace de solutions en simulant le processus de sélection naturelle. Il maintient une population de solutions candidates, les croise et les mute pour créer de nouvelles générations. Ces nouvelles solutions sont ensuite évaluées selon une fonction de perte, et le processus se répète pour rechercher des solutions de plus en plus performantes en fonction des critères définis. Ce type d'approche peut être utilisé dans des contextes où les gradients ne sont pas facilement calculables.

1.1.3 Régularisation et transformations

L'algorithme d'apprentissage effectue une recherche de paramètres sans contraintes intrinsèques pour réduire l'espace de recherche. Cependant, dans certains cas, l'objectif n'est pas seulement d'optimiser les performances du modèle, mais aussi de restreindre l'espace de recherche des paramètres. Comme mentionné précédemment, le modèle peut être considéré comme une fonction, ou même une famille de fonctions, où chaque fonction diffère par son ensemble de paramètres. Pour certains problèmes, la fonction optimale n'est pas nécéssairement la fonction la plus performante. Par exemple, supposons qu'un modèle présente une excellente performance, mais une très grande variance dans une situation où être consistant est un atout. Dans un tel contexte, le modèle plus consistant sera préféré et non celui plus performant. Dans un tel cas, un terme proportionnel à la variance du modèle $\mathcal{L}_{\text{regul}}$ est ajouté dans la fonction de perte afin d'obtenir $\mathcal{L}_{\text{tot}} \propto \mathcal{L}_{\text{pred}} + \mathcal{L}_{\text{regul}}$.

Afin de finaliser la structure du modèle, il est également envisageable d'incorporer des transformations à l'entrée et à la sortie du modèle, comme illustré à la figure 1.1.5. Ces transformations peuvent servir à diverses fins. Elles peuvent, par exemple, accélérer l'apprentissage ou l'inférence (c'est-à-dire la prédiction) en déplaçant les données vers un GPU. De plus, elles ont la capacité de transférer les données soit dans l'espace de la dynamique d'entrée, soit dans l'espace propre au problème. Prenons l'exemple où la dynamique d'entrée requiert des données normalisées entre 0 et 1 ou une série temporelle créée à partir d'une image. Dans ce cas, une transformation appropriée peut être appliquée en entrée du modèle.

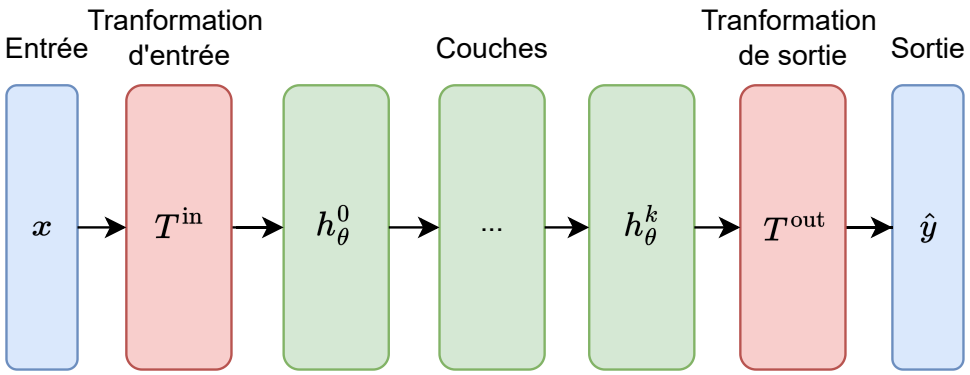

FIGURE 1.1.5 – Structure d'un modèle général. L'ajout de transformations en entrée T^{in} et sortie T^{out} du modèle servant à encoder et décoder les données d'entrées x et de sortie h_θ^k respectivement.

1.1.4 L'apprentissage par renforcement

L'apprentissage par renforcement (RL) est une autre méthode utilisée afin d'optimiser un réseau de neurones artificiels. Les méthodes classiques utilisent un ou des critères clairs et différentiables ainsi qu'un ensemble de données statique (défini préalablement) afin d'optimiser les paramètres d'un modèle. En RL, le critère de performance est nommé récompense (*rewards*) et est en général créé subjectivement de façon à être proportionnel à la qualité de décision du modèle à optimiser. De plus, on ne parle plus simplement d'un modèle, mais bien d'un agent. Un agent est un objet comportant un ou plusieurs modèle(s) et évolue dans un environnement où il doit prendre des décisions dans l'objectif de maximiser ses récompenses. La figure 1.1.6 illustre le flux d'information dans un contexte de RL. L'ensemble de variables de cette figure $(s_t, a_t, r_{t+1}, s_{t+1})$ étant respectivement l'état courant (ou l'observation courante), l'action courante, la récompense suite à l'action et le nouvel état (ou la prochaine observation) désigne une expérience au temps t. L'ensemble des expériences de $t = 0$ à $t = T$ d'un agent désigne une trajectoire de cet agent. Le critère de fin de trajectoire est une caractéristique propre à chaque environnement.

Un exemple populaire en RL afin de bien comprendre ce flux d'information est celui d'un maître apprenant à son chien à s'asseoir lorsque demandé. Dans cet exemple, le chien est l'agent qui évolue dans son environnement, soit la maison du maître. Les actions possibles du canin sont en réalité infinies, mais seront restreintes à japper, rouler ou s'asseoir pour simplifier le tout. Les récompenses sont les morceaux de foie séché dont il raffole. L'état du chien se trouve à être la position qu'il occupe un instant t et les observations du chien sont ce qu'il est capable de voir ou d'entendre. En RL, l'état et les observations de l'agent sont souvent confondus en une seule variable et font en général référence à l'entrée du modèle que l'agent utilisera pour prendre une décision (une action). Dans l'exemple du chien, celui-ci reçoit donc

une observation au temps $t = 0$ qui sera de s'asseoir, il effectuera ensuite une action ce qui changera son état afin d'obtenir une nouvelle observation et une récompense en retour. Lors de son apprentissage, le meilleur ami de l'homme effectuera une série d'actions et recevra une série de récompenses qui lui permettra d'associer les observations aux bonnes actions.

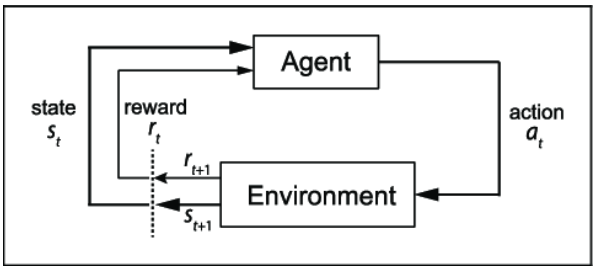

FIGURE 1.1.6 – Flux d'information dans un contexte d'apprentissage par renforcement. Figure tirée de [59].

Un autre concept important en apprentissage par renforcement est l'horizon. En reprenant l'exemple du même chien apprenant à rapporter une balle lancée par son maître. Dans cette situation, le chien est à nouveau l'agent qui interagit avec l'environnement, représenté par le jardin où ils jouent. Les actions possibles du chien incluent courir vers la balle, aboyer ou simplement rester immobile. Les récompenses sont des friandises que le maître donne au chien lorsqu'il rapporte la balle.

L'horizon en RL se réfère à la durée sur laquelle les récompenses futures sont prises en compte dans la prise de décision de l'agent. Si l'horizon est court, le chien accorde plus d'importance aux récompenses immédiates. Cependant, si l'horizon est long, le chien prend en compte les récompenses futures plus lointaines. Par exemple, s'il décide de s'entraîner à courir lentement vers la balle, il pourrait obtenir une récompense plus grande une fois qu'il l'aura rapportée. Dans ce cas, le chien sacrifie la récompense immédiate en échange d'une récompense plus grande dans le futur.

Un concept lié à l'horizon est celui des "récompenses pondérées" (ou *discounted rewards* en anglais). Cela signifie que les récompenses futures sont souvent moins importantes que les récompenses immédiates. Le facteur de décroissance, notée par γ, est utilisé pour modéliser cet effet. L'équation pour les récompenses pondérées est donnée par :

$$R_t = \sum_{t=1}^{T} \gamma^{t-1} r_t$$

où R_t représente la récompense pondérée à l'instant t, r_t est la récompense à l'instant t, $\gamma \in [0, 1]$ est le facteur de décroissance, et $T \in [1, \infty]$ est l'horizon. Le facteur γ détermine à quel point les récompenses futures sont prises en compte par rapport aux récompenses immédiates. Si γ est proche de 0, les récompenses futures auront moins d'impact, tandis que

si γ est proche de 1, les récompenses futures auront plus d'importance dans les décisions de l'agent.

Ainsi, l'horizon en RL détermine la façon dont l'agent considère les conséquences à court et à long terme de ses actions, ce qui influence la manière dont il prend des décisions pour optimiser ses récompenses.

1.2 Dynamiques

1.2.1 Linéaire

Dans un premier temps, une couche en apprentissage machine fait référence à une unité de base dans un réseau de neurones artificiels. Elle exécute la transformation des entrées reçues en sorties en appliquant des opérations mathématiques sur ces entrées. Chaque couche traite les données de manière séquentielle, permettant ainsi au réseau de neurones de réaliser des tâches complexes en combinant plusieurs couches afin d'apprendre des motifs et des caractéristiques des données fournies. La couche linéaire aussi nommée *fully connected* [60] est certainement la plus utilisée en apprentissage machine. Cette dynamique est représentée par les équations affines

$$y_j = \sum_{i=1}^{M} x_i W_{ij}^{\text{in}} + b_j \tag{1.2.1}$$

où $j \in [1, \dots, N]$ et ses paramètres sont :

- y_j désigne la j-ième sortie de la couche ;
- W_{ij}^{in} désigne l'élément (i, j) de la matrice de connexions avec l'entrée du réseau ;
- b_j le j-ième biais de la couche associé au neurone j ;
- x_i désigne la i-ième entrée de la couche de taille M.

1.2.2 Linéaire récurrente

La couche linéaire peut aussi se généraliser à une couche récurrente en y ajoutant un état caché arbitraire qui sera nommé V^t. Ce système dynamique est formé de l'équation de récurrence suivante :

$$V_j^{t+\Delta t} = \sum_{k=1}^{M} x_k^t \, W_{kj}^{in} + \sum_{i=1}^{N} V_i^t \, W_{ij}^{rec} + b_j \tag{1.2.2}$$

$$y_j^{t+\Delta t} = V_j^{t+\Delta t} \tag{1.2.3}$$

où $j \in [1, \dots, N]$ et ses paramètres sont :

- x_k^t désigne la k-ième entrée de la couche à l'instant t, avec $k \in \{1, \dots N\}$;
- V_i^t désigne l'état caché du i-ième neurone du réseau récurrent de N neurones à l'instant t ;
- y_j désigne la j-ième sortie de la couche ;

- W^{in}_{kj} désigne l'élément (k, j) de la matrice de connexion avec l'entrée du réseau ;
- W^{rec}_{ij} désigne l'élément (i, j) de la matrice de connexion de la couche ;
- b_j est le biais du j-ième neurone du réseau récurrent.

L'introduction de la récurrence soulève la nécessité de clarifier les différents types d'entrées possibles pour un réseau. Le premier type est externe, où x représente une source externe de stimuli. Un autre type est la source récurrente, provenant du réseau lui-même, où l'entrée du réseau correspond à la sortie au pas de temps précédent : $x^{t+\Delta t} = y^t$. Le troisième type d'entrée est interne au modèle, où l'entrée d'une couche est la sortie de la couche précédente au même instant t, soit $x^t_\ell = y^t_{\ell-1}$, avec ℓ, l'indice de la couche.

Une autre notion importante est que la matrice de connectivité récurrente W^{rec} peut être omise. Dans ce cas, toute la récurrence de la dynamique est contrôlée par la matrice de connectivité W^{in}, lorsque $x^{t+\Delta t}$ dépend de y^t. Lorsqu'une seule couche est utilisée avec récurrence, W^{in} peut être directement considérée comme des connexions récurrentes. Cependant, dans le cas multicouche, cette récurrence devient plus complexe à interpréter, bien qu'elle reste présente.

1.2.3 *Leaky-integrate* (LI)

La dynamique *leaky-integrate* (LI) est considérée comme la forme la plus simple de dynamique récurrente, car elle implique l'intégration de la somme pondérée des entrées dans la couche. Elle peut être perçue comme une généralisation de la couche linéaire (section 1.2.1) en récurrence. De plus, la dynamique LI ajoute un facteur de fuite qui agit comme un filtre passe-bas sur le signal de sortie. En raison de l'absence de génération d'impulsion, cette dynamique est rarement utilisée comme couche d'entrée ou couche cachée dans les réseaux neuronaux à impulsions (SNN). Cependant, la sortie continue de cette couche peut être avantageuse lorsqu'elle est utilisée en fin de réseau. On l'appelle couramment une couche de lecture ou une couche de sortie dans ce contexte. L'intégration temporelle de cette dynamique est réalisée à l'aide des équations de récurrences

$$V^{t+\Delta t}_j = \kappa V^t_j + \sum_{k=1}^{M} x^t_k W^{\text{in}}_{kj} + b_j \tag{1.2.4}$$

$$y^{t+\Delta t}_j = V^{t+\Delta t}_j \tag{1.2.5}$$

telle que proposée par Bellec et al. [17]. Les paramètres de l'équation (1.2.4) sont :

- V^t_j désigne le potentiel du neurone j au pas de temps t où $j \in [1, \ldots, N]$;
- Δt désigne le pas de temps de l'intégration ;
- κ la constante de décroissance exprimée par $\kappa = e^{-\Delta t/\tau_{\text{mem}}} \approx \left(1 - \frac{\Delta t}{\tau_{\text{mem}}}\right)$;
- y_j désigne la j-ième sortie de la couche ;
- W^{in}_{kj} désigne l'élément (k, j) de la matrice de connexions avec l'entrée du réseau ;
- b_j le j-ième biais de la couche associé au neurone j ;

· x_k^t désigne la k-ième entrée de la couche au temps t de taille M.

1.2.4 *Adaptative leaky-integrate-and-fire* (ALIF)

La dynamique *adaptative leaky-integrate-and-fire* (ALIF) est une modification du modèle *leaky-integrate-and-fire* (LIF) inspirée de Bellec et al. [17] (sera présenté plus en détails dans la section 2.6.1), qui introduit une fonction d'activation à la dynamique LI (section 1.2.3). Le modèle LIF représente le potentiel synaptique et les impulsions d'un neurone au fil du temps. Bien que la forme de ce potentiel ne soit pas considérée comme réaliste [14], le moment auquel le potentiel dépasse un certain seuil est pertinent. Ce potentiel est déterminé par les équations de récurrences

$$V_j^{t+\Delta t} = \left(\alpha V_j^t + \sum_{i=1}^{N} z_i^t W_{ij}^{\text{rec}} + \sum_{k=1}^{M} x_k^t W_{kj}^{\text{in}}\right)\left(1 - z_j^t\right) \tag{1.2.6}$$

où $j \in [1, \ldots, N]$.

ALIF est très similaire à la dynamique LIF, car elle utilise la même équation de mise à jour du potentiel (équation (1.2.6)). La différence réside dans le fait que le seuil de potentiel varie dans le temps et dépend de l'entrée du neurone, contrairement à un seuil constant dans le cas de LIF. En effet, le seuil est augmenté à chaque impulsion de sortie, puis il est réduit avec un certain taux afin de revenir à sa valeur initiale V_{th}. La sortie du neurone j à l'instant t, notée z_j^t, est définie par l'équation

$$z_j^{t+\Delta t} = H(V_j^{t+\Delta t} - A_j^{t+\Delta t}) \tag{1.2.7}$$

où la fonction $H(\cdot)$ est la fonction Heaviside définie comme

$$H(x) = \begin{cases} 1 & \text{quand} \quad x \geq 0\,; \\ 0 & \text{sinon.} \end{cases} \tag{1.2.8}$$

La mise à jour du seuil d'activation est décrite par l'équation

$$A_j^{t+\Delta t} = V_{\text{th}} + \beta a_j^{t+\Delta t} \tag{1.2.9}$$

où la variable d'adaptation $a_j^{t+\Delta t}$ est définie par l'équation

$$a_j^{t+\Delta t} = \rho a_j^t + z_j^t \tag{1.2.10}$$

avec un facteur de décroissance ρ décrit par

$$\rho = e^{-\Delta t/\tau_a} \tag{1.2.11}$$

et β un facteur d'amplification plus grand que 1 et typiquement équivalent à $\beta \approx 1.6$ [17]. Finalement, la sortie de la couche est dictée par

$$y_j^{t+\Delta t} = z_j^{t+\Delta t}. \tag{1.2.12}$$

Un avantage significatif de la dynamique ALIF par rapport au LIF est sa capacité à conserver des informations sur le long terme. En effet, comme l'ont démontré Bellec et al. [17], la dynamique ALIF est analogue à un *long short term memory* (LSTM) utilisé en apprentissage automatique. Cette capacité accrue de mémoire est attribuée au fait que l'ALIF dispose de plusieurs variables d'état qui conservent l'information au fil du temps. La première variable d'état est le potentiel qui s'intègre dans le temps et est réinitialisée à chaque impulsion. La deuxième variable est le seuil adaptatif, qui n'est jamais réinitialisé pendant toute la durée de l'intégration. En revanche, LIF ne possède qu'un potentiel variant dans le temps, ce qui signifie que ce modèle conserve uniquement les informations des pas de temps précédents jusqu'à la prochaine impulsion où le neurone est remis à sont état initial.

Adaptative-leaky-integrate-and-fire with explicit synaptic current (SpyALIF)

La dynamique *Adaptative-leaky-integrate-and-fire with explicit synaptic current* (SpyALIF) est une variante plus complexe de la dynamique ALIF (section 1.2.4), offrant une plus grande expressivité. Cette variante, inspirée par Neftci et al. [29], comprend deux équations différentielles : une pour le potentiel membranaire des neurones et une pour le courant synaptique. Bien que l'équation différentielle supplémentaire qui modélise le courant synaptique soit inspirée par Neftci et al., la composante adaptative de cette dynamique est inspirée par Bellec et al. [17]. L'équation

$$I_{\text{syn},j}^{t+\Delta t} = \alpha I_{\text{syn},j}^{t} + \sum_{i=1}^{N} z_i^t W_{ij}^{\text{rec}} + \sum_{k=1}^{M} x_k^t W_{kj}^{\text{in}} \tag{1.2.13}$$

présente la mise à jour du courant synaptique du neurone j utilisant l'intégration d'Euler, tandis que l'équation

$$V_j^{t+\Delta t} = \left(\beta V_j^t + I_{\text{syn},j}^{t+\Delta t}\right)\left(1 - z_j^t\right) \tag{1.2.14}$$

présente la mise à jour du potentiel membranaire du même neurone. Les autres équations d'état provenant de l'ALIF restent inchangées et la sortie de la couche reste toujours

$$z_j^{t+\Delta t} = H(V_j^{t+\Delta t} - A_j^{t+\Delta t}); \tag{1.2.15}$$

$$y_j^{t+\Delta t} = z_j^{t+\Delta t}. \tag{1.2.16}$$

Dans les précédentes équations (1.2.13) et (1.2.14), les paramètres sont :

- $I_{\text{syn},j}^t$ désigne le courant synaptique du neurone j au pas de temps t ;
- V_j^t désigne le potentiel du neurone j au pas de temps t ;
- N désigne le nombre de neurones du réseau récurrent ;
- Δt désigne le pas de temps de l'intégration ;
- α la constante de décroissance du courant exprimé par $\alpha = e^{-\Delta t/\tau_{\text{syn}}}$;
- β la constante de décroissance du potentiel exprimé par $\beta = e^{-\Delta t/\tau_{\text{mem}}}$;
- W_{kj}^{in} désigne l'élément (k, j) de la matrice de connexions avec l'entrée du réseau ;

- W_{ij}^{rec} désigne l'élément (i, j) de la matrice de connexions récurrentes avec la convention que i est connecté vers j $(i \to j)$;
- x_k^t désigne la k-ième entrée de la couche au temps t de taille M.

1.3 Algorithmes d'apprentissage

Dans le cadre d'un modèle neuronal tel que le modèle séquentiel qui sera présenté dans la section 2.6.4, plusieurs couches de neurones sont organisées en séquence pour former un ensemble cohérent. Ces différentes couches peuvent avoir des dynamiques différentes (section 2.6.1) ainsi que des méthodes d'apprentissage variées. En effet, dans certains cas, il peut être nécessaire d'utiliser un réseau où toutes les couches apprennent par rétropropagation du gradient en utilisant l'algorithme de rétropropagation à travers le temps [60], tandis que dans d'autres applications, une méthode d'apprentissage plus simple et plus rapide est préférable. Il est par exemple possible de souhaiter que seule la couche de sortie optimise ses poids, tandis que les autres couches du réseau restent statiques, ce qui confère au modèle la propriété d'un réseau réservoir [61]. De plus, dans certaines applications, l'apprentissage par renforcement est nécessaire, ce qui demande des méthodes d'apprentissage spécifiques pour les réseaux et peut être complexe à mettre en œuvre. Dans cette section, différentes méthodes d'apprentissage sont présentées, commençant par les méthodes d'apprentissage classiques (sections 1.3.1, 1.3.2 et 1.3.3), pour ensuite aborder l'algorithme d'optimisation le plus répandu en apprentissage par renforcement (section 1.3.4).

1.3.1 Rétropropagation à travers le temps

La plupart des réseaux résultants des dynamiques présentées dans la section 2.6.1 sont récurrents et demandent une stratégie adaptée afin d'optimiser leurs paramètres (les matrices de poids synaptiques W) pour effectuer une tâche quelconque. La rétropropagation à travers le temps ou *Backpropagation-through-time* en anglais, (BPTT) [17, 62] est certainement le type d'apprentissage le plus populaire au sein de la communauté de l'apprentissage machine. En effet, cette méthode est en fait une extension de la rétropropagation du gradient utilisé dans les réseaux de neurones classiques. Ce qui en fait l'état de l'art pour entraîner des réseaux récurrents en apprentissage profond. Essentiellement, ce dernier algorithme est la simple application de la descente de gradient et de la règle de dérivées en chaîne. Une illustration du fonctionnement et du flot d'information peut être observée à la figure 1.3.1. En considérant l'entrée du modèle à optimiser comme x, la sortie de ce modèle $\hat{y}$ et la sortie désirée y, la fonction d'erreur définie par l'utilisateur est dénoté $E(\hat{y}, y)$. La descente de gradient stochastique permettant de minimiser cette quantité est effectuée par l'intermédiaire de

$$W_{ij} \mapsto W_{ij} + \Delta W_{ij} \tag{1.3.1}$$

qui représente la mise à jour de poids synaptiques avec

$$\Delta W_{ij} = -\eta \frac{\partial E(\hat{y}, y)}{\partial W_{ij}} \tag{1.3.2}$$

selon un taux d'apprentissage η.

La règle de dérivées en chaîne dicte donc que la dérivée de l'erreur s'écrive comme suit :

$$\frac{\partial E(\hat{y}, y)}{\partial W_{ij}} = \sum_t \frac{\partial E(\hat{y}, y)}{\partial h_j^t} \frac{\partial h_j^t}{\partial W_{ij}} \tag{1.3.3}$$

où la quantité h_j^t correspond à la fonction d'activation du neurone j au temps t.

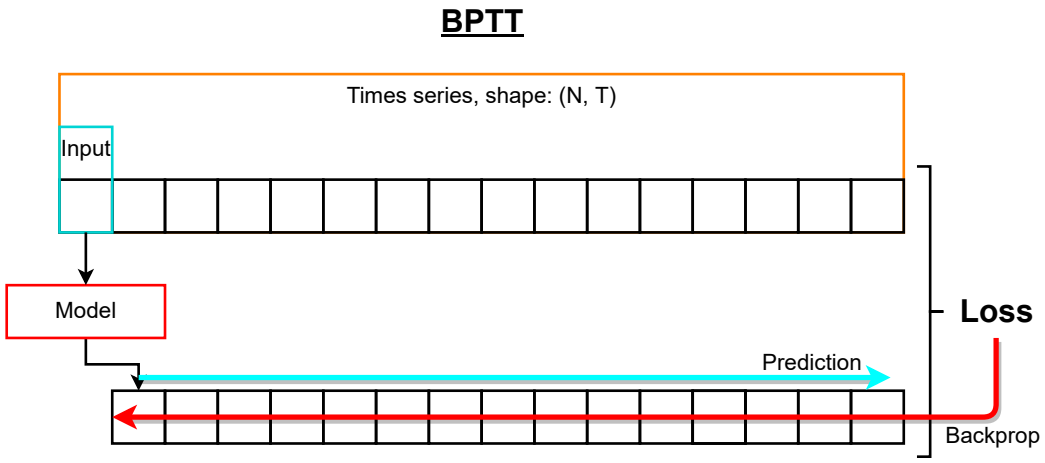

FIGURE 1.3.1 – Fonctionnement de la rétropropagation à travers le temps.

Dans l'objectif de clarifier le fonctionnement de BPTT voici un exemple correspondant au cas $N = 1$ et $T = 3$ avec une seule couche linéaire contenant deux paramètres ω et b dotée de la fonction d'activation identité $h(x) = x$. Dans cet exemple, l'entrée du modèle sera le scalaire x et la sortie désirée sera le vecteur $y = [y^0, y^1, y^2]$. Le vecteur de sortie $\hat{y}$ du modèle sera donc

$$\begin{aligned}
\hat{y}^0 &= h(x\omega + b), \\
\hat{y}^1 &= h(\hat{y}^0\omega + b), \\
\hat{y}^2 &= h(\hat{y}^1\omega + b),
\end{aligned}$$

ou plus explicitement

$$\begin{aligned}
\hat{y}^0 &= h(x\omega + b), \\
\hat{y}^1 &= h(h(x\omega + b)\omega + b), \\
\hat{y}^2 &= h(h(h(x\omega + b)\omega + b)\omega + b).
\end{aligned}$$

Pour simplifier le tout, en utilisant $h(x) = x$, il est possible de simplifier les équations comme

$$\begin{aligned}
\hat{y}^0 &= x\omega + b \\
\hat{y}^1 &= (x\omega + b)\omega + b \\
\hat{y}^2 &= ((x\omega + b)\omega + b)\omega + b.
\end{aligned}$$

La fonction d'erreur E dans cet exemple sera l'erreur quadratique $E(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$ ou plus explicitement

$$\begin{aligned}
E(\hat{y}^0, y^0) &= \frac{1}{2}\big((x\omega + b) - y^0\big)^2 \\
E(\hat{y}^1, y^1) &= \frac{1}{2}\big(((x\omega + b)\omega + b) - y^1\big)^2 \\
E(\hat{y}^2, y^2) &= \frac{1}{2}\big((((x\omega + b)\omega + b)\omega + b) - y^2\big)^2
\end{aligned}$$

où les dérivées partielles par rapport aux paramètres seront

$$\begin{aligned}
\frac{\partial E(\hat{y}^0, y^0)}{\partial \omega} &= x(x\omega + b - y^0) \\
\frac{\partial E(\hat{y}^1, y^1)}{\partial \omega} &= (2x\omega + b)\big((x\omega + b)\omega + b - y^1\big) \\
\frac{\partial E(\hat{y}^2, y^2)}{\partial \omega} &= (2b\omega + b + 3x\omega^3)(x\omega^3 - b(\omega^2 + \omega + 1) - y^2)
\end{aligned}$$

et

$$\begin{aligned}
\frac{\partial E(\hat{y}^0, y^0)}{\partial b} &= x\omega + b - y^0 \\
\frac{\partial E(\hat{y}^1, y^1)}{\partial b} &= (\omega + 1)(\omega(x\omega + b) + b - y^1) \\
\frac{\partial E(\hat{y}^2, y^2)}{\partial b} &= (\omega^2 + \omega + 1)(\omega(\omega(x\omega + b) + b) + b - y^2).
\end{aligned}$$

La mise à jour des paramètres ω et b s'effectuera ainsi

$$\begin{aligned}
\omega &\mapsto \omega - \eta\left\{\frac{\partial E(\hat{y}^0, y^0)}{\partial \omega} + \frac{\partial E(\hat{y}^1, y^1)}{\partial \omega} + \frac{\partial E(\hat{y}^2, y^2)}{\partial \omega}\right\} \\
b &\mapsto b - \eta\left\{\frac{\partial E(\hat{y}^0, y^0)}{\partial b} + \frac{\partial E(\hat{y}^1, y^1)}{\partial b} + \frac{\partial E(\hat{y}^2, y^2)}{\partial b}\right\}
\end{aligned}$$

et plus explicitement

$$\begin{aligned}
\omega \mapsto \omega - \eta\Big\{ & x(x\omega + b - y^0) \\
& + (2x\omega + b)\big(w(x\omega + b) + b - y^1\big) \\
& + (2b\omega + b + 3x\omega^3)(b(\omega^2 + \omega + 1) + x\omega^3 - y^2)\Big\} \\
b \mapsto b - \eta\Big\{ & x\omega + b - y^0 \\
& + (\omega + 1)(\omega(x\omega + b) + b - y^1) \\
& + (\omega^2 + \omega + 1)(\omega(\omega(x\omega + b) + b) + b - y^2)\Big\}.
\end{aligned}$$

Dans le cas des réseaux de neurones à impulsions (SNN) et donc des dynamiques associées, la fonction d'activation est la fonction Heaviside (1.2.8). Évidemment, la dérivée de la fonction Heaviside est la fonction delta de Dirac, ce qui cause de grandes instabilités numériques. Afin de régler cette instabilité numérique, plusieurs options sont possibles. Parmi celles-ci, une des approche très utilisée est d'approximer la fonction Heaviside comme une fonction sigmoïde rapide (*fast sigmoid*) [63]

$$\sigma(v) = \frac{1}{2}\left(\frac{v}{1 + |v|} + 1\right) \tag{1.3.4}$$

où v est une variable réelle quelconque, et dont la dérivée est

$$\frac{\mathrm{d}\sigma(v)}{\mathrm{d}v} \propto \frac{1}{(1 + \xi|v|)^2}. \tag{1.3.5}$$

À noter que dans la précédente dérivée, le paramètre ξ est ajouté afin d'être en mesure de paramétrer la pente de la dérivée. De plus, seulement la proportionnalité de la dérivée est considérée puisque son coefficient sera absorbé par le taux d'apprentissage de la descente de gradient.

Par exemple, dans le cas de la dynamique LIF, l'état caché du neurone j au temps t sera

$$h_j^t = \sigma\big(v_j^t - V_{\mathrm{th}}\big), \tag{1.3.6}$$

avec

$$v_j^t = \left(\alpha V_j^{t-\Delta t} + \sum_{i=1}^{M} x_i^{t-\Delta t} W_{ij}\right)\Big(1 - z_j^{t-\Delta t}\Big), \tag{1.3.7}$$

où V_j^t est le potentiel du neurone j au temps t et V_{th} est le seuil d'activation du neurone. La dérivée de la fonction d'activation est donc l'approximation suivante :

$$\frac{\partial h_j^t}{\partial W_{ij}} \approx \frac{\partial h_j^t}{\partial v_j^t}\frac{\partial v_j^t}{\partial W_{ij}}, \tag{1.3.8}$$

qui est développé comme

$$\frac{\partial h_j^t}{\partial W_{ij}} \approx \frac{1}{\left(1+\xi\left|v_j^t - V_{\text{th}}\right|\right)^2}\frac{\partial v_j^t}{\partial W_{ij}}, \tag{1.3.9}$$

ce qui donne plus explicitement l'équation suivante :

$$\frac{\partial h_j^t}{\partial W_{ij}} \approx \frac{1}{\left(1+\xi\left|v_j^t - V_{\text{th}}\right|\right)^2}\left(x_i^{t-\Delta t} + W_{ij}\frac{\partial x_i^{t-\Delta t}}{\partial W_i j}\right)\left(1 - z_j^{t-\Delta t}\right). \tag{1.3.10}$$

Une autre alternative proposée par Bellec et collab. est d'utiliser la fonction (1.3.12) [17] comme approximation de la dérivée de h_j^t par rapport à v_j^t, pouvant être inséré dans (1.3.8). Celle-ci devient donc

$$\frac{\partial h_j^t}{\partial v_j^t} \approx \psi_j^t \tag{1.3.11}$$

avec

$$\psi_j^t = \frac{\gamma_{\text{pd}}}{V_{\text{th}}}\max\left(0, \quad 1 - \left|\frac{v_j^t - A_j^t}{V_{\text{th}}}\right|\right) \tag{1.3.12}$$

où $\gamma_{\text{pd}} = 0,3$ pour les neurones ALIF, et pour les neurones LIF, A_j^t est remplacé par V_{th}.

1.3.2 Rétropropagation à travers le temps tronqué

La rétropropagation à travers le temps (BPTT section 1.3.1) de base donne de bons résultats la plupart du temps. En effet, comme énoncé dans la section attitrée à cet algorithme, il s'agit de l'état de l'art. Or, cette méthode n'est pas parfaite. Dans le cas, où la série temporelle en question est très grande, le réseau apprendra à prédire la moyenne de l'activité. Ce problème arrive souvent lorsque la série temporelle est grande par rapport au taux de changement de son activité. Afin de pallier à un tel désavantage, une solution simple est de séparer le problème en sous-problèmes. La méthode reste très semblable, il suffit de faire une rétropropagation à tous les τ pas de temps. En choisissant un τ comparable à la période de variation de l'activité, le réseau aura tendance à prioriser la prédiction sur le court terme au lieu de sur le long terme (ce qui est préférable à certains moments). Ce nouvel algorithme se nomme la rétropropagation à travers le temps tronqué ou plus communément nommé TBPTT, la *Truncated-Backpropagation-through-time*. Une illustration du fonctionnement et du flot d'information de cet algorithme peut être observée à la figure 1.3.2. L'équation de dérivée de BPTT (1.3.3) devient donc

$$\frac{\partial E(x,y)}{\partial W_{ij}} = \sum_{t=t_0}^{t_0+\tau}\frac{\partial E(x,y)}{\partial h_j^t}\cdot\frac{\partial h_j^t}{\partial W_{ij}} \tag{1.3.13}$$

où t_0 est le pas de temps initial de la rétropropagation et τ est le nombre de pas de temps à rétropropager. La TBPTT peut aussi être vue comme une fenêtre glissante sur la série temporelle telle une convolution en appliquant l'algorithme BPTT.

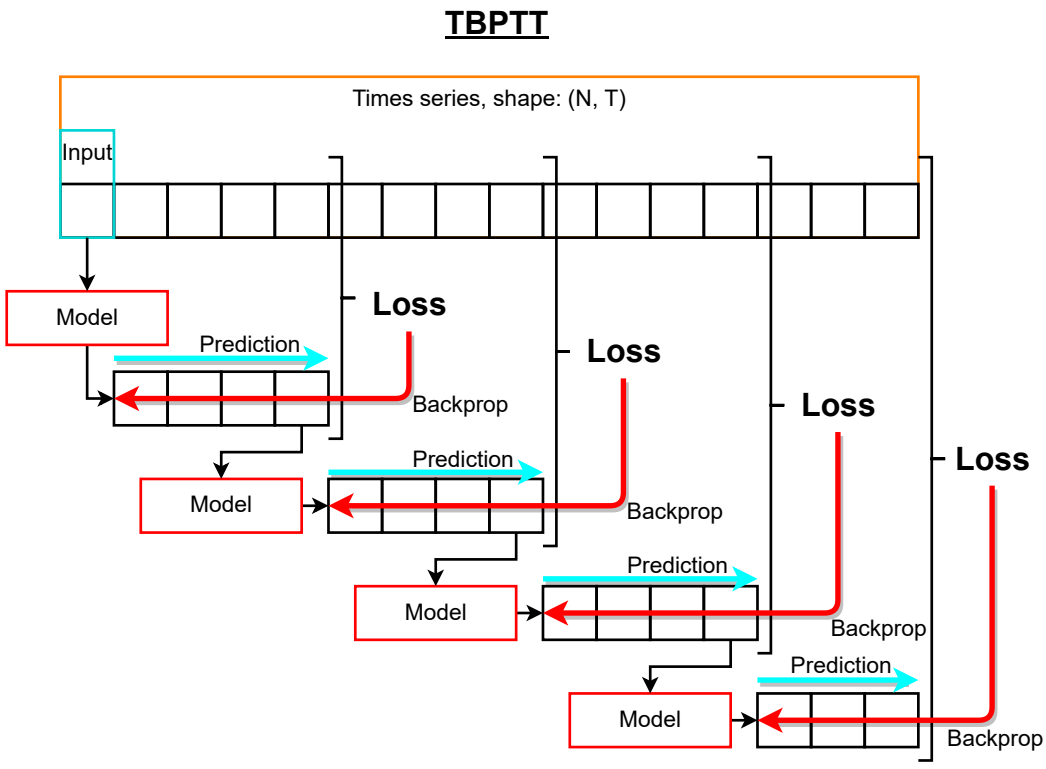

FIGURE 1.3.2 – Fonctionnement de la rétropropagation à travers le temps tronqué.

Dans l'objectif de clarifier le fonctionnement de TBPTT, reprenons l'exemple précédent avec $N = 1$ et $T = 3$, avec une seule couche linéaire contenant deux paramètres ω et b doté de la fonction d'activation identité $h(x) = x$. Dans cet exemple, l'entrée du modèle sera le scalaire x et la sortie désirée sera le vecteur $y = [y^0, y^1, y^2]$ et l'hyperparamètre τ sera 1. Pour rendre les mises à jour des poids plus claires, un indice t sera ajouté aux paramètres. Le vecteur de sortie $\hat{y}$ du modèle sera donc

$$\begin{aligned}\hat{y}^0 &= h(x\omega_0 + b_0)\\ \hat{y}^1 &= h(\hat{y}^0\omega_1 + b_1)\\ \hat{y}^2 &= h(\hat{y}^1\omega_2 + b_2).\end{aligned}$$

Pour simplifier le tout, en utilisant $h(x) = x$, il est possible de simplier les équations comme

$$\begin{aligned}\hat{y}^0 &= x\omega_0 + b\\ \hat{y}^1 &= \hat{y}^0\omega_1 + b_1\\ \hat{y}^2 &= \hat{y}^1\omega_2 + b_2.\end{aligned}$$

La fonction d'erreur E dans cet exemple sera l'erreur quadratique $E(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$ ou plus

explicitement

$$E(\hat{y}^0, y^0) = \frac{1}{2}\big((x\omega_0 + b_0) - y^0\big)^2$$
$$E(\hat{y}^1, y^1) = \frac{1}{2}\big((\hat{y}^0\omega_1 + b_1) - y^1\big)^2$$
$$E(\hat{y}^2, y^2) = \frac{1}{2}\big((\hat{y}^1\omega_2 + b_2) - y^2\big)^2$$

où les dérivées partielles par rapport aux paramètres seront

$$\frac{\partial E(\hat{y}^0, y^0)}{\partial \omega_0} = x(x\omega_0 + b_0 - y^0)$$
$$\frac{\partial E(\hat{y}^1, y^1)}{\partial \omega_1} = \hat{y}^0(\hat{y}^0\omega_1 + b_1 - y^1)$$
$$\frac{\partial E(\hat{y}^2, y^2)}{\partial \omega_2} = \hat{y}^1(\hat{y}^1\omega_2 + b_2 - y^2)$$

et

$$\frac{\partial E(\hat{y}^0, y^0)}{\partial b_0} = x\omega_0 + b_0 - y^0$$
$$\frac{\partial E(\hat{y}^1, y^1)}{\partial b_1} = \hat{y}^0\omega_1 + b_1 - y^1$$
$$\frac{\partial E(\hat{y}^2, y^2)}{\partial b_2} = \hat{y}^1\omega + b_2 - y^2.$$

La mise à jour des paramètres ω et b s'effectuera en trois pas de temps consécutifs :

$$\omega_1 \mapsto \omega_0 - \eta\frac{\partial E(\hat{y}^0, y^0)}{\partial \omega_0}, \qquad b_1 \mapsto b_0 - \eta\frac{\partial E(\hat{y}^0, y^0)}{\partial b_0},$$
$$\omega_2 \mapsto \omega_1 - \eta\frac{\partial E(\hat{y}^1, y^1)}{\partial \omega_1}, \qquad b_2 \mapsto b_1 - \eta\frac{\partial E(\hat{y}^1, y^1)}{\partial b_1},$$
$$\omega_3 \mapsto \omega_2 - \eta\frac{\partial E(\hat{y}^2, y^2)}{\partial \omega_2}, \qquad b_3 \mapsto b_2 - \eta\frac{\partial E(\hat{y}^2, y^2)}{\partial b_2}$$

et plus explicitement

$$\omega_1 \mapsto \omega_0 - \eta\big(x(x\omega_0 + b_0 - y^0)\big), \qquad b_1 \mapsto b_0 - \eta\big(x\omega_0 + b_0 - y^0\big),$$
$$\omega_2 \mapsto \omega_1 - \eta\big(\hat{y}^0(\hat{y}^0\omega_1 + b_1 - y^1)\big), \qquad b_2 \mapsto b_1 - \eta\big(\hat{y}^0\omega_1 + b_1 - y^1\big),$$
$$\omega_3 \mapsto \omega_2 - \eta\big(\hat{y}^1(\hat{y}^1\omega_2 + b_2 - y^2)\big), \qquad b_3 \mapsto b_2 - \eta\big(\hat{y}^1\omega_2 + b_2 - y^2\big).$$

1.3.3 *Recursive-Least-Square* (RLS)

Les méthodes d'optimisation basées uniquement sur la rétropropagation comme BPTT (section 1.3.1) et TBPTT (section 1.3.2) sont relativement simples à implémenter et à utiliser. À cause de cette simplicité et des bonnes performances que procurent ces algorithmes, elles sont actuellement les plus populaires et devraient rester la première approche aux problèmes

d'optimisation. Par contre, ces méthodes sont relativement lentes à converger et ont parfois tendance à rester dans des minima locaux des fonctions à minimiser. D'un autre côté l'algorithme RLS permet d'accélérer la convergence de l'optimisation, mais requiert plus de mémoire et est plus complexe à calculer. Toutefois, les avantages de cet algorithme surpassent ses désavantages pour certains problèmes, tel que la prédiction de séries temporelles dans le cadre des neurosciences [3].

Dans un premier temps, la méthode générale de RLS inspirée des filtres de Kalman [43] sera présentée. Ensuite, plusieurs variantes de cette méthode seront expliquées telles que RLS-Inputs, RLS-Outputs, RLS-Grad, RLS-Jacobien et finalement RLS-Jacobien avec gain.

La méthode générale d'optimisation RLS utilisée dans NeuroTorch est grandement inspirée de l'algorithme MRLS (*Modified-Recursive-Least-Square*) de Al-Batah et collab. [44] et de Azimi-Sadjadi et collab. [64, 65], de l'algorithme CURBD de Perich et collab. [3], du travail de Zhang et collab. [45] ainsi que des filtres de Kalman [43].

On considère un réseau possédant L couches, où l'ensemble de paramètres de la couche ℓ sont représentés par $\hat{\Theta}_\ell$, avec $\ell \in [0, L-1]$ où zéro est l'indice de la couche d'entrée. La matrice d'entrée et de sortie du réseau seront dénotés respectivement x et $\hat{y}$ tandis que le tenseur de sortie désiré est y. La matrice x est de taille (B, f_{in}) avec B la taille du lot de données courant (*minibatch*) et f_{in} le nombre de valeurs d'entrée. Pour les entrées et sorties spécifiques aux couches du modèle, celles-ci seront notées avec l'indice ℓ approprié. L'erreur de prédiction est donc définie comme

$$\varepsilon = \hat{y} - y \tag{1.3.14}$$

où les quantités $\hat{y}$ et y seront de taille (B, f_{out}) avec f_{out} le nombre de valeurs de sortie. La quantité ε sera donc aussi de taille (B, f_{out}). Dans les prochaines équations, l'indice i dénotera le numéro de l'itération d'optimisation. Cet index sera utilisé seulement pour lever certaines ambiguïtés afin de ne pas trop alourdir la notation. La fonction de coût à minimiser est donc définie comme $C(\hat{\Theta})$ par l'équation

$$C(\hat{\Theta}) = \frac{1}{B} \sum_{b=1}^{B} \varepsilon_b \Lambda^{-1} \varepsilon_b^T \tag{1.3.15}$$

où le symbole Λ désigne une matrice symétrique et positive de taille (f_{out}, f_{out}). De plus, la quantité B réfère au nombre de données prédites. Pour arriver à minimiser l'équation (1.3.15), les étapes suivantes sont effectuées.

(1) La matrice de corrélation inverse P est initialisée comme $\delta\mathbb{I}$ où $\mathbb{I}$ est la matrice identité et δ est un scalaire habituellement égal à 1. Le facteur d'oubli λ ainsi que le facteur de gain κ sont aussi initialisés selon les valeurs au tableau 1.3.1. La taille et la signification de la matrice P

changent dépendemment de la sous-méthode d'RLS utilisée. Cette quantitié sera donc mieux définie dans les prochaines sous-sections.

(2) Dans un premier temps, l'erreur de prédiction ε est calculée à l'aide de l'équation

$$\overline{\varepsilon}_i = \frac{1}{B}\sum_{b=1}^{B} \varepsilon_{i,b} \tag{1.3.16}$$

où l'indice b fait référence à l'indice de la donnée dans la *minibatch*. L'erreur de prédiction moyennée à l'itération i sera définie par $\overline{\varepsilon}_i$ et sera de taille $(1, f_{out})$.

(3) Le tenseur d'état ϕ_i correspond essentiellement à l'état du modèle à l'itération i. Sa taille est donnée par $(D_0, ..., D_{d-1})$, où D et d sont définie par le choix de la méthode d'RLS. Cette quantité est utilisée par la suite pour mettre à jour les paramètres du modèle et ainsi minimiser la fonction de coût. Le choix de ϕ va déterminer la nature de la matrice de corrélation inverse P de taille (D_{d-1}, D_{d-1}) ainsi que le gain de Kalman K présenté à l'étape 4. En général, la matrice de corrélation inverse peut donc être vue comme la corrélation inverse des valeurs d'état dans ϕ_i. Cette étape correspond donc au calcul de ϕ_i.

(4) Le gain de Kalman K de taille $(D_{d-1}, ..., D_0)$ est ensuite calculé. Cette valeur à l'itération i est définie par

$$K_i = P_{i-1}\phi_i^T. \tag{1.3.17}$$

(5) Le facteur de normalisation h est ensuite défini par

$$h_i = (\lambda + \kappa\phi_i K_i)^{-1} \tag{1.3.18}$$

où λ et κ sont respectivement des facteurs d'oubli et de gain. À noter que h_i n'a rien à voir avec la fonction d'activation de l'unité i d'un réseau de neurones, désignée par le même symbole à la section précédente.

(6) Par la suite, les paramètres $\hat{\Theta}_\ell$ sont mis à jour par l'équation de réccurence

$$\hat{\Theta}_{i,\ell} = \hat{\Theta}_{i-1,\ell} - h_i K_i \overline{\varepsilon}. \tag{1.3.19}$$

Il est à noter qu'en général, la quantité $h_i K_i \overline{\varepsilon}$ peut aussi être considérée comme une approximation du gradient de la fonction de coût par rapport aux paramètres du modèle. Dans ce qui suit, ce gradient est désigné par $\nabla\hat{\Theta}_{i,\ell}$ au lieu de $\nabla_{\hat{\Theta}_{i,\ell}} C(\hat{\Theta}_i)$ pour simplifier la notation. Cette interprétation permet entre autres d'utiliser les différents algorithmes d'optimisation du gradient offerts par les modules d'autodifférentiation comme PyTorch.

(7) Finalement, la matrice P, considérée comme la matrice de corrélation inverse de l'état ϕ et définie par l'équation

$$P_i = \lambda P_{i-1} - \kappa h_i K_i K_i^T \tag{1.3.20}$$

est mise à jour à l'aide du gain de Kalman.

(8) Les étapes (2) à (7) sont répétées jusqu'à atteindre la convergence. La convergence, dans ce contexte, est définie par l'utilisateur et correspond généralement au point où la fonction de coût a suffisamment diminué pour considérer que la tâche a été accomplie.

Variable	Initialisation	Description
δ	1.0	Facteur d'initialisation de P.
P	$\delta\mathbb{I}$	Matrice de corrélation inverse.
λ	0.99	Facteur d'oubli des anciennes valeurs de P.
κ	0.99	Facteur de gain des nouvelles valeurs de P.

Tableau 1.3.1 – Résumé de l'initialisation des variables de l'algorithme RLS.

Variable	Taille	Description
x	(B, f_{in})	Entrée du réseau.
$\overline{x}$	$(1, f_{in})$	Entrée moyenne du réseau.
$\hat{y}$	(B, f_{out})	Sortie du réseau.
$\hat{\overline{y}}$	$(1, f_{out})$	Sortie moyenne du réseau.
y	(B, f_{out})	Sortie désirée du réseau.
ε	(B, f_{out})	Erreur de prédiction.
$\hat{\Theta}_\ell$	(N_{in}, N_{out})	Paramètres du réseau de la couche ℓ.
ϕ	$(D_0, ..., D_{d-1})$	Tenseur d'état du modèle.
K	$(D_{d-1}, ..., D_0)$	Gain de Kalman.
h	$(D_0, ..., D_{d-2}, D_{d-2}, ..., D_0)$	Facteur de normalisation du gain.
P	(D_{d-1}, D_{d-1})	Matrice de corrélation inverse.

Tableau 1.3.2 – Résumé des variables de l'algorithme RLS.

RLS-*Inputs*

La version *Inputs* est certainement la méthode la plus simple à implémenter, mais possède tout de même beaucoup de contraintes. En effet, dans cette version, le tenseur d'état ϕ, défini comme

$$\phi = \frac{1}{B}\sum_{b}^{B} x_b, \tag{1.3.21}$$

est la moyenne des données d'entrée. Ceci amène les différentes tailles résumées au tableau 1.3.3. De plus, il est possible de déduire à partir des différentes tailles et des équations précédentes que les contraintes suivantes doivent être respectées :

— $f_{in} = N_{in}$;
— $f_{out} = N_{out}$.

Variable	Taille	Description
ϕ	$(1, f_{in})$	Tenseur d'état du modèle.
K	$(f_{in}, 1)$	Gain de Kalman.
h	$(1,)$	Facteur de normalisation du gain.
P	(f_{in}, f_{in})	Matrice de corrélation inverse.

Tableau 1.3.3 – Résumé des variables de l'algorithme RLS-*Inputs*.

RLS-*Outputs*

La version *Outputs* inspirée de l'algorithme CURBD de Perich et al. [3] est aussi simple à implémenter que RLS-*Inputs*, mais possède une contrainte supplémentaire. En effet, dans cette version, le tenseur d'état ϕ défini par l'équation

$$\phi = \frac{1}{B} \sum_{b}^{B} \hat{y}_b \tag{1.3.22}$$

est la moyenne des données prédites par le modèle. Ceci amène les différentes tailles résumées au tableau 1.3.4. De plus, il est possible de déduire à partir des différentes tailles et des équations précédentes que les contraintes suivantes doivent être respectées :

— $f_{out} = N_{in}$;
— $f_{out} = N_{out}$;
— ce qui implique nécessairement que $N_{in} = N_{out}$.

Variable	Taille	Description
ϕ	$(1, f_{out})$	Tenseur d'état du modèle.
K	$(f_{out}, 1)$	Gain de Kalman.
h	$(1,)$	Facteur de normalisation du gain.
P	(f_{out}, f_{out})	Matrice de corrélation inverse.

Tableau 1.3.4 – Résumé des variables de l'algorithme RLS-*Outputs*.

RLS-Grad

La version Grad, inspirée du travail de Zhang et al. [45], est un peu plus compliquée à implémenter que RLS-*Inputs*, mais possède moins de contraintes. En effet, dans cette version, le tenseur d'état ϕ, défini par l'équation

$$\phi = \frac{1}{B} \sum_{b}^{B} x_b, \tag{1.3.23}$$

est la moyenne des données du modèle. De plus, l'étape **(6)** qui consiste à la mise à jour des paramètres du modèle de RLS est modifier par la suivante :

$$\hat{\Theta}_{i,\ell} = \hat{\Theta}_{i-1,\ell} - h_i P_i \nabla_{\hat{\Theta}_{i,\ell}} \mathcal{L}(\hat{y}_i, y_i) \tag{1.3.24}$$

où $\mathcal{L}(\cdot)$ est une fonction de perte scalaire et différentiable définie par l'utilisateur. Ceci amène les différentes tailles résumées au tableau 1.3.5. De plus, il est possible de déduire à partir des différentes tailles et des équations précédentes que les contraintes suivantes doivent être respectées :

— $f_{in} = N_{in}$.

Variable	Taille	Description
ϕ	$(1, f_{in})$	Tenseur d'état du modèle.
K	$(f_{in}, 1)$	Gain de Kalman.
h	$(1,)$	Facteur de normalisation du gain.
P	(f_{in}, f_{in})	Matrice de corrélation inverse.

Tableau 1.3.5 – Résumé des variables de l'algorithme RLS-Grad.

RLS-Jacobien

La version jacobienne inspirée de l'algorithme MRLS (*Modified-Recursive-Least-Square*) de Al-Batah et collab. [44] est certainement la version la plus compliquée à implémenter, mais possède le moins de contraintes. En effet, dans cette version, le tenseur d'état ϕ est défini par

$$\phi = \begin{bmatrix} \frac{\partial \hat{\tilde{y}}_1}{\partial \hat{\Theta}_{\ell,1}} & \cdots & \frac{\partial \hat{\tilde{y}}_1}{\partial \hat{\Theta}_{\ell,N_{flat}}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \hat{\tilde{y}}_{f_{out}}}{\partial \hat{\Theta}_{\ell,1}} & \cdots & \frac{\partial \hat{\tilde{y}}_{f_{out}}}{\partial \hat{\Theta}_{\ell,N_{flat}}} \end{bmatrix}^T, \tag{1.3.25}$$

ce qui est le jacobien de la prédiction du réseau par rapport aux paramètres de celui-ci. Dans cette dernière, les indices i ont été omis pour ne pas alourdir la notation. De plus, la constante N_{flat} est définie comme $N_{flat} = N_{in} \cdot N_{out}$ étant le nombre total de paramètres de la couche ℓ. Ce qui amène les différentes tailles résumées au tableau 1.3.6. Finalement, il est à noter que dans cette version de l'algorithme, le facteur $h = 1$.

Variable	Taille	Description
ϕ	(N_{flat}, f_{out})	Tenseur d'état du modèle.
K	(f_{out}, N_{flat})	Gain de Kalman.
h	$(1,)$	Facteur de normalisation du gain ($h = 1$).
P	(f_{out}, f_{out})	Matrice de corrélation inverse.

Tableau 1.3.6 – Résumé des variables de l'algorithme RLS-Jacobien.

RLS-Jacobien avec gain

La version jacobienne avec gain est une version modifiée de la version Jacobien. En effet, dans cette version, le tenseur d'état ϕ est encore défini par l'équation (1.3.25) qui est le jacobien de la prédiction du réseau par rapport aux paramètres de celui-ci. L'objectif ici est d'utiliser le facteur de normalisation h_i ce qui va changer la mise à jour du gradient de la fonction de

coût par rapport aux paramètres avec

$$\nabla\hat{\Theta}_{i,\ell} = (K_i h_i)^T \bar{\varepsilon}_i^T \tag{1.3.26}$$

ainsi que la mise à jour de la matrice P avec

$$P_i = \lambda P_{i-1} - \kappa(K_i h_i) K_i^T. \tag{1.3.27}$$

Ceci amène les différentes tailles résumées au tableau 1.3.7.

Variable	Taille	Description
ϕ	(N_{flat}, f_{out})	Tenseur d'état du modèle.
K	(f_{out}, N_{flat})	Gain de Kalman.
h	(N_{flat}, N_{flat})	Facteur de normalisation du gain.
P	(f_{out}, f_{out})	Matrice de corrélation inverse.

Tableau 1.3.7 – Résumé des variables de l'algorithme RLS-Jacobien avec gain.

1.3.4 *Proximal Policy Optimization* (PPO)

L'algorithme d'optimisation *Proximal Policy Optimization* proposé par Schulman et al. [46], considéré comme l'état de l'art et l'un des plus populaires en RL, sert essentiellement à changer les paramètres d'un modèle afin d'effectuer le même type d'association que le chien a fait entre l'ordre de s'asseoir et ses morceaux de foie séché. La popularité de PPO est due à sa simplicité associée à une grande performance pour tout type de problème.

Comme la majorité des problèmes d'apprentissage machine ont comme objectif de minimiser une fonction par descente de gradient, une fonction de perte est requise. C'est en fait cette fonction de perte qui donne toute la puissance à PPO.

Le premier terme de celle-ci est représenté par l'équation

$$\mathcal{L}^{\text{CLIP}}(\theta) = -\hat{\mathbb{E}}_k\Big[\min(r_k(\theta)\hat{A}_k,\ \text{clip}(r_k(\theta),\ 1-\varepsilon,\ 1+\varepsilon)\hat{A}_k)\Big] \tag{1.3.28}$$

où k est l'index de la donnée d'entraînement et θ est l'ensemble des paramètres du modèle à optimiser. Cette fonction est construite de façon à ce que chaque changement du modèle soit petit. En effet, la fonction *clip* avec le ratio ε (généralement de valeur 0.1 ou 0.2) force la fonction de perte à être petite et donc force le modèle à rester dans un voisinage local. Ce type d'approche est souvent utilisé en apprentissage par renforcement, puisque les grands changements dans les paramètres du réseau semblent avoir tendance à le faire diverger. De son côté, le terme $r_k(\theta)$ est défini par

$$r_k(\theta) = \frac{\pi_\theta(a_k|s_k)}{\pi_{\theta_{\text{old}}}(a_k|s_k)} \tag{1.3.29}$$

et représente le ratio entre les prédictions des deux estimateurs. En effet, l'estimateur π_θ est le réseau à son état courant tandis que $\pi_{\theta_{\text{old}}}$ est le même réseau, mais à l'itération d'entraînement

précédente. Considérant que π_θ peut être un scalaire, un vecteur, une matrice ou plus généralement un tenseur dépendamment de l'environnement dans lequel est l'agent, la division entre les deux versions de l'estimateur est exécutée élément par élément. De plus, les variables a_k et s_k correspondent respectivement à l'action prise et à l'observation reçue pour l'expérience k. Le terme $\hat{A}_k$ de l'équation (1.3.28) est l'avantage calculé à partir de

$$\hat{A}_t = \sum_{j=t}^{T-1} (\gamma\lambda)^{j-t}\delta_j \tag{1.3.30}$$

et représente l'avantage de prendre une certaine action à l'observation s_k. Dans cette dernière, l'indice t représente le pas de temps de la trajectoire de l'agent dans l'environnement.

Les indices t et k sont tous deux utilisés en tant qu'indices pour désigner des éléments d'un ensemble de données. Cependant, l'indice t fait référence à un épisode dans une trajectoire où les épisodes sont séquentiels et continus, ce qui implique un ordre spécifique. En revanche, l'indice k fait référence à une donnée d'entraînement k, qui correspond également à un épisode, mais qui est sélectionnée de manière aléatoire parmi l'ensemble des trajectoires de l'agent pour l'itération d'entraînement en cours. Il est à noter que dans certains cas, selon le choix de l'utilisateur, l'indice k peut être traité comme l'indice t, notamment si l'ordre séquentiel est nécessaire pour la tâche ou l'analyse spécifique en cours.

Pour revenir à l'équation, δ est une estimation de la récompense pondérée (plus connue sous le terme *discounted rewards*). Celle-ci est définie par l'équation

$$\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t) \tag{1.3.31}$$

et introduit la fonction scalaire $V_\phi(s_t)$ communément nommée *value function*. Cette fonction est utilisée pour estimer la récompense pondérée d'un état s_t et dépend des paramètres ϕ du réseau. La variable r_t est la récompense obtenue par l'environnement au temps t. Les facteurs de décroissance γ et λ doivent être plus petits que 1 et sont généralement équivalents à 0.99. Afin de clarifier l'estimateur de récompenses pondérées, ce dernier tente de minimiser la fonction de perte

$$\mathcal{L}^{\mathrm{VF}} = \hat{\mathbb{E}}_k\left[\left(V_\phi(s_k) - V_k^{\mathrm{target}}\right)^2\right] \tag{1.3.32}$$

où

$$V_t^{\mathrm{target}} = \sum_{j=t}^{T-1} \gamma^{j-t} r_t. \tag{1.3.33}$$

Dans cette dernière, le terme V_t^{target} est aussi connu, dans le vocabulaire technique du RL, comme étant l'estimation du *returns* à l'instant t de la trajectoire. De plus, la fonction de perte (1.3.32) est présentée sous la forme d'une erreur quadratique moyenne (*MSE* ou $L2$), mais pourrait aussi être changée par une autre fonction de régression comme la fonction $L1$ lisse (*smooth* $L1$ *Loss*).

Finalement, la fonction de perte totale à minimiser est représentée par l'équation

$$\mathcal{L}^{\text{CLIP+VF+S}} = \hat{\mathbb{E}}_k \left[\mathcal{L}_k^{\text{CLIP}} + c_{\text{VF}} \mathcal{L}_k^{\text{VF}} - c_{\text{S}} S[\pi_\theta](s_k) \right] \tag{1.3.34}$$

où trois nouveaux termes sont introduits. Les deux premiers c_{VF} et c_{S} sont seulement des constantes de pondération généralement respectivement équivalentes à 0.5 et 0.01. La dernière quantité $S[\pi_\theta](s_k)$ est un bonus d'entropie. Ce terme d'entropie est certainement le plus complexe. Pour bien comprendre ce dernier, il faut s'attarder à comment l'agent prend ses décisions. Dans le cas où les actions à prendre sont discrètes, la sortie du réseau est une distribution de probabilités avec une entrée associée à chacune des actions possibles. Dans le cas avec des actions continues, une distribution de probabilité multimodale est créée avec l'incertitude du réseau comme étant la covariance de la distribution et les valeurs de sorties du réseau comme étant les moyennes. Alors, il y a une distribution de probabilité associée à la prise de décision de l'agent indépendamment du type d'actions. En entraînement, l'agent prend donc une action en tirant une valeur de sa distribution et l'exécute dans l'environnement. Cette méthode consistant à tirer aléatoirement une action d'une distribution est du *Thompson Sampling* [66-68]. Ce qui est intéressant avec cette méthode est que le ratio exploration-exploitation est intrinsèquement pris en compte dans la prise de décision. Pour revenir au terme d'entropie, celui-ci est défini comme étant l'entropie de la distribution de la prise de décision.

Finalement, une fois la fonction de perte calculée, il est possible d'utiliser n'importe quel algorithme d'optimisation pour la minimiser. Dans le cas général, l'optimiseur Adam est utilisé avec la descente de gradient.

1.4 Régularisations

1.4.1 Régularisation *Lp*

La régularisation en apprentissage automatique vise à contrôler la complexité d'un modèle pour éviter le surapprentissage, où des modèles trop complexes mémorisent les données d'entraînement plutôt que de généraliser pour de nouvelles données. Elle introduit des contraintes sur les paramètres du modèle pour favoriser la simplicité et la stabilité, par exemple en restreignant les valeurs des paramètres ou en ajoutant des termes de pénalité à la fonction de perte. Cette approche améliore la généralisation du modèle sur des données non vues. En neuroscience, la régularisation joue un rôle essentiel pour contrôler les caractéristiques des réseaux neuronaux. Elle permet de moduler l'éparsité (*sparsity*) des réseaux, reflétant souvent l'organisation cérébrale, et favorise des modèles biologiquement plausibles en introduisant des contraintes sur les paramètres du réseau.

En apprentissage automatique, une forme courante de régularisation consiste à pénaliser la p-norme des paramètres du modèle, connue sous le nom de *Lp loss*. Cette méthode est décrite

par l'équation suivante :

$$\text{loss}(\theta) = \lambda \sum_{i=1}^{N} \|\theta_i\|_p, \tag{1.4.1}$$

où θ représente la liste des paramètres d'entrée, N est le nombre de paramètres, θ_i est le i-ième paramètre pouvant être un scalaire, un vecteur, une matrice ou plus généralement un tenseur, et $\|\cdot\|_p$ désigne la p-norme dictée par l'équation

$$\|v\|_p = \left(\sum_{k=1}^{K} |v|^p \right)^{1/p}, \tag{1.4.2}$$

où v est une variable réelle quelconque de taille K. Il est important de noter que les régularisations populaires $L1$ et $L2$ sont des cas particuliers de la régularisation Lp, avec $p = 1$ et $p = 2$ respectivement.

1.4.2 Ratio Excitateurs-Inhibiteurs

La loi de Dale [69, 70], en neuroscience, stipule qu'un neurone individuel libère un seul type de neurotransmetteur et donc qu'un neurone est exclusivement excitateur ou inhibiteur. C'est-à-dire qu'un neurone possède des connexions externes uniquement positives ou négatives. Introduire ce type de contrainte dans un modèle est intéressant pour le rendre plus interprétable biologiquement. Or, en utilisant la loi de Dale dans un modèle, il est souvent souhaitable d'avoir un ratio biologiquement plausible de neurones excitateurs et inhibiteurs [71-74]. C'est dans cette perspective que la régularisation décrite par

$$\text{loss}(\theta) = \lambda \sum_{i=1}^{N} |(\mathbb{E}[\text{sign}(\theta_i)] + 1) - 2\,\text{target}| \tag{1.4.3}$$

est présentée. Dans cette équation, θ est la liste des paramètres d'entrée, N est le nombre de paramètres, θ_i est le i-ième paramètre pouvant prendre la forme d'un scalaire, d'un vecteur, d'une matrice ou plus généralement, d'un tenseur. De plus, $\text{sign}(\theta_i)$ est le signe de l'élément θ_i, $\mathbb{E}[\text{sign}(\theta_i)]$ est la moyenne des signes des éléments dans θ_i, target est la valeur cible de la fraction de neurones excitateurs et λ est le poids de la régularisation.

Par exemple, dans le cas où les paramètres à régulariser sont

$$W_0 = \begin{bmatrix} 0.1 & -0.2 \\ 1.4 & 0.5 \end{bmatrix}, \qquad W_1 = \begin{bmatrix} 1 & 2 & -3 \end{bmatrix},$$

et que la cible pour le ratio est 0.5, signifiant que la moitié des neurones sont excitateurs, le calcul de la perte de régularisation se fera ainsi

$$\begin{aligned} \text{loss}(x) &= \lambda\{|(\mathbb{E}[\text{sign}(W_0)] + 1) - 2 \cdot \text{target}| + |(\mathbb{E}[\text{sign}(W_1)] + 1) - 2 \cdot \text{target}|\} \\ &= \lambda\{|(0.5 + 1) - 1| + |(0.33 + 1) - 1|\} \\ \implies \text{loss}(x) &= 0.833\lambda. \end{aligned}$$

Après avoir calculé la perte liée à cette régularisation, les paramètres peuvent être optimisés en effectuant une descente de gradient.

En raison de sa forme générale, cette fonction de régularisation peut être appliquée à divers formats de paramètres, tels que scalaires, vecteurs, matrices, etc. Cependant, elle se combine particulièrement bien avec l'application de la loi de Dale, qui sera expliquée plus en détail dans le chapitre 2 à l'équation (2.6.17). Dans ce chapitre, il est exposé que les paramètres d'une matrice de connectivité sont divisés en deux parties : une partie qui détermine les signes des connexions sortantes et une partie qui détermine les normes des connexions en valeur absolue. La partie concernant les signes des connexions sortantes, désignée par le vecteur S, détermine si les neurones de la matrice de connectivité résultante sont excitateurs ou inhibiteurs. Ainsi, c'est pour optimiser ce vecteur que la présente régularisation a été conçue. Pour mieux comprendre, l'exemple précédent peut être repris en considérant que l'on veut seulement optimiser le vecteur

$$S = \Big[1, -1, 1\Big],$$

ce qui donne la perte suivante :

$$\begin{aligned} \text{loss}(x) &= \lambda \left|(\mathbb{E}[\text{sign}(S)] + 1) - 2 \cdot \text{target}\right| \\ &= \lambda \left|(0.333 + 1) - 1\right| \\ \implies \text{loss}(x) &= 0.333\lambda. \end{aligned}$$

1.5 Transformations

1.5.1 Taux de décharge à impulsions (*LinearRateToSpikes*)

Les modèles avec une dynamique à impulsions requièrent généralement une entrée sous forme d'impulsions (des zéros ou des uns). Toutefois, la plupart du temps, les données du monde réel sont sous forme continue, c'est-à-dire, contenant des valeurs appartenant à l'ensemble des réels ($\mathbb{R}$). Afin de pouvoir tout de même donner des impulsions aux modèles, il faut transformer ces valeurs réelles (x) en séries temporelles de zéros et de uns. Pour y arriver, une méthode est de normaliser les données d'entrée entre une valeur minimale ($v_{\min}$) et maximale ($v_{\max}$) et calculer le taux de décharge r avec l'équation

$$r = \frac{x - v_{\min}}{v_{\max} - v_{\min} + \epsilon}$$

où ϵ est une très petite constante utilisée pour s'assurer d'une stabilité numérique lors de la division. Ces taux de décharges sont ainsi changés en périodes p dicté par

$$p = \lfloor T \cdot (1 - r) \rfloor$$

afin d'être en mesure de produire les séries temporelles de T pas de temps.

1.5.2 Pixels vers impulsions (*ImgToSpikes*)

Il est aussi possible de transformer les valeurs continues en séries temporelles d'impulsions de façon logarithmique. Cette transformation suggérée par Neftci [29] est présentée par l'équation

$$T_{\text{spike},j} = \begin{cases} \tau_x \log\left(\frac{x_j}{x_j - x_{\text{thr}}}\right) & x_j \geq x_{\text{thr}} \\ T_{\max} & x_j < x_{\text{thr}} \end{cases} \tag{1.5.1}$$

où x_j correspond au pixel j de l'image x et $T_{\text{spike},j}$ correspond au temps auquel l'impulsion sera envoyée pour ce pixel. L'image x étant une matrice en deux dimensions de taille [hauteur, largeur] (des images en noir et blanc sont considérées ici) est donc transformée en tenseur en trois dimensions de taille [hauteur, largeur, t_{int}] où t_{int} correspond au temps d'intégration du système.

Les constantes τ_x, $T_{\max}$ et x_{thr} correspondent respectivement au temps caractéristique de décroissance exponentielle, au temps d'impulsion maximal (typiquement équivalent au temps d'intégration t_{int}) et au seuil d'activation.

1.5.3 Transformation constante

Il est possible de remarquer que les dynamiques ne requièrent pas nécessairement des impulsions en entrée afin de générer des impulsions en sortie. Une astuce possible utilisée pour faire la transformation de l'entrée du réseau est de seulement considérer les valeurs d'entrée comme des constantes pour un certain temps T. De cette façon, les valeurs d'entrée deviennent des séries temporelles et il est possible de mettre une couche d'entrée avec une matrice de poids identité statique (qui n'est pas optimisée durant l'entraînement). Par exemple, en utilisant une dynamique LIF décrite par

$$V_j^{t+\Delta t} = \left(\alpha V_j^t + x_j\right)\left(1 - z_j^t\right) \tag{1.5.2}$$

pour un certain vecteur de pixel x, on obtient les pixels transformés z_j

$$z_j^t = H(V_j^t - V_{\text{th}}) \tag{1.5.3}$$

où H est la fonction Heaviside. On remarque donc que les pixels d'entrée sont maintenant considérés comme des courants externes constants. Avec cette stratégie, il est possible d'effectuer un grand ensemble de transformations en changeant la dynamique ou en rendant les poids synaptiques ajustables au fil de l'entraînement par exemple.

1.5.4 Auto-Encodeur à Impulsions

Un auto-encodeur [20, 75-77] est un type de modèle très utilisé en apprentissage machine pour effectuer diverses tâches telles que la réduction dimensionnelle, l'apprentissage non supervisé, l'apprentissage semi-supervisé, la régularisation, etc. Ce type de modèle, comme illustré à la

figure 1.5.1, est constitué de deux sous-modèles et consiste à prendre des données en entrée, transformer ces données dans une représentation latente et ensuite reconstruire les données à l'aide de cette représentation. Le premier sous-modèle est donc celui qui est responsable de transformer l'entrée dans la représentation latente. Cette dernière est en général la plus petite, la plus éparse et la plus distinctive possible. En effet, une représentation plus petite ou à plus basse dimension aide non seulement à la compressibilité des données, mais aussi à l'interprétation puisqu'il est plus facile de visualiser un espace à petite dimension qu'à haute dimension d'où l'utilisation d'outils comme UMAP [78], PCA [79] ou T-SNE [80] pour la visualisation de données. Une représentation latente plus éparse aide surtout à la compressibilité des données puisqu'une matrice dense (avec des valeurs non nulles partout) prend beaucoup plus d'espace mémoire qu'une matrice éparse (avec beaucoup de valeurs nulles).

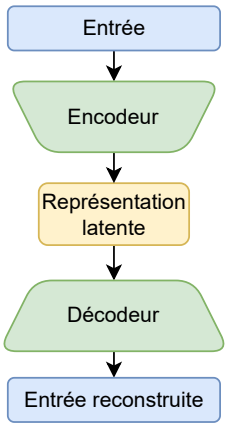

FIGURE 1.5.1 – Illustration d'un auto-encodeur. Les modules bleus de cette figure correspondent aux données d'entrées et de sorties de l'auto-encodeur. Les modules verts correspondent aux sous-modèles (encodeur et décodeur) de l'auto-encodeur. Le module jaune correspond à l'espace latent de l'auto-encodeur.

Finalement, l'espace latent doit bien être en mesure de capter les caractéristiques importantes des données d'entrées de façon à ce que chaque représentation latente de chacune des données soit distincte. Avec une telle représentation latente, il sera beaucoup plus facile pour le décodeur de faire son travail de reconstruction.

Les dynamiques à impulsions comme présentées dans ce travail sont de bonnes candidates pour effectuer un encodage de qualité [81] puisque la sortie de ce type de dynamique est par définition éparse. NeuroTorch permet ainsi de créer un auto-encodeur à impulsions et de spécifier la dynamique d'encodage et de décodage.

Dans ce chapitre, diverses dynamiques neuronales, algorithmes d'apprentissage, fonctions de régularisation et transformations utilisées en neurosciences ont été présentés, tout en fournissant les bases théoriques de l'apprentissage automatique pour comprendre le fonctionnement

et l'optimisation des modèles. Cependant, certaines dynamiques importantes et un algorithme d'apprentissage clé n'ont pas encore été abordés, car ils sont traités dans l'article présenté dans le prochain chapitre. Celui-ci permettra aux lecteurs de mieux comprendre le rôle de la librairie NeuroTorch en neurosciences computationnelles tout en complétant la présentation des éléments théoriques.

Chapitre 2

NeuroTorch : A Python library for neuroscience-oriented machine learning

2.1 Résumé

L'apprentissage automatique ou le *machine learning* (ML) est un outil puissant pour l'analyse de données en neurosciences. Cependant, l'aspect technique des algorithmes de ML limite leur utilisation conforme aux principes biologiques observés dans le cerveau. Pour remédier à cela, nous présentons NeuroTorch, une librairie Python de ML spécialement conçue pour les neuroscientifiques. NeuroTorch permet de construire des réseaux neuronaux récurrents avec des dynamiques à impulsions ou à taux de décharge, en tenant compte de contraintes biologiques telles que la loi de Dale. Elle propose diverses méthodes d'apprentissage, dont la rétropropagation dans le temps et la trace d'éligibilité. NeuroTorch a été testé sur des ensembles de données populaires et a obtenu des performances comparables à d'autres librairies. Il a aussi été utilisé sur des données d'activité neuronale chez le poisson-zèbre. Enfin, nous avons constaté que les modèles respectant la loi de Dale et les dynamiques à impulsions sont plus résilients aux ablations de connexion.

2.2 Abstract

Machine learning (ML) has become a powerful tool for data analysis, leading to significant advances in neuroscience research. While ML algorithms are proficient in general-purpose tasks, their highly technical nature often hinders their compatibility with the observed biological principles and constraints in the brain, thereby limiting their suitability for neuroscience applications. In this work, we introduce NeuroTorch, a comprehensive ML pipeline specifically designed to assist neuroscientists in leveraging ML techniques using biologically inspired neu-

ral network models. NeuroTorch enables the training of recurrent neural networks equipped with either spiking or firing-rate dynamics, incorporating additional biological constraints such as Dale's law and synaptic excitatory-inhibitory balance. The pipeline offers various learning methods, including backpropagation through time and eligibility trace forward propagation, aiming to allow neuroscientists to effectively employ ML approaches. To evaluate the performance of NeuroTorch, we conducted experiments on well-established public datasets for classification tasks, namely MNIST, Fashion-MNIST, and Heidelberg. Notably, NeuroTorch achieved accuracies that replicated the results obtained using the Norse and SpyTorch packages. Additionally, we tested NeuroTorch on real neuronal activity data obtained through volumetric calcium imaging in larval zebrafish. On training sets representing 9.3 minutes of activity under darkflash stimuli from 522 neurons, the mean proportion of variance explained for the spiking and firing-rate neural network models, subject to Dale's law, exceeded 0.97 and 0.96, respectively. Our analysis of networks trained on these datasets indicates that both Dale's law and spiking dynamics have a beneficial impact on the resilience of network models when subjected to connection ablations. NeuroTorch provides an accessible and well-performing tool for neuroscientists, granting them access to state-of-the-art ML models used in the field without requiring in-depth expertise in computer science.

2.3 Introduction

The early stages of machine learning (ML) were deeply influenced by neuroscience [82]. Key influences include the first abstract neuron models [83], artificial neural networks for data classification [84], recurrent neural networks with associative memory [85], and reinforcement learning [86]. Today, ML is reciprocating this influence by driving significant progress in the field of neuroscience. It facilitates the analysis of vast amounts of data generated by brain imaging techniques such as functional magnetic resonance imaging (fMRI) [87] and electroencephalography (EEG) [88] as well as the segmentation of cells in microscopy data [89]. Machine learning also aids in the diagnosis of brain diseases, such as Alzheimer's disease [90], Parkinson's disease [91], and schizophrenia [92], and contributes to the modeling of neural processes, including organizational principles in the sensory cortex [93]. Despite these advances, ML has yet to be broadly adopted across all neuroscience research. This may be attributed to the computational resources and advanced programming skills required for the implementation of some machine learning techniques, as well as a current lack of models that are either inspired by or constrained by biological plausibility, which would likely be more interpretable to neuroscientists [82, 94, 95].

Among all Machine Learning techniques, Recurrent Neural Networks (RNNs) [Fig. 2.3.1 (a)] uniquely incorporate fundamental structural features inspired by biological neural networks [96, 97], such as recurrent connections that mimic neural feedback loops and the capacity for both local and long-range connections. On the functional level, RNNs operate based on

non-linear dynamics governed by threshold-like activation functions, echoing the dynamical behavior regulating the firing rates of interconnected biological neurons [49]. Furthermore, RNNs are recognized as universal approximators of dynamical systems [55, 56], meaning that any time series can be modeled as the output of an adequately complex RNN. Consequently, RNNs exhibit a rich variety of dynamical behaviors - from attractor dynamics, through oscillatory, to even chaotic regimes. This diversity offers a versatile framework for understanding how computations can be performed by neural populations [98].

Owing to a diverse range of training algorithms [99-101], which especially permit the modification of connection weights to fit target data [Fig. 2.3.1 (b)–(c)], RNNs have garnered significant successes in neuroscience. They have been instrumental in unveiling altered intrahabenula interactions contributing to behavioral passivity in larval zebrafish [102], and identifying single-cell targets for circuit control in epileptic animals [103]. Very recently, RNNs have demonstrated high precision in replicating neuronal recordings and forecasting the behavior of animals in decision-making or reward-learning tasks.[104-107] The rapid progress in this field strongly suggests that single RNN models may soon possess the capability to simulate neural activity during complex behaviors and various perturbations within the relevant neural circuitry or behavioral contexts, making them a highly potent tool for systems neuroscience when utilized as surrogates for new experimental data.[108]

Another fundamental principle bridging ML and neurobiology comes from spiking neural networks (SNNs), which offer relatively realistic biological dynamics through their spike-based activity.[109, 110] In addition, the training of those networks substitutes the traditional backpropagation training with biologically plausible processes such as Spike-Timing-Dependent Plasticity (STDP) [16] and eligibility trace forward propagation (e-prop) [17], using local information rather than global gradients [18, 19]. By adopting these biologically inspired dynamics and learning mechanisms, deep learning models can better align with real neural processes, enabling the study of biological phenomena through numerical simulations. The emergence of neuromorphic processors, such as Intel's Loihi [22] and SpiNNaker [23], has played a crucial role in the growing popularity of spiking neural networks [24]. These processors leverage the specific characteristics of SNNs, including their sparsity and asynchronous nature, allowing for efficient time and energy utilization.[111-113]

All these advances plead in favour of systematically incorporating SNNs in any ML framework intended to assist neuroscientists in data analysis and neuronal mechanism modeling. However, the mainstream ML libraries like TensorFlow [27] and PyTorch [26], both written in Python [25], do not natively support SNNs. These libraries are primarily focused on the traditional artificial neural networks, and they do not include out-of-the-box support for the special neuron models and learning rules that are characteristic of SNNs. Even high-level librairies built on top of PyTorch, such as PyTorch Lightning [35] and Poutyne [36], while providing invaluable tools to expedite and simplify PyTorch development processes without compromi-

sing flexibility and control, do not extend their offerings to include models or training methods drawn from neuroscience.

To address this gap, more specialized libraries like Norse [28], SpyTorch [29], and snnTorch [30] have been developed to implement spiking neural network architectures, facilitating research in neuromorphic computing. These Python-based libraries provide a set of spiking learning layers that can be used with PyTorch, demonstrating their functionality in tasks such as the classification of popular datasets like MNIST [31, 32] and Heidelberg [33]. A more extensive catalog of SNN-centric libraries, each offering a diverse assortment of learning rules, is presented in the paper by Manna et al. [34]. Despite the breadth of available options, none of these libraries currently incorporate the e-prop learning rule —an algorithm of significant relevance given its biological plausibility and superior performance as demonstrated by Bellec et al. [17]. In addition, other Python packages have successfully incorporated a certain number of neurobiological constraints and techniques. The package proposed by Suarez et al. [10] constructs recurrent neural networks (RNNs) with firing-rate dynamics and network architecture based on real connectomes. These RNNs can be partially trained using reservoir computing, where only the weights of the readouts are trained, to evaluate their ability to store information. Although the learning rule is not biologically realistic and is limited to a portion of the network, this approach is relevant as it incorporates biological constraints through the network structure. Concurrently, yet another package has been developed by D'Amicelli et al. [11], which also employs reservoir computing with a connectome-constrained structure.

Given the pressing need for more realistic neural network models, including biologically plausible learning algorithms, coupled with the current lack of comprehensive ML libraries offering learning pipelines specifically tailored to neuroscience research, we have developed NeuroTorch [42]. This library, built upon PyTorch, leverages the latest advances in RNN and SNN libraries. It provides features for neuronal time-series analysis, regularization based on connectome metrics, and optimization methods such as backpropagation-through-time (BPTT) [18, 62], eligibility trace forward propagation (e-prop) [17], recursive-least-square (RLS) [3, 44, 45, 114], and proximal policy optimization (PPO) [46]. The library also incorporates popular dynamical models for neuronal activity, such as the firing-rate dynamics of Wilson-Cowan (WC) [15, 47-50] and the spiking dynamics leaky-integrate-and-fire with explicit synaptic current (SpyLIF) [29], along with other tools for neuromorphic computing research.

The paper presents an in-depth exploration of NeuroTorch. We start in Section 2.4 by highlighting the results obtained from various numerical experiments, validating the performance of NeuroTorch. Specifically, we demonstrate that NeuroTorch achieves comparable results to other existing packages when tested on benchmark datasets such as MNIST, Fashion-MNIST, and Heidelberg. Furthermore, we delve into the reproduction of neuronal activity patterns in the ventral habenula of a larval zebrafish using NeuroTorch's e-prop implementation. This showcases the capability of NeuroTorch to handle time series data from experimental measu-

rements. Finally, we explore the concept of resilience in relation to sparsity and hierarchical ablation, shedding light on the robustness of NeuroTorch's trained models [Fig. 2.3.1 (d)]. Moreover, we investigate the impact of Dale's principle (a.k.a Dale's law) [69, 70] on the trained models, providing insights into the significance of incorporating realistic biological rules in enhancing model performance and resilience against connection ablations. Figure 2.3.1 gives a conceptual overview of our computationnal experiments. In Section 2.5, we dissect these results further and identify certain limitations in our work while also proposing future avenues for exploration. Finally, Section 2.6 offers a detailed description of NeuroTorch, outlining its design, implementation, and key features, to provide readers with essential technical information about the framework.

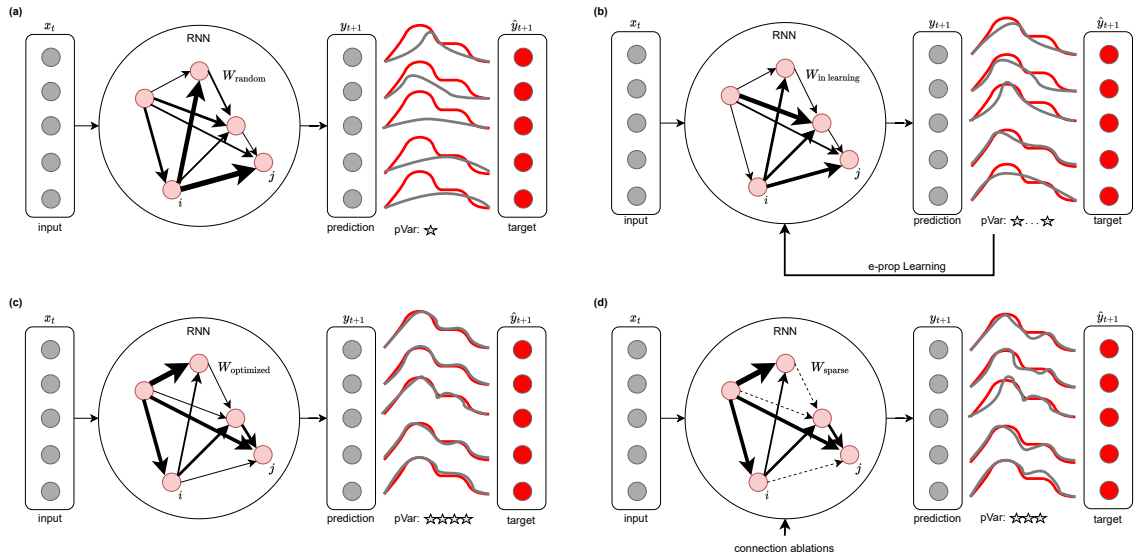

FIGURE 2.3.1 – Numerical experiment during which a RNN is trained to fit time series data. (a) A recurrent model is created with random weights to make time series predictions. The gray traces represent the model's predicted traces, while the red traces represent the targets. The pVar of these predictions is poor. (b) The model is then trained with the e-prop algorithm to optimize its weights and reproduce the target time series. The pVar gradually increases during training. (c) The now trained model is capable of reproducing the target time series with a high pVar. (d) We remove weights from the model to make it more sparse and observe the impact on the pVar.

2.4 Results

This section presents the results of our computational experiments conducted with NeuroTorch [42], the library developed and introduced in this paper. The experiments include testing the NeuroTorch pipeline with BPTT learning algorithm on benchmark datasets, demonstrating its effectiveness in classification models based on SNNs. Furthermore, we explore the application of NeuroTorch in the analysis of real time series data, specifically focusing on

neuronal measurements of the ventral habenula in a larval zebrafish. By optimizing firing-rate and spiking dynamics using the e-prop learning algorithm, we assess the effectiveness of NeuroTorch in handling time series data and capturing the dynamics of neuronal activity. Additionally, we investigate the resilience of the optimized networks against connection loss and examine its consequences on the properties of the connectome produced by the type of neuronal dynamics employed, as well as the imposition of yet another biologically realistic rule, namely Dale's Law. This analysis allows us to gain insights into the models' ability, when submitted to biologically plausible constraints, to maintain performance and adaptability in the face of parameter perturbations or changes in their connectivity.

2.4.1 Data classification benchmark

Before dealing with typical neuroscience data, we evaluated the performance of the NeuroTorch package and its BPTT algorithm by conducting experiments on three commonly used benchmark datasets for classification, namely MNIST [31, 32], Fashion-MNIST [115], and Heidelberg [33]. The two former datasets contain grayscale images of handwritten digits and fashion products, respectively, while the latter dataset contains spike trains representing audio recordings of spoken digits in both German and English. Our objective was to assess the functionality of NeuroTorch and determine how it compares to other established packages (SpyTorch [29] and Norse [28]) in terms of prediction accuracy when performing classification tasks.

The SNN models implemented using NeuroTorch, along with other selected packages, were tested to ensure fair and comparable evaluations. It is important to note that there are some differences in the training and implementation methods, which justify the variations in the results [see the method for more details on the implementation]. The models used in these experiments were not necessarily optimized specifically for the task at hand, but rather designed to provide a baseline assessment of the packages' performance. The results presented in Table 2.4.1 demonstrate that NeuroTorch performs comparably to the other packages. The models implemented using NeuroTorch achieve prediction accuracies that are on par with the established packages, confirming that NeuroTorch functions as intended.

2.4.2 Neuronal activity prediction

To investigate the applicability of the package to model neuronal activity data, we collected a neuronal activity dataset from the ventral habenula of a larval zebrafish. The dataset comprises recordings from 522 neurons, obtained through two-photon imaging on 6 dpf larval zebrafish expressing the genetically-encoded calcium indicator GCaMP6s. The neuronal activity of head-restrained larvae in agarose was captured under darkflash stimuli. We then extracted the time series describing the time evolution of each neuron's activity. Subsequently, we implemented two network models in NeuroTorch, one with fining-rate (WC) dynamics and

Dataset	Model	Package	Test accuracy [%]
MNIST	SpyLIF100+SpyLI	NeuroTorch	**96.37**
	SpyLIF100+SpyLI	SpyTorch [29]	94.60
	LIF100+FC	Norse [28]	95.35
Fashion-MNIST	SpyLIF100+SpyLI	NeuroTorch	**85.58**
	SpyLIF100+SpyLI	SpyTorch [29]	83.40
	LIF100+FC	Norse [28]	82.97
Heidelberg	SpyLIF200+SpyLI	NeuroTorch	74.16
	SpyLIF200+SpyLI	SpyTorch [29]	66.70
	LIF200+FC	Norse [28]	**75.68**

Table 2.4.1 – Prediction accuracy of different SNN classification models computed with the test-subset of the benchmark datasets MNIST, Fashion-MNIST and Heidelberg. The models are named as "<Input Dynamics><number of units>+<Output Dynamics>". The spiking dynamics are defined in Section 2.6.1. The number highlighted in bold represents the highest achieved performance for each dataset, indicating the best result. All models were trained using BPTT.

the other with spiking (SpyLIF-LPF) dynamics, each containing exactly 522 units. We trained these models with the learning algorithm e-prop to reproduce the activity of all recorded neurons. Once trained, a model can predict the activity of each neuron at all time $t > 0$ given the initial activity of all neurons at time $t = 0$. Figure 2.4.1 showcases heatmaps illustrating the activity of the zebrafish, along with corresponding predictions based on WC and SpyLIF-LPF dynamics. Neuronal measurements were categorised into 13 clusters using the k-means algorithm, effectively highlighting the distinct activity patterns observed in the data.

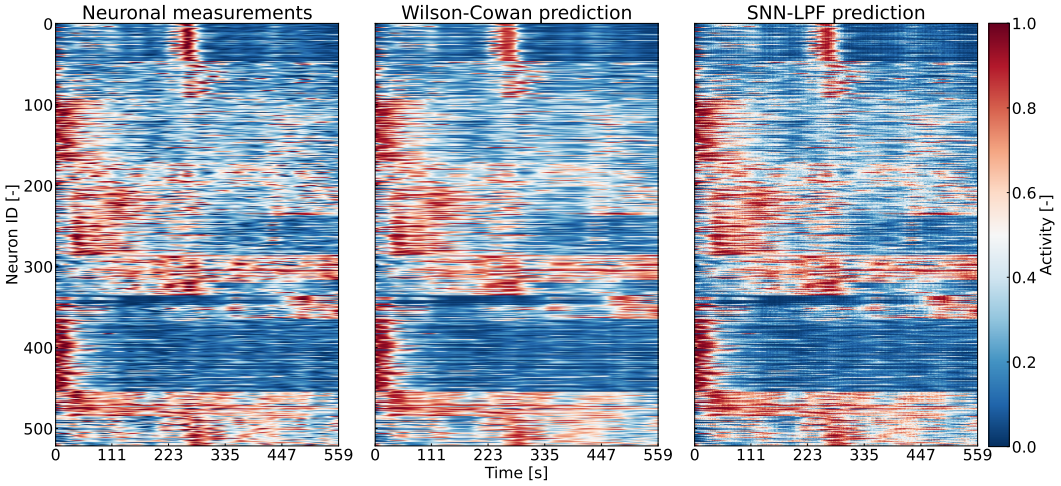

FIGURE 2.4.1 – Heatmaps describing the normalized neuronal activity in the ventral habenula of a larval zebrafish and corresponding predictions based on Wilson-Cowan and SpyLIF-LPF dynamics. The neuronal activity measurements were grouped into 13 neuron clusters by the k-means algorithm to highlight the different activity patterns.

Figure 2.4.2 shows neuronal measurements and corresponding predictions obtained using Neu-

roTorch. The real data describe the time evolution of neurons' activity in the ventral habenula, smoothed using Gaussian smoothing with $\sigma = 10$. The figure includes two metrics : the macro pVar (equation 2.6.18), which quantifies the overall similarity between the real and predicted series, and the average and standard deviation of the micro pVar (equation 2.6.19), which assesses the prediction accuracy at the individual neuron level. Panel **(a)** represents all original time series and WC predictions as two distinct trajectories projected into a 2D UMAP space [78, 116], confirming the overall good performance with a macro pVar of 0.96 and a micro pVar of 0.95 ± 0.03. Panel **(b)** compares the real series with a typical prediction obtained by NeuroTorch on the dataset. Panels **(c)** and **(d)** present the corresponding results obtained with SpyLIF-LPF dynamics. The overall prediction demonstrates a macro pVar of 0.97 with a micro pVar of 0.96 ± 0.01. These results highlight the accuracy and reliability of NeuroTorch in predicting time series of neuronal activity in the ventral habenula. Considering that the maximum pVar in this situation is 1.0, both WC and SpyLIF-LPF predictions are very close to the ground truth. The UMAP projection in 2D dimensions serves precisely to visualize how the entire modeled population behaves compared to the ground truth, providing a comprehensive overview of the prediction performance. Furthermore the panels with the UMAP projection suggest that not only have individual neuron activities been well predicted, but the entire time series is consistent. Overall, our library offers numerous tools to model neuronal activity recordings from a standard and widespread imaging modality. From trained network models, connectivity can be studied and deconstructed numerically, a process we describe in the following section.

2.4.3 Resilience of predictions under connection removal

The animal and human brains exhibit a remarkable level of sparsity, indicating that only a small fraction of all possible connections between neurons or brain regions is observed.[117-121] In the pursuit of developing biologically plausible neural network models, it becomes intriguing to explore methods for maximizing sparsity and creating networks that closely resemble the architecture of the brain. Building upon this line of investigation, we conducted a computational experiment aimed at determining how the choice of dynamics and the application of Dale's law could influence the capacity of a network model to maintain realistic neuronal activity while increasing its sparsity. We thus studied the resilience of different network models trained on calcium imaging data when subjected to connection loss.

We selected four types of recurrent network models, each having 522 units, as follows : we either chose the SpyLIF-LPF or the WC neuronal dynamics and we imposed or not the network connections to comply with Dale's principle. When the Dale's principle is enforced, every outgoing synaptic connections of a neuron are either positive or negative. All models were first trained with e-prop to adequately predict the neuronal activity data (see previous section). We then submitted the models to two distinct connection ablation strategies : (1) hierarchi-

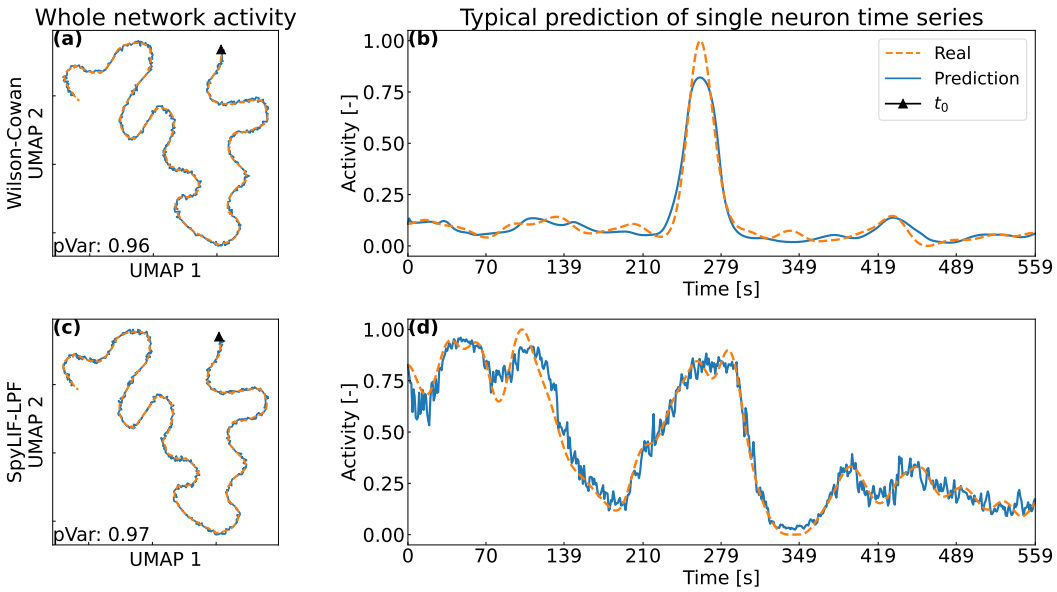

FIGURE 2.4.2 – Comparison of neuronal activity measurements and corresponding predictions obtained using NeuroTorch. The real data describe the time evolution of single-neuron activity in the ventral habenula of a larval zebrafish displayed in Figure 2.4.1, Gaussian-smoothed with $\sigma = 10$ along the time axis. The metrics used are respectively the macro pVar (equation 2.6.18) and the average and standard deviation of the micro pVar (equation 2.6.19). **(a)** All smoothed original time series and Wilson-Cowan predictions projected as trajectories in a 2D UMAP space [78], confirming the overall good performance, where the triangle denotes the initial time step. This time course prediction has a macro pVar of 0.96 with a micro pVar of 0.95 ± 0.03. **(b)** Real series vs. typical prediction obtained by NeuroTorch on the dataset. **(c and d)** Corresponding results obtained with a SpyLIF-LPF dynamics. The whole time course prediction has a macro pVar of 0.97 with a micro pVar of 0.96 ± 0.01.

cal connection ablations, where the connection with the smallest weight (in absolute value) is successively removed ; (2) random connection ablations, where at each step, a randomly selected connection is removed, disregarding its weight. To assess the resilience of the models under connection removal, we computed the pVar each time a connection was removed and then compared it to the original pVar to obtain a performance ratio.

Figure 2.4.3 summarizes the results of this analysis. Panel **(a)** shows that in the case of hierarchical connection ablations, the performance-ratio curve for WC+Dale initially reaches a sparsity of approximately 0.2 without significant change before rapidly decreasing. In contrast, the corresponding curve for SpyLIF-LPF+Dale reaches a higher sparsity level of around 0.4 before experiencing a significant decline. These findings suggest that the SpyLIF-LPF dynamics tolerates a higher level of sparsity compared to the WC dynamics. Moving on to Panel **(b)**, which focuses on random connection ablations, all performance-ratio curves demonstrate a rapid decline with respect to sparsity. As for the hierarchical connection ablation, the WC+Dale curve remains higher than the WC curve, suggesting potential advantages of incor-

porating Dale's law. Interestingly, the SpyLIF-LPF curve not subjected to Dale's law shows a slower initial decline than the corresponding curve subjected to Dale's law, but the extent to which Dale's law contributes to this particular dynamics in the context of random connection ablations remains uncertain. Finally, Panel **(c)** shows the Area Under the Curve (AUC), which measures the average performance ratio over sparsity, for each curve in Panel **(a)**, while Panel **(d)** presents the AUC for each curve in Panel **(b)**. Analyzing Panel **(c)**, it is clear that Dale's law enhances the resilience of network models based on both types of dynamics against hierarchical ablation. However, Panel **(d)** indicates that the distributions of AUCs for the SpyLIF-LPF dynamics are not statistically significantly different, making it challenging to draw definitive conclusions. Conversely, the distributions of AUCs for the WC dynamics in Panel **(d)** show that Dale's law contributes to their overall resilience against this type of ablation. Therefore, based on these observations, we conclude that Dale's law and the spiking dynamics have a positive impact on the resilience of network models trained on experimental data, especially in the context of hierarchical connection ablation.

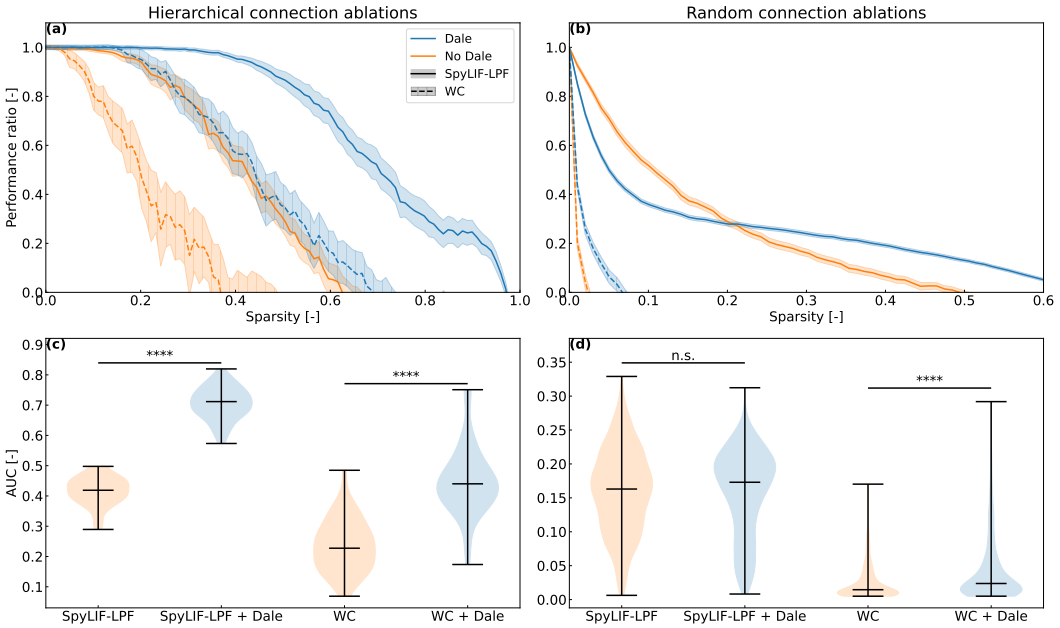

FIGURE 2.4.3 – Resilience analysis of the network models based on the SpyLIF-LPF and the WC dynamics, with and without the enforced Dale law, all trained with e-prop. **(a)** Hierarchical connection ablations : at each step, the connection with the smallest weight in absolute value is removed from the network. **(b)** Random connection ablations : random connections are successively removed from the network. Here 32 random seeds by model are used to choose the ablated weights which produces a distribution of performance ratios for each sparsity value. Each solid or dashed line follows the mean of the corresponding distribution while each shaded region around a line represents a 95% confidence interval. **(c, d)** Area under the curve (AUC) for each curve in (a) and (b) respectively, where **** indicates that the p-value is less than 10^{-4} while n.s. stands for non statistically significant.

2.5 Discussion

The accuracy results of 96.37%, 85.58% and 74.16% shown in table 2.4.1 for all the benchmark datasets respectively demonstrate that the training pipeline and the dynamics implemented in NeuroTorch work as intended. Indeed, these results are superior to SpyTorch and Norse for image classification and slightly inferior for audio classification. It should be noted that it is possible to obtain better results than those presented by increasing the number of neurons used or by changing the dynamics used. Similarly for the SpyTorch and Norse packages, however, this configuration has been used in order to compare the different packages fairly.

In 2020, Bellec & al introduced the eligibility-propagation algorithm (e-prop), a new learning algorithm that is biologically plausible, online and approaches the performance of backpropagation through time (BPTT). However, one of the constraints of the different implementations of this algorithm so far has been that the computation of the eligibility trace is hard-coded, meaning that one needs to apply the derivative of the output of the layer with respect to each parameters such as described by equation 2.6.14. In other words, the derivative of the output z_ℓ^t with regards to each parameters must be computed analytically and coded manually which can be a tedious and even daunting for newcomers. NeuroTorch incorporates a simple and elegant solution to this shortcoming. Neurotorch computes the eligibility trace by exploiting the computation power of the python module PyTorch. In our package, we apply PyTorch's automatic differentiation engine to compute the eligibility trace for *any* dynamic without the need for an analytical derivation and a manual implementation. Hence, a NeuroTorch's user can, in a few simple lines of code, add his own dynamic and apply the e-prop algorithm right away. Our implementation accelerates and significantly simplifies the use of the e-prop algorithm.

The e-prop algorithm is well suited for the reproduction of time series. Indeed, the results presented in figure 2.4.2 reveal that NeuroTorch's implementation of e-prop can optimize a model that reproduces with great precision and fidelity the neuronal activity both on a local and global scale. The trajectory of the predicted time series in the UMAP space fits well the prediction of the real time series. The match between both the predicted and real trajectory in UMAP shows that, *in general*, the time series is well reproduced. This is also confirmed by the precision of our prediction. A relative precision of 3.2% and 1.0% is achieved for the prediction with the Wilson-Cowan and SNN dynamics respectively. The small uncertainty means that most neurons are well fitted. This conclusion is important because a high macro pVar can be achieved if the trainer fits well the neuron with poor activity and fit badly the neuron that are active. This is especially true for a dataset where only a few neurons are highly active. Hence, the use of the pVar (instead of the MSE) allows us to not only fit the mostly inactive neurons, but also to fit those who are highly active. Furthermore, the high quality of the prediction for each individual neuron can be confirmed by comparing the heatmaps of the predicted time series with the real heatmap. In fact, the prediction of a typical neuron is shown in the figure 2.4.2 where the high micro pVar indicates that the variance is preserved.

With the capability to construct computational models using e-prop, we can now delve into their analysis to derive conclusions. One such application is the evaluation of the model's resilience. Understanding the brain's resilience is still an ongoing endeavor. Hence, it is essential to provide an easy and elegant way to measure this metric in different contexts. For instance, we know that the sparsity in the brain is extremely low at around one of 10 millions [122]. Hence, a computational model of a neuronal region allows us to test its resilience as a function of the sparsity. In NeuroTorch, it is possible to measure the resilience of a model that was trained with different parameters. Such computational liberty opens up a realm of possibilities for analyzing and improving our understanding of resilience. For instance, the results in the figure 2.4.3 reveal that the performance of the network can be maintained for a higher sparsity when the model are trained using dale's law and when using a hierarchical connection ablation. The results presented in the figure 2.4.3 suggest that Dale's law improve the resilience of networks to hierarchical ablation using spiking or Wilson-Cowan dynamics.

The study presented in this paper demonstrates that NeuroTorch performs similarly to other existing packages when tested on benchmark datasets. The results in Table 2.4.1 indicate high accuracies for image classification tasks, slightly inferior for audio classification. It is worth noting that the package's performance could potentially be improved by adjusting parameters such as the number of neurons or the dynamics used. The implementation of Bellec's algorithm in NeuroTorch simplifies the computation of the eligibility trace, making it more accessible and efficient for users. The precise reproduction of neuronal activity patterns in the ventral habenula of a larval zebrafish further demonstrates the effectiveness of NeuroTorch's e-prop implementation to analyse neuronal activity data. The analysis of resilience in relation to sparsity and hierarchical ablation suggests that Dale's law enhances the resilience of spiking and Wilson-Cowan dynamics, as evidenced by the observed performance curves and distributions. These findings contribute to our understanding of the impact of Dale's law on network resilience in different dynamics and ablation scenarios. Finally, the study highlights the need for further analysis of the resulting connectome structures, the inclusion of biologically plausible constraints on connectome models, and the consideration of other biologically relevant parameters.

2.6 Methods

2.6.1 Neuronal network models

To perform data prediction using artificial intelligence, specific network architecture, neuronal dynamics and learning algorithms must be specified. The dynamics determine how artificial neurons communicate and process information in the network, while learning algorithms update network parameters, including connection weights, to minimize a loss function. In this section, we present the dynamics and the learning algorithms used in this paper for valida-

tion, including the prediction of experimental neuronal activity. Additional methods are also available in NeuroTorch [42] ; please refer to the library's documentation for a comprehensive list of implemented modules.

Network architecture

Each network contains a fixed number N of units, here interpreted as neurons. The network is moreover divided into L sub-networks, called layers. The ℓ-th layer contains N_ℓ neurons, where $\ell \in \{1, \dots, L\}$. Hence, $N_1 + N_2 + \dots + N_L = N$. The set of connections going from layer ℓ to layer m is encoded into a $N_\ell \times N_m$ matrix $W^{\ell m}$, whose (i, j) element is a real number that gives the weight of the connection going from i-th neuron in layer ℓ to the j-neuron in layer m, denoted as $i \to j$. By convention, there is :

— an inhibitory connection from i to j if $W_{ij}^{\ell m} < 0$,
— no connection at all from i to j if $W_{ij}^{\ell m} = 0$,
— excitatory connection from i to j if $W_{ij}^{\ell m} > 0$.

Inside layer ℓ, the neurons may be connected among themselves, forming a recurrent network. The latter is described by the matrix $W^{\text{rec}} = W^{\ell\ell}$, with the element W_{ij}^{rec} corresponding to the weight of the connection from i to j, where $i, j \in \{1, 2, \dots, N_\ell\}$. When focusing on the connections arriving at layer ℓ and coming from layer k, we write W_{ij}^{in} to denote the weight of the connection coming from neuron i in layer k to neuron j in layer ℓ.

Neuronal dynamics

Leaky-integrate-and-fire

The leaky-integrate-and-fire (LIF) is a classical model in neuroscience [123] that describes a neuron's activity in terms of its membrane potential and incoming synaptic currents. Although the shape of the membrane potential is not realistic [14], the model includes essential features of neuronal activity like : integration of the synaptic inputs, threshold-based firing, resetting of membrane potential after firing, and decay of the membrane potential over time. In its networked and time-discretized version [17, 29], the LIF model is determined by the recursion equation

$$V_j^{t+\Delta t} = \left(\alpha V_j^t + \sum_{i=1}^{N} z_i^t W_{ij}^{\text{rec}} + \sum_{i=1}^{M} x_i^t W_{ij}^{\text{in}} \right) \left(1 - z_j^t\right), \tag{2.6.1}$$

where N designates the number of neurons of the layer, V_j^t denotes the membrane potential of neuron j at time step t, Δt denotes the time step of the Euler integration, x_i^t denotes the input coming from neuron i at time t in another layer of size M. In the eventuality that x_i^t is a function of $z^{t_i - 1}$, then W_{in} is considered recurrent. Moreover, the parameter α is a decay constant that can be written as $\alpha = \exp\left(-\Delta t / \tau_m\right)$, with τ_m being the decay time constant of the membrane potential, which is usually 20 ms. Finally, z_j^t denotes the output of neuron j at

time t, which is defined as

$$z_j^t = H(V_j^t - V_{\text{th}}), \tag{2.6.2}$$

where V_{th} denotes the activation threshold of the neuron and H is the Heaviside step function satisfying $H(x) = 1$ when $x \geq 0$ and $H(x) = 0$ otherwise.

Leaky-integrate with explicit synaptic current

We call leaky-integrate with explicit synaptic current (SpyLI), a dynamics inspired by Neftci's SNN equations [29] that integrates the input of the layer without generating spikes. Although rarely used as an input or hidden layer in SNNs, it is commonly employed at the network's output due to its continuous activity. SpyLI is represented by two differential equations for the membrane potential and synaptic current, serving as a more complex variant of the LI dynamic introduced by Bellec et al. [17]. Using Euler integration, the synaptic current is updated by the recursion equation

$$I_{\text{syn},j}^{t+\Delta t} = \alpha I_{\text{syn},j}^t + \sum_{i=1}^{M} x_i^t W_{ij}^{\text{in}}, \tag{2.6.3}$$

while the the synaptic potential is updated by

$$V_j^{t+\Delta t} = \beta V_j^t + I_{\text{syn},j}^{t+\Delta t} + b_j \tag{2.6.4}$$

where $I_{\text{syn},j}^t$ denotes the synaptic current of neuron j at time step t, $\alpha = e^{-\Delta t/\tau_{\text{syn}}}$ is the synaptic decay constant, $\beta = e^{-\Delta t/\tau_{\text{mem}}}$ is the membrane decay constant, and b_j the bias weight of the layer associated with neuron j.

Leaky-integrate-and-fire with explicit synaptic current

The leaky-integrate-and-fire with explicit synaptic current (SpyLIF) dynamic inspired by Neftci's SNN equations [29] is a more complex variant of the LIF dynamics in Eq. (2.6.1) that contains two differential equations like the previously introduced SpyLI dynamics. The equation

$$I_{\text{syn},j}^{t+\Delta t} = \alpha I_{\text{syn},j}^t + \sum_{i=1}^{N} z_i^t W_{ij}^{\text{rec}} + \sum_{i=1}^{M} x_i^t W_{ij}^{\text{in}} \tag{2.6.5}$$

updates of the synaptic current with Euler integration while the equation

$$V_j^{t+\Delta t} = \left(\beta V_j^t + I_{\text{syn},j}^{t+\Delta t}\right)\left(1 - z_j^t\right) \tag{2.6.6}$$

updates of the synaptic potential. As for the LIF dynamics, the output of neuron j at time t, denoted z_j^t, is defined in Eq. (2.6.2).

Spiking neural network - low pass filter

In some applications, such as the e-prop learning algorithm (see below), a smooth version of the spiking dynamics is required. The smoothing is achieved by the low pass filter (LPF) used by Bellec. et al [17, 124], which modifies the vector x^t of neuron activity at time t as

$$\mathcal{F}_\kappa\big(x^t\big) = \kappa\mathcal{F}_\kappa\big(x^{t-1}\big) + x^t. \tag{2.6.7}$$

where the decay rate κ is set to 0.001 by default in NeuroTorch. However, this decay rate can be changed to zero if no filter is required. Every model of the spiking dynamics presented in this paper includes an LPF version in which the output spikes of the neural network are smoothed.

Wilson-Cowan

Neurobiologists often measure calcium activity as an approximation or as a proxy of neuronal activity. By its continuous nature, the Wilson-Cowan (WC) dynamics [15, 47-50] is well-suited to model this type of signal. It describes the time evolution of spiking rates or activation probabilities of neuronal units (either neuron populations or single cells) in terms of their interactions occurring along the network connections. In this paper, we use the time-discretized form of the Wilson-Cowan model :

$$y_j^{t+\Delta t} = y_j^t\left(1 - \frac{\Delta t}{\tau}\right) + \frac{\Delta t}{\tau}\big(1 - r_j y_j^t\big)\sigma\left(\sum_{i=1}^{M} x_i^t W_{ij}^{\text{in}} + \sum_{i=1}^{N} y_i^t W_{ij}^{\text{rec}} - \mu_j\right), \tag{2.6.8}$$

where y_i^t denotes the activity of unit j at time t, τ designates the rate of decrease of the activity, r_j is the transition rate of unit j. This parameter represents the time a unit needs to "recover" before becoming "sensitive" to activation. The symbol σ stands for the sigmoid function while μ_j designates the activation threshold of unit j.

Learning algorithm — Eligibility trace forward propagation

The eligibility trace forward propagation (e-prop) learning algorithm [17, 124] performs weight updates using only local information, meaning that a connection's weight at time $t+\Delta t$ changes as a function of the activity of the adjacent neurons at time t, without considering the activity of other neurons in the network. This algorithm is biologically plausible since the rules that regulate the changes of synaptic connections of biological neurons only involve information that is local in time and space [125].

The implementation of e-prop in NeuroTorch offers flexibility as it can be applied to all types of layers. Additionally, the level of locality in the implementation can be adjusted according to the user's choice. Initially, lets consider a network with L layers containing the parameters $\hat{\Theta}_{\ell,k}$ where ℓ is the index of the layer and k is the index of the parameter in this layer. The layer indexes $\ell \in [0, L-1]$ begin at zero which is the index of the input layer. Each parameter can be

a scalar, a vector, a matrix or a tensor. In the following equations, the indices t indicates the current time step. The input and output tensors of the network denotes respectively x and $\hat{y}$ while the desired output tensor y. For the layer-specific inputs and outputs of the model, these are denoted by the quantity z with the appropriate subscript ℓ if necessary. The prediction error is thus defined as

$$\varepsilon^t = y^t - \hat{y}^t \tag{2.6.9}$$

where the symbols $\hat{y}$ and y are of size (B, f_{out}), with B being the number of data in the current minibatch (a subset of the dataset) and f_{out} the number of output values. Therefore the quantity ε is of size (B, f_{out}). Figure 2.6.1 provides a simplified representation of the e-prop algorithm as implemented in NeuroTorch, while Table 2.6.1 summarizes the variables utilized in the algorithm. The loss of the network is determined by the user-defined function

$$\epsilon^t = \mathcal{L}(y^t, \hat{y}^t), \tag{2.6.10}$$

where ϵ^t is a scalar. Then, the main goal of the current algorithm is to minimize

$$\epsilon = \sum_t \epsilon^t, \tag{2.6.11}$$

which represents the overall loss over time. To achieve this in an online and biological way, the network needs to be updated with local information. For more clarity, we divide the update of the network parameters into two steps : the update of the parameters of the output layer of the network and the update of the other parameters.

Output parameters : The update of the output parameters is similar to an update by TBPTT [20]. So the gradient of these parameters will be determined by the recursive equation

$$\nabla\hat{\Theta}_{L-1,k} = \sum_{t=t'}^{t'+\tau} \mathcal{F}_\kappa\left(\frac{\partial \epsilon^t}{\partial \hat{\Theta}_{L-1,k}}\right) \tag{2.6.12}$$

where the quantity τ is the number of time steps that the gradient is accumulated. To be more local in time and also more biologically accurate, τ must be as close to one as possible. However, contrary to Bellec's work [17], in which τ is equal to one, NeuroTorch allows the user to determine it's own τ. Hence, τ is now simply an hyper-parameter that can be adjusted by the user. In addition, the low pass filter $\mathcal{F}_\kappa$ mentioned in Equation 2.6.12 is defined in Eq. (2.6.7).

Hidden parameters : The update of synaptic weights according to e-prop is done by approximating the gradient of the backpropagation throught time (BPTT) [17, 62] as the set of partial derivatives of the user defined prediction error E,

$$\frac{\partial E}{\partial \hat{\Theta}_\ell} = \sum_t L_\ell^t \odot e_\ell^t \tag{2.6.13}$$

where L_ℓ is the learning signal and e_ℓ is the eligible trace. Here, the operator $\odot$ is defined as the element-wise product (a.k.a. Hadamard product). This eligible trace is described by the equation

$$e^t_{\ell,k} = \frac{\partial z^t_\ell}{\partial \hat{\Theta}_{\ell,k}} \tag{2.6.14}$$

which can be simply interpreted as the partial derivative of the local output signal. The learning signal defined as

$$L^t_{\ell,k} = \overline{\varepsilon^t}\ R_{\ell,k} \tag{2.6.15}$$

is interpreted as the error feedback from the network output with $R_{\ell,k}$ as a feedback matrix that can be initialized in several ways described in [17, 126] with the nomenclature B_{jk}. To maintain the locality of the algorithm, $R_{\ell,k}$ is often initialized as a random matrix to broadcast a random feedback to the hidden parameters which is the case in this work. Using these last expressions, the gradient of the weights $\hat{\Theta}_{\ell,k}$ is equal to

$$\nabla \hat{\Theta}_{\ell,k} = \sum_t L^t_{\ell,k} \mathcal{F}_\gamma(e^t_{\ell,k}) \tag{2.6.16}$$

which is used in a gradient descent method to optimize the weights.

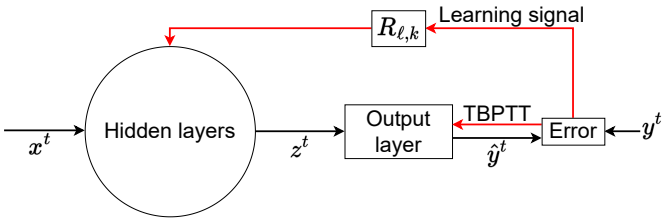

FIGURE 2.6.1 – Representation of the eligibility-propagation algorithm. The hidden layers contain the desired dynamics while the output layer is generally a fully-connected linear layer. The parameters of the output layers are trained with TBPTT while those of the hidden layer are trained by computing a learning signal through a random matrix specific for each parameters.

In the NeuroTorch implementation of the e-prop algorithm, it is possible to clip some variables to stabilize the training. Among these variables, there are the gradients of the parameters, the eligibility trace, the learning signal and the feedback weights.

Application of Dale's law

To ensure biological realism in predicting neuronal time series, NeuroTorch implements the Dale law. By enforcing this well-known law discovered by Sir Henry Dale in 1935, which states that a neuron can release only one type of neurotransmitter for all its synapses [127], NeuroTorch restricts the solution space and converges towards a biologically plausible neuronal connectome. This is achieved by decomposing the connectivity matrix into a sign vector S and a weight matrix, with the sign vector representing the excitatory or inhibitory property of each

Variable	Size	Description
x	(B, f_{in})	Input of the network.
$\hat{y}$	(B, f_{out})	Output of the network.
y	(B, f_{out})	Desired output of the network.
ε	(B, f_{out})	Prediction error.
z_ℓ	$(B, N_{\ell,out})$	Output of the layer ℓ.
$\hat{\Theta}_{\ell,k}$	$(N_{\ell,k,in}, N_{\ell,k,out})$	Parameter k of the layer ℓ.
$R_{\ell,k}$	$(f_{out}, N_{\ell,k,out})$	Feedback matrix of the parameter k of the layer ℓ.
$L_{\ell,k}$	$(1, N_{\ell,k,out})$	Learning signal of the parameter k of the layer ℓ.
$e_{\ell,k}$	$(N_{\ell,k,in}, N_{\ell,k,out})$	Eligible trace of the parameter k of the layer ℓ.

Table 2.6.1 – Summary of the variables of the e-prop algorithm.

neuron and the weight matrix squared to ensure positive weights. Both the vector and the square root of the weight matrix are trained by the learning algorithms, as shown the equation

$$w_{ij} = \prod_{i,j} s_i w_{\text{root},ij}^2 \tag{2.6.17}$$

where $\mathbf{W}_{\text{root}}$ is the square root element-wise of the weight matrix.

Evaluation metrics

The proportion of variance metric (pVar) [3] is a loss function used to train neural networks on times series and to evaluate their effectiveness. This metric is bound between $[-\infty, 1]$ where 1 is the optimal value. The pVar is essentially a mean square error (MSE) that is weighted by the variance of the timeseries. By doing so, we ensure that the timeseries with higher variance are well trained rather than just being averaged by the prediction. There are two variants of the metric, one is the macro pVar that evaluates the overall performance over all variables of the times series and the other is the micro pVar that evaluates the individual matches of the variables.

Let $X_{t,i}$ be the predicted time series of variable i (neuron or unit) at time step t and $Y_{t,i}$ the corresponding target time series. The macro P-Variance is describe by the equation

$$\mathcal{L}^{\text{macro}}(X, Y) = 1 - \frac{\text{MSE}(X, Y)}{\text{Var}(Y)} \tag{2.6.18}$$

and the micro P-Variance is describe by the equation

$$\mathcal{L}_i^{\text{micro}}(X, Y) = 1 - \frac{\text{MSE}(X_i, Y_i)}{\text{Var}(Y_i)}. \tag{2.6.19}$$

Where the macro version is a scalar, the micro one is a vector giving the P-Variance for all variables in the times series. In this case, it's useful to show the micro pVar in form of it's average ($\mathbb{E}\big[\mathcal{L}_i^{\text{micro}}\big]$) and standard deviation ($\text{std}\big[\mathcal{L}_i^{\text{micro}}\big]$) statistics.

2.6.2 Numerical experiments with NeuroTorch

Performance validation against benchmark datasets

The NeuroTorch performances were compared to the packages SpyTorch and Norse onto three benchmark datasets : MNIST, Fashion-MNIST and Heidelberg.

To obtain the prediction accuracies with the SpyTorch network [29], the authors' code was run with minimal modifications. For the MNIST dataset, the script "SpyTorchTutorial3.ipynb" was used, in which the dataset name was switched from Fashion-MNIST to MNIST, the regularization loss was removed, and the activation threshold for converting pixels to spikes was adjusted to 0.1. The other parameters remained unchanged from the version 0.4 of the repository. For the Fashion-MNIST dataset, the same script was run without any changes.

To generate results for the Norse library [28], the script "mnist_classifiers.ipynb" from the "notebooks" subrepository was used for the MNIST and Fashion-MNIST datasets. The script came from the commit with the id "bc627fc" and was slightly modified to allow comparison with SpyTorch results. The hyperparameters used were an integration time step of $T = 100$, a learning rate of $lr = 2e - 4$, a hidden size of $nb\_hidden = 100$, and a number of epochs of $epochs = 30$. The script generated results with different input encoders, but only the results from the "ConstantCurrentLIFEncoder" were used due to its higher accuracy compared to the other encoders.

The results for the Heidelberg dataset were obtained using the script "SpyTorchTutorial4.ipynb" from the version 0.4 of the SpyTorch repository [29]. To produce results for the Norse package [28] (version 1.0.0), the model from the Mnist experiment was used, but without the encoder and with a hidden size of 200 units.

To generate the prediction accuracy results for the Heidelberg, Fashion-MNIST, and MNIST datasets with NeuroTorch, the scripts "spike_trace_classification /gen_heidelberg_results.py", "images_classification /gen_fashion_mnist_results.py" and "images_classification /gen_mnist_results.py" of the sub-repository "NeuroTorch_PaperWithCode" were used respectively. To ensure a fair comparison, the hyperparameters were set to match those used with the other packages. The models were trained using the Backpropagation Through Time (BPTT) learning algorithm in NeuroTorch.

Experimental acquisition of a neuronal activity dataset

A neuronal activity dataset of 522 neurons from the ventral habenula of a larval zebrafish was collected. The neuronal activity data were acquired by two-photon imaging on 6 dpf larval zebrafish expressing a pan-neuronal genetically-encoded calcium indicator GCaMP6s. Using a resonant scanner and piezo-driven objective, the neurons of head-restrained larvae in agarose were recorded with a darkflash [128] stimuli.

Prediction validation with neuronal activity datasets

To demonstrate that the package can be used with time series we attempted reproducing the neuronal activity from the experimental dataset collected in the ventral habenula region of a zebrafish. Two networks were employed for comparison : one based on continuous dynamics (Wilson-Cowan) and the other on spiking dynamics (Leaky integrate and fire). The training utilized the e-prop learning algorithm to showcase its performance and flexibility. The model consisted of two layers : the input layer employing either Wilson-Cowan or SpyLIF-LPF dynamics, and the output layer comprising a linear layer with sigmoid activation. Specifically, for the SpyLIF-LPF dynamics, only the forward weights W_{ij}^{in} were optimized during training, while for the Wilson-Cowan dynamics, the forward weights, as well as the τ, μ, and r parameters, were optimized. If the Dale's law is enforced in the model, the parameters W_{ij}^{in} that are optimized are replaced by $\vec{S}$ and $\mathbf{W}_{\text{root}}$ as explained in 2.6.1. The experiments focused on a single hidden layer per model, omitting the use of the recurrent weights W_{ij}^{rec}. Since the model operates recurrently, meaning it uses its own predictions as inputs, the weight matrix W_{ij}^{in} is considered to be recurrent. The script to reproduce the training of these networks is "neuronal_time_series_reproduction/main.py" which can be found under the sub-repository "NeuroTorch_PaperWithCode". To create trajectories using UMAP [78, 116], a powerful dimensionality reduction technique, we first applied the UMAP algorithm to the target time series, projecting them into 2 dimensions, whose coordinates were denoted UMAP 1 and UMAP 2. Afterwards, we utilized the trained UMAP to perform the same transformations on the predicted time series, effectively mapping them onto UMAP 1 and UMAP 2.

2.6.3 Resilience computational experiments

The resilience of a neural network, defined as its ability to accurately predict neuronal activity even when connections are removed, was measured by evaluating its performance as a function of its sparsity. The performance was quantified as the ratio between the loss function after alteration of connectivity and the original loss function, with a performance of 1 indicating normal behaviour post-alteration. Sparsity, denoted as $\mathcal{S} = (\sum_{i,j} \delta_{W_{ij},0})/(\sum_{i,j} 1)$, represents the proportion of unconnected pairs of neurons in the network. The resilience of trained neuronal network models was assessed using e-prop learning, with 50 trainings conducted for both SpyLIF-LPF and Wilson-Cowan dynamics, with and without the enforcement of the Dale law. Two types of ablation were performed : hierarchical connection ablation, where connections with the smallest impact on the network (i.e., smallest absolute weight values) were sequentially removed from the network setting their weight to zero in matrix W^{in} ; random connection ablation, where connections were randomly removed from the network by random selected elements in W^{in} and set their value to zero. Each ablation method was tested with 100 levels of sparsity, considering only the layer with the specific dynamics for ablation while keeping the output layer weights unaffected. Random ablation tests were conducted with 32

different seeds to ensure robustness of the results.

2.6.4 Design and implementation of NeuroTorch

NeuroTorch is built with a modular design and a pipeline structure. Several tutorials are available with NeuroTorch to learn how to use the package. For advanced users who want to fully exploit the pipeline features, here is an overview of its architectural characteristics.

One of the fundamental components of NeuroTorch is the sequential model, which serves as a versatile framework for sequencing a set of learning layers. As illustrated in Figure 2.6.2, this model adopts a "H" shape, allowing a network generalization that accommodates multiple inputs and outputs. To ensure flexibility and to remove any limitations on input and output types, NeuroTorch associates a layer and a transform to each input and output of the network. As a result of this design, the user is able to choose the number and type of layers, which can come from NeuroTorch, Norse, or any other module built with PyTorch. Since the transforms are also PyTorch modules, they can be selected by the user as well.

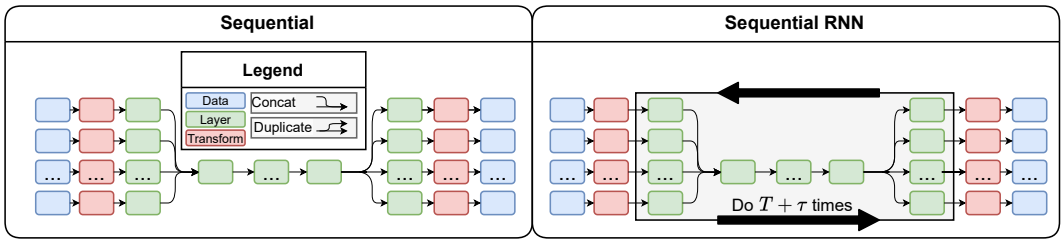

FIGURE 2.6.2 – Representation of the sequential model and the recursive sequential model. T is defined as the time associated to the input as initial conditions. τ is the forecasting time used for the prediction.

Unlike other machine learning pipeline in which most of the modules are fixed, the NeuroTorch training pipeline offers flexibility and dynamic adaptability. This enables the user to modify the pipeline objects during the training process, which can, among other things, involve altering the dataset to make it progressively more challenging as the model learns. Furthermore, users have the flexibility to establish a general training configuration for a collection of experiments. By incorporating a single callback that is specific to each experiment, they can easily adapt the training process to meet the specific requirements of that particular experiment. This allows for a streamlined approach in modifying the training setup and ensuring it aligns with the unique demands of each experiment. This feature of the library relies on five objects of its implementation. First, the `trainer`, which is responsible for controlling the execution flow of the learning phase by providing the data and stopping the learning. The trainer updates the different variables of its `state` based on training results (e.g. the current iteration, the current loss value, etc). It also calls `callbacks` at appropriate times. A callback is a function that can perform multiple actions to facilitate training such as updating the datasets, the

model or simply calculate metrics on the training process. The flexibility of the pipeline relies on the core of the algorithm : the learning or optimization process, which is performed by a special callback called the `learning algorithm`. This object updates the weights of the model according to a specified method (e.g. e-prop or BPTT). It also implements important events, such as the `on_optimization_begin` event that performs a round of optimization of the model parameters, and utilizes data provided by the user and managed by the `dataloader` to update the `model`'s weights. During the execution of a training, callbacks have the ability to modify the trainer's state and behavior at runtime, such as setting a `stop_training_flag` or adjusting `n_iterations` for early stopping. The training loop with the NeuroTorch pipeline is depicted in Figure 2.6.3.

The following definitions are necessary to understand the description of the execution of the main algorithm during the training phases. A *pass* corresponds to both the prediction of data by the model and the update of its weights. An *iteration* covers a full pass through both the training and validation datasets. It corresponds to the outer loop in Figure 2.6.3. An *epoch* represents a complete pass through either dataset, it corresponds to the inner loop in Figure 2.6.3. A *batch* signifies a forward pass through the network, and *training* and *validation* encompass full passes through their respective datasets. Just to clarify, an iteration can consist of multiple epochs (depending on the user's choice), and each epoch can consist of multiple batches depending on the amount of data and the user's choice of hyperparameters. After an iteration, a checkpoint state containing all the relevant information on the state of the execution of the pipeline in this instant can be saved. The user can set a callback to trigger the creation of a checkpoint state or determine a backup frequency.

The part of the algorithm executing callback events can be summarized as follows : the learning process is initiated by loading a checkpoint state and this is followed by a loop executing a specified number of iterations. Within each iteration, events occur for training and validation, including their beginnings and endings. Iterations consist of multiple epochs, which, in turn, comprise multiple batches. Each batch involves optimization-related events. The process concludes with the closing step.

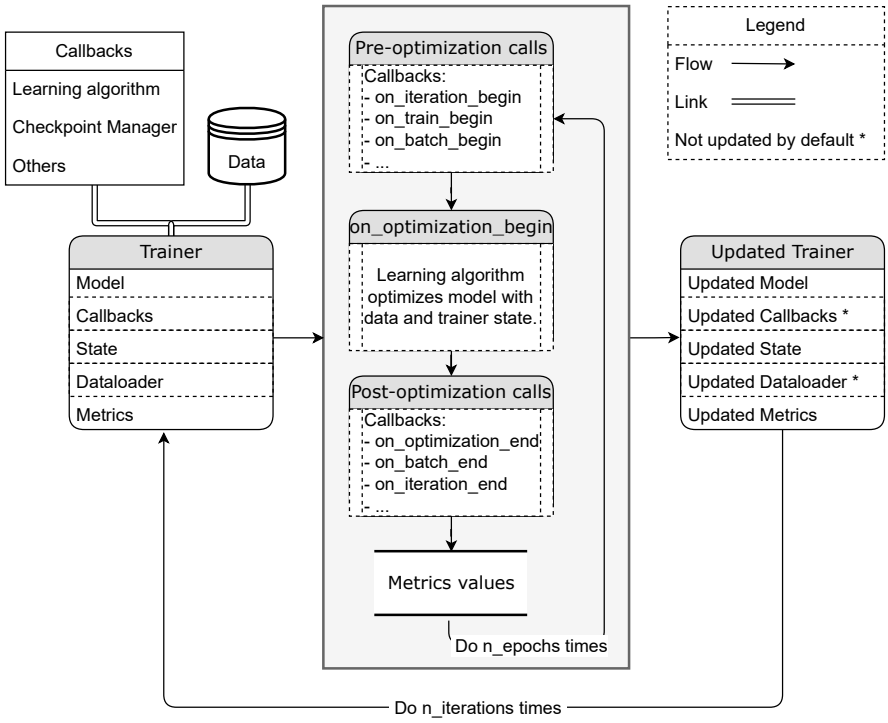

FIGURE 2.6.3 – Representation of the training pipeline. The callbacks and the data provided by the user are processed using the trainer. The trainer then calls the events of the callbacks during the training. The trainer is updated accordingly.

References

[3] *Inferring brain-wide interactions using data-constrained recurrent neural network models*, 2021.

[8] M. Bahdine, "Simulation et inférence de réseaux de neurones à l'aide d'intelligence artificielle", mém. de mast. (Université Laval, 2020).

[10] L. E. Suárez, B. A. Richards, G. Lajoie et B. Misic, "Learning function from structure in neuromorphic networks", Nature Machine Intelligence **3**, 771-786 (2021).

[11] F. Damicelli, C. C. Hilgetag et A. Goulas, "Brain connectivity meets reservoir computing", PLOS Computational Biology **18**, e1010639 (2022).

[14] E. M. Izhikevich, *Dynamical systems in neuroscience* (MIT Press, 2007), 522 p.

[15] B. Ermentrout et D. H. Terman, *Mathematical foundations of neuroscience*, t. 35 (Springer, 2010).

[16] T. Iakymchuk, A. Rosado-Muñoz, J. F. Guerrero-Martinez, M. Bataller-Mompeán et J. V. Francés-Villora, "Simplified spiking neural network architecture and STDP learning algorithm applied to image classification", EURASIP Journal on Image and Video Processing **2015**, 1-11 (2015).

[17] G. Bellec, F. Scherr, A. Subramoney, E. Hajek, D. Salaj, R. Legenstein et W. Maass, "A solution to the learning dilemma for recurrent networks of spiking neurons", Nature Communications **11**, 3625 (2020).

[18] T. P. Lillicrap et A. Santoro, "Backpropagation through time and the brain", Current opinion in neurobiology **55**, 82-89 (2019).

[19] G.-q. Bi et M.-m. Poo, "Synaptic modifications in cultured hippocampal neurons : dependence on spike timing, synaptic strength, and postsynaptic cell type", Journal of neuroscience **18**, 10464-10472 (1998).

[20] I. Goodfellow, Y. Bengio et A. Courville, *Deep learning* (MIT Press, 2016), 801 p.

[22] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang et H. Wang, "Loihi : A Neuromorphic Manycore Processor with On-Chip Learning", IEEE Micro **38**, 82-99 (2018).

[23] S. B. Furber, F. Galluppi, S. Temple et L. A. Plana, "The spinnaker project", Proceedings of the IEEE **102**, 652-665 (2014).

[24] T. DeWolf, "Spiking neural networks take control", Science Robotics **6**, eabk3268 (2021).

[25] G. Van Rossum et F. L. Drake, *Python 3 Reference Manual* (CreateSpace, 2009).

[26] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai et S. Chintala, "PyTorch : An Imperative Style, High-Performance Deep Learning Library", *Advances in Neural Information Processing Systems 32* (Curran Associates, Inc., 2019), p. 8024-8035.

[27] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu et Xiaoqiang Zheng, *TensorFlow : Large-Scale Machine Learning on Heterogeneous Systems*, 2015.

[28] C. Pehle et J. E. Pedersen, *Norse - A deep learning library for spiking neural networks*, version 0.0.7, 2021.

[29] E. O. Neftci, H. Mostafa et F. Zenke, "Surrogate Gradient Learning in Spiking Neural Networks : Bringing the Power of Gradient-Based Optimization to Spiking Neural Networks", IEEE Signal Processing Magazine **36**, 51-63 (2019).

[30] J. K. Eshraghian, M. Ward, E. Neftci, X. Wang, G. Lenz, G. Dwivedi, M. Bennamoun, D. S. Jeong et W. D. Lu, "Training spiking neural networks using lessons from deep learning", arXiv preprint arXiv :2109.12894 (2021).

[31] L. Deng, "The MNIST database of handwritten digit images for machine learning research", IEEE Signal Process. Mag. **29**, 141 (2012).

[32] Y. LeCun, C. Cortes et C. Burges, *MNIST handwritten digit database*, (1998)

[33] B. Cramer, Y. Stradmann, J. Schemmel et F. Zenke, "The Heidelberg Spiking Data Sets for the Systematic Evaluation of Spiking Neural Networks", IEEE Transactions on Neural Networks and Learning Systems, 1-14 (2022).

[34] D. L. Manna, A. Vicente-Sola, P. Kirkland, T. J. Bihl et G. D. Caterina, *Frameworks for SNNs : a Review of Data Science-oriented Software and an Expansion of SpykeTorch*, 2023.

[35] W. Falcon et The PyTorch Lightning team, *PyTorch Lightning*, version 1.4, 2019.

[36] F. Paradis, D. Beauchemin, M. Godbout, M. Alain, N. Garneau, S. Otte, A. Tremblay, M.-A. Bélanger et F. Laviolette, *Poutyne : A Simplified Framework for Deep Learning*, 2020.

[42] J. Gince, *NeuroTorch : A Python library for machine learning and neuroscience*, https://github.com/NeuroTorch, 2022.

[44] M. S. Al-Batah, N. A. Mat Isa, K. Z. Zamli et K. A. Azizli, "Modified recursive least squares algorithm to train the hybrid multilayered perceptron (HMLP) network", Applied Soft Computing **10**, 236-244 (2010).

[45] C. Zhang, Q. Song, H. Zhou, Y. Ou, H. Deng et L. T. Yang, *Revisiting Recursive Least Squares for Training Deep Neural Networks*, 2021.

[46] J. Schulman, F. Wolski, P. Dhariwal, A. Radford et O. Klimov, "Proximal Policy Optimization Algorithms", CoRR **abs/1707.06347** (2017).

[47] H. R. Wilson et J. D. Cowan, "Excitatory and inhibitory interactions in localized populations of model neurons", Biophysical journal **12**, 1-24 (1972).

[48] V. Painchaud, N. Doyon et P. Desrosiers, "Beyond Wilson-Cowan dynamics : oscillations and chaos without inhibition", Biological Cybernetics **116**, 527-543 (2022).

[49] T. P. Vogels, K. Rajan et L. Abbott, "NEURAL NETWORK DYNAMICS", Annual Review of Neuroscience **28**, 357-376 (2005).

[50] S. Grossberg, "Nonlinear neural networks : Principles, mechanisms, and architectures", Neural Networks **1**, 17-61 (1988).

[55] K.-i. Funahashi et Y. Nakamura, "Approximation of dynamical systems by continuous time recurrent neural networks", Neural Networks **6**, 801-806 (1993).

[56] L. Jin, P. N. Nikiforuk et M. M. Gupta, "Approximation of discrete-time state-space trajectories using dynamic recurrent neural networks", IEEE Transactions on Automatic Control **40**, 1266-1270 (1995).

[62] P. Werbos, "Backpropagation through time : what it does and how to do it", Proceedings of the IEEE **78**, 1550-1560 (1990).

[69] J. C. Eccles, "Chemical transmission and Dale's principle", Progress in Brain Research **68**, 3-13 (1986).

[70] P. Strata, R. Harvey *et al.*, "Dale's principle", Brain Research Bulletin **50**, 349-350 (1999).

[78] L. McInnes, J. Healy et J. Melville, "UMAP : Uniform Manifold Approximation and Projection for Dimension Reduction", arXiv preprint arXiv :1802.03426 (2018).

[82] D. Hassabis, D. Kumaran, C. Summerfield et M. Botvinick, "Neuroscience-inspired artificial intelligence", Neuron **95**, 245-258 (2017).

[83] W. S. McCulloch et W. Pitts, "A logical calculus of the ideas immanent in nervous activity", The bulletin of mathematical biophysics **5**, 115-133 (1943).

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

[85] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities.", Proceedings of the national academy of sciences **79**, 2554-2558 (1982).

[86] R. S. Sutton et A. G. Barto, "Toward a modern theory of adaptive networks : expectation and prediction.", Psychological review **88**, 135 (1981).

[87] F. Pereira, T. Mitchell et M. Botvinick, "Machine learning classifiers and fMRI : a tutorial overview", Neuroimage **45**, S199-S209 (2009).

[88] M.-P. Hosseini, A. Hosseini et K. Ahi, "A review on machine learning for EEG signal processing in bioengineering", IEEE reviews in biomedical engineering **14**, 204-218 (2020).

[89] C. Stringer, T. Wang, M. Michaelos et M. Pachitariu, "Cellpose : a generalist algorithm for cellular segmentation", Nature Methods **18**, 100-106 (2021).

[90] F. Zhang, B. Pan, P. Shao, P. Liu, S. Shen, P. Yao, R. X. Xu, A. D. N. Initiative *et al.*, "A single model deep learning approach for Alzheimer's disease diagnosis", Neuroscience **491**, 200-214 (2022).

[91] J. Mei, C. Desrosiers et J. Frasnelli, "Machine learning for the diagnosis of Parkinson's disease : a review of literature", Frontiers in aging neuroscience **13**, 633752 (2021).

[92] J. Oh, B.-L. Oh, K.-U. Lee, J.-H. Chae et K. Yun, "Identifying schizophrenia using structural MRI with a deep learning algorithm", Frontiers in psychiatry **11**, 16 (2020).

[93] D. L. Yamins et J. J. DiCarlo, "Using goal-driven deep learning models to understand sensory cortex", Nature neuroscience **19**, 356-365 (2016).

[94] A. H. Marblestone, G. Wayne et K. P. Kording, "Toward an integration of deep learning and neuroscience", Frontiers in Computational Neuroscience, 94 (2016).

[95] A. M. Zador, "A critique of pure learning and what artificial neural networks can learn from animal brains", Nature communications **10**, 3770 (2019).

[96] M. P. Van den Heuvel, E. T. Bullmore et O. Sporns, "Comparative connectomics", Trends in cognitive sciences **20**, 345-361 (2016).

[97] R. F. Betzel et D. S. Bassett, "Specificity and robustness of long-distance connections in weighted, interareal connectomes", Proceedings of the National Academy of Sciences **115**, E4880-E4889 (2018).

[98] S. Vyas, M. D. Golub, D. Sussillo et K. V. Shenoy, "Computation through neural population dynamics", Annual Review of Neuroscience **43**, 249-275 (2020).

[99] D. Sussillo et L. F. Abbott, "Generating coherent patterns of activity from chaotic neural networks", Neuron **63**, 544-557 (2009).

[100] B. DePasquale, C. J. Cueva, K. Rajan, G. S. Escola et L. Abbott, "full-FORCE : A target-based method for training recurrent networks", PloS one **13**, e0191527 (2018).

[101] T. Miconi, "Biologically plausible learning in recurrent neural networks reproduces neural dynamics observed during cognitive tasks", eLife **6**, e20899 (2017).

[102] A. S. Andalman, V. M. Burns, M. Lovett-Barron, M. Broxton, B. Poole, S. J. Yang, L. Grosenick, T. N. Lerner, R. Chen, T. Benster *et al.*, "Neuronal dynamics regulating brain and behavioral state transitions", Cell **177**, 970-985 (2019).

[103] D. Hadjiabadi, M. Lovett-Barron, I. G. Raikov, F. T. Sparks, Z. Liao, S. C. Baraban, J. Leskovec, A. Losonczy, K. Deisseroth et I. Soltesz, "Maximally selective single-cell target for circuit control in epilepsy models", Neuron **109**, 2556-2572 (2021).

[104] A. Valente, J. W. Pillow et S. Ostojic, "Extracting computational mechanisms from neural data using low-rank RNNs", Advances in Neural Information Processing Systems **35**, 24072-24086 (2022).

[105] L. Ji-An, M. K. Benna et M. G. Mattar, "Automatic Discovery of Cognitive Strategies with Tiny Recurrent Neural Networks", bioRxiv, 1-33 (2023).

[106] K. J. Miller, M. Eckstein, M. M. Botvinick et Z. Kurth-Nelson, "Cognitive Model Discovery via Disentangled RNNs", bioRxiv, 1-12 (2023).

[107] J. G. Orlandi, M. Abdolrahmani, R. Aoki, D. R. Lyamzin et A. Benucci, "Distributed context-dependent choice information in mouse posterior cortex", Nature Communications **14**, 192 (2023).

[108] T. A. Machado, I. V. Kauvar et K. Deisseroth, "Multiregion neuronal activity : the forest and the trees", Nature Reviews Neuroscience **23**, 683-704 (2022).

[109] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier et A. Maida, "Deep learning in spiking neural networks", Neural networks **111**, 47-63 (2019).

[110] A. Taherkhani, A. Belatreche, Y. Li, G. Cosma, L. P. Maguire et T. M. McGinnity, "A review of learning in biologically plausible spiking neural networks", Neural Networks **122**, 253-272 (2020).

[111] A. McKenzie, D. W. Branch, C. Forsythe et C. D. James, "Toward exascale computing through neuromorphic approaches", Sandia Report SAND2010-6312, Sandia National Laboratories (2010).

[112] W. Maass, C. H. Papadimitriou, S. Vempala et R. Legenstein, "Brain computation : a computer science perspective", Computing and Software Science : State of the Art and Perspectives, 184-199 (2019).

[113] A. Siddique, M. I. Vai et S. H. Pun, "A low cost neuromorphic learning engine based on a high performance supervised SNN learning algorithm", Scientific Reports **13**, 6280 (2023).

[114] R. Siegwart, I. R. Nourbakhsh et D. Scaramuzza, *Introduction to Autonomous Mobile Robots*, 2nd (The MIT Press, 2011).

[115] H. Xiao, K. Rasul et R. Vollgraf, *Fashion-MNIST : a Novel Image Dataset for Benchmarking Machine Learning Algorithms*, 2017.

[116] E. Becht, L. McInnes, J. Healy, C.-A. Dutertre, I. W. Kwok, L. G. Ng, F. Ginhoux et E. W. Newell, "Dimensionality reduction for visualizing single-cell data using UMAP", Nature biotechnology **37**, 38-44 (2019).

[117] S. Herculano-Houzel, "The remarkable, yet not extraordinary, human brain as a scaled-up primate brain and its associated cost", Proceedings of the National Academy of Sciences **109**, 10661-10668 (2012).

[118] S. Herculano-Houzel *et al.*, "Energy supply per neuron is constrained by capillary density in the mouse brain", Frontiers in Integrative Neuroscience **16**, 760887 (2022).

[119] G. A. Wildenberg, M. R. Rosen, J. Lundell, D. Paukner, D. J. Freedman et N. Kasthuri, "Primate neuronal connections are sparse in cortex as compared to mouse", Cell Reports **36**, 109709 (2021).

[120] S. Loomba, J. Straehle, V. Gangadharan, N. Heike, A. Khalifa, A. Motta, N. Ju, M. Sievers, J. Gempt, H. S. Meyer *et al.*, "Connectomic comparison of mouse and human cortex", Science **377**, eabo0924 (2022).

[121] T. P. Peixoto, *The Netzschleuder network catalogue and repository – Connectome data*, `https://networks.skewed.de/?search=connectome`, 2020.

[122] R. C. Gerum, A. Erpenbeck, P. Krauss et A. Schilling, "Sparsity through evolutionary pruning prevents neuronal networks from overfitting", Neural Networks **128**, 305-312 (2020).

[123] N. Brunel et M. C. Van Rossum, "Lapicque's 1907 paper : from frogs to integrate-and-fire", Biological cybernetics **97**, 337-339 (2007).

[124] G. Bellec, F. Scherr, E. Hajek, D. Salaj, A. Subramoney, R. Legenstein et W. Maass, "Eligibility traces provide a data-inspired alternative to backpropagation through time", Real Neurons & Hidden Units : Future directions at the intersection of neuroscience and artificial intelligence@ NeurIPS 2019 (2019).

[125] M. London et M. Häusser, "Dendritic computation", Annu. Rev. Neurosci. **28**, 503-532 (2005).

[126] T. P. Lillicrap, D. Cownden, D. B. Tweed et C. J. Akerman, "Random synaptic feedback weights support error backpropagation for deep learning", Nature Communications **7**, 13276 (2016).

[127] H. Dale, "Pharmacology and Nerve-endings", Proceedings of the Royal Society of Medicine (1934).

[128] R. E. Blaser et Y. M. Peñalosa, "Stimuli affecting zebrafish (Danio rerio) behavior in the light/dark preference test", Physiology & Behavior **104**, 831-837 (2011).

Code availability

The complete NeuroTorch package, along with its source codes, is available at https://github.com/NeuroTorch/NeuroTorch. This repository contains all the necessary information for installing NeuroTorch and accessing benchmark data. Additionally, there are comprehensive tutorials available on Google Colab that provide hands-on learning experiences with the package. You can find a tutorial on Wilson-Cowan dynamics in NeuroTorch by following this link : Wilson-Cowan tutorial. Another tutorial available is focused on MNIST : MNIST tutorial. Furthermore, there is a tutorial specifically designed for the Heidelberg dataset : Heidelberg tutorial.

Acknowledgements

The authors are grateful to Mohamed Bahdine, whose scientific recommendations and master's research work [8] had a considerable impact on the elaboration of the current project. This work was supported by the the Natural Sciences and Engineering Research Council of Canada (A.L., P.D.K., P.D., S.H.), and the Sentinelle Nord program of Université Laval, funded by the Canada First Research Excellence Fund (P.D.K., P.D., S.H.). J.G. and A.L. want to thank the UNIQUE Research Center, funded by the Fonds de recherche du Québec – Nature et technologies, for Ph.D. and M.Sc. Excellence Scholarships. The authors acknowledge Calcul Québec and Digital Research Alliance of Canada for their technical support and computing infrastructures.

Author contributions statement

J.G., P.D., and S.V.H. designed the project. J.G. devised and coded the package and conducted the numerical experiments ; A.D. assisted J.G. in his computational work. A.L. collected the neuronal activity data ; P.D.K. supervised A.L.'s experimental work. J.G. drafted the manuscript and the NeuroTorch documentation. All authors analyzed the results and reviewed the manuscript.

Chapitre 3

Apprentissage par renforcement

Le chapitre 2 présente et teste un pipeline d'apprentissage standard avec des ensembles de données populaires. Cependant, celui-ci n'est pas adapté à l'apprentissage par renforcement (RL) qui gagne en importance dans le domaine de l'intelligence artificielle en raison des parallèles observés entre l'apprentissage du cerveau biologique et celui des systèmes artificiels [38-40]. Afin de palier ce manque, le chapitre 3 présente une adaptation de ce pipeline pour permettre l'apprentissage d'agents dans un contexte de RL, en utilisant un environnement couramment utilisé en RL.

Le chapitre se concentre sur la présentation du pipeline d'entraînement pour l'apprentissage par renforcement. Ensuite, une expérience de validation du pipeline est réalisée en utilisant l'environnement LunarLander de la librairie python Gym [51], avec des agents dotés soit d'un modèle de décision classique, soit d'un modèle de décision récurrent utilisant les dynamiques présentées dans les chapitres précédents. Enfin, une brève analyse de la résilience de ces modèles est effectuée, offrant ainsi une perspective sur les différentes caractéristiques des dynamiques utilisées.

3.1 Pipeline d'apprentissage par renforcement

Un pipeline d'apprentissage par renforcement consiste à faire évoluer un ou des agent(s) dans un ou des environnement(s). L'objectif de ce pipeline est d'entraîner ces agents à prendre des décisions qui maximisent les récompenses reçues. Ce pipeline se doit donc d'être en mesure d'exécuter des trajectoires de l'agent dans son environnement, de garder en mémoire ses expériences et de les utiliser afin d'optimiser les paramètres du modèle de l'agent. Finalement, considérant que les environnements possibles sont infinis, le pipeline se doit d'être le plus général possible.

Dans le chapitre 2, le pipeline classique présenté possède un ensemble de données préconstruit en entrée. Étant donné son implémentation flexible, son ensemble de données peut être changé

dynamiquement durant l'entraînement. Ceci permet à un utilisateur de créer, avec un peu de travail, son propre pipeline de RL en utilisant cette base. En effet, il peut collecter manuellement les expériences d'un agent dans un environnement et ainsi modifier cet ensemble de données à chaque itération. Toutefois, collecter manuellement ces expériences de façon efficace et lier ce processus à l'optimisation de l'agent requiert des connaissances et du temps. Ce dernier est une ressource rare dans le domaine de la recherche. C'est pourquoi la variante du pipeline de base nommé `RLAcademy` entre en jeu. En effet, contrairement à son prédécesseur, `RLAcademy` prend en entrée un environnement quelconque de la librairie Gym [51] et crée lui-même l'ensemble de données afin d'optimiser l'agent. La fonction `RLAcademy.generate_trajectories` est responsable de produire ces données (expériences et trajectoires de l'agent) en utilisant l'interface de l'API de la librairie Gym. Ce nouveau schéma d'apprentissage peut être observé à l'aide de la figure 3.1.1.

Une grande différence entre le pipeline de RL et le pipeline classique est l'importance des données d'entraînement. En effet, les données d'entraînement d'un problème classique sont habituellement statiques [1], c'est-à-dire qu'elles ne changent pas tout au long de l'entraînement, tandis que ce n'est pas nécessairement le cas en RL. Lorsque les données sont statiques en RL, c'est maintenant de l'apprentissage hors ligne (*offline learning*) [129] et cela est dû en général au fait que l'agent n'a pas accès à l'environnement ou ne peut interagir avec celui-ci lors de son entraînement. Toutefois, lorsque l'agent peut interagir avec son environnement et peut collecter des expériences de celui-ci, c'est maintenant de l'apprentissage en ligne (*online learning*) [130] et c'est dans cette optique que le pipeline de RL de NeuroTorch est construit. En apprentissage en ligne, énormément de données sont générées par les nombreux essais et erreurs de l'agent dans l'environnement et celles-ci se doivent d'être gérées intelligemment et efficacement. Pour ce faire, les expériences sont triées en ordre d'importance et les moins importantes sont éliminées à une certaine fréquence ou lorsqu'un seuil de quantité est atteint. L'objet `ReplayBuffer` est responsable de cette tâche et peut par exemple trier les expériences en termes d'avantage absolu afin de retirer celles qui influencent le moins la mise à jour des paramètres avec PPO (section 1.3.4). Le `ReplayBuffer` peut aussi être simplement vidé entre chaque itération comme cela est souvent pratiqué en RL.

L'apprentissage par renforcement se caractérise par son utilisation dans des environnements dynamiques, où l'agent interagit avec son environnement et accumule constamment des expériences. Après avoir exploré ce pipeline dynamique de NeuroTorch, il est primordial d'envisager la validation des performances pour évaluer son efficacité. La prochaine section abordera les tests de validation visant à évaluer la capacité du pipeline de RL de NeuroTorch à réussir une tâche spécifique.

1. Notons toutefois qu'avec NeuroTorch, il est possible de changer les données dynamiquement pendant l'entraînement.

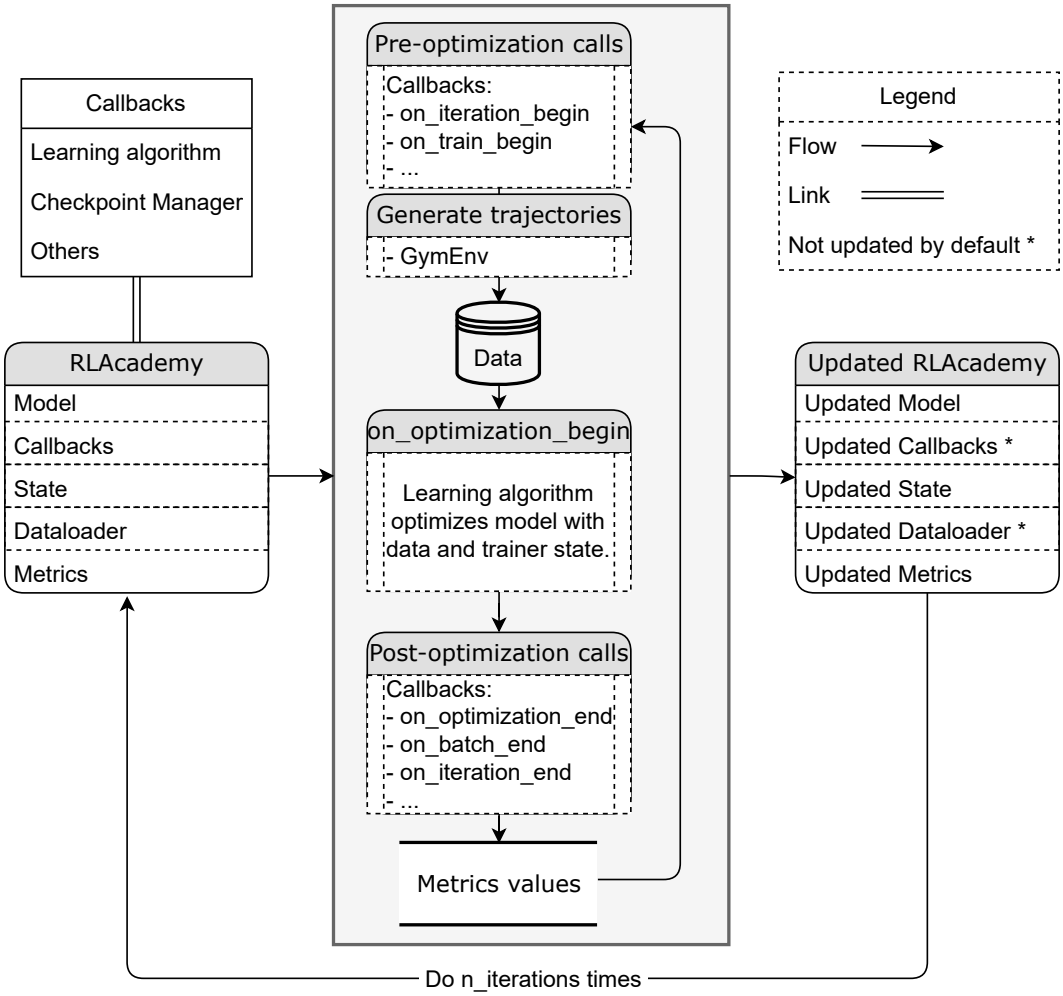

FIGURE 3.1.1 – Illustration du pipeline d'entraînement par renforcement (`RLAcademy`). Les objets `callbacks` sont donnés en entrée au pipeline et ce dernier appelle les évènements de ces objets aux moments prédéterminés. Le pipeline utilise l'environnement de type Gym afin de générer l'ensemble de données (d'expériences) qui seront utilisées par les `callbacks` pour optimiser le modèle de l'agent. Finalement, le pipeline et ses objets se voient modifiés pour la prochaine itération.

3.2 Validation

L'apprentissage par renforcement est extrêmement populaire en jeux vidéo puisqu'un jeu vidéo est l'environnement parfait pour simuler un agent/modèle dans un espace contrôlé. OpenAI a donc créé une plateforme python, Gym [51], afin de tester des modèles d'apprentissage par renforcement sur des environnements provenant de la théorie du contrôle et des jeux vidéo. Parmi ceux-ci, il y a LunarLander qui consiste à contrôler un vaisseau afin de le faire atterrir sur une plateforme. Cet environnement 2D, voir l'illustration à la figure 3.2.1, est caractérisé par une fonction de récompense attribuant un pointage pour le temps utilisé, la position finale et la vitesse finale de l'agent. Lorsque l'agent atteint une récompense cumulative de 200 (somme des récompenses de chaque expérience dans une trajectoire), la trajectoire est considérée comme réussie. Les interactions entre l'agent et l'environnement se font par l'intermédiaire des observations et des actions. Les observations données à chaque pas de temps à l'agent par l'environnement est un vecteur de huit éléments composé comme

1. Position horizontale (intervalle $[-1.5, 1.5]$) ;
2. Position verticale (intervalle $[-1.5, 1.5]$) ;
3. Vitesse horizontale (intervalle $[-5, 5]$) ;
4. Vitesse verticale (intervalle $[-5, 5]$) ;
5. Angle (intervalle $[-3.14, 3.14]$) ;
6. Vitesse angulaire (intervalle $[-5, 5]$) ;
7. Contact pied gauche («Vrai» ou «Faux») ;
8. Contact pied droite («Vrai» ou «Faux»).

Les actions que l'agent peut exécuter dans le cas présent est un vecteur de 2 éléments composé comme :

1. Moteur principal (intervalle $[-1, 1]$) ;
2. Moteur latéral (intervalle $[-1, 1]$).

LunarLander étant très populaire pour la validation et la comparaison de méthodes d'apprentissage, cet environnement est utilisé afin de valider le bon fonctionnement du pipeline de RL de NeuroTorch. Les paramètres d'environnement utilisés sont explicités dans le tableau 3.2.1. Pour confirmer que le pipeline de RL fonctionne comme souhaité, une expérience est menée en comparant les performances de différents modèles entraînés avec l'algorithme *proximal policy optimization* (PPO, voir section 1.3.4) sur cet environnement. Il est attendu que la majorité des modèles atteignent le critère de réussite afin de confirmer la fonctionnalité du pipeline.

Dans cette expérience, l'agent contient deux modèles. Le premier est la politique (*policy network*) qui est responsable de prendre les décisions et le second est un réseau de valeur (*value network* ou *critic network*) qui est responsable de quantifier la qualité des états de l'agent.

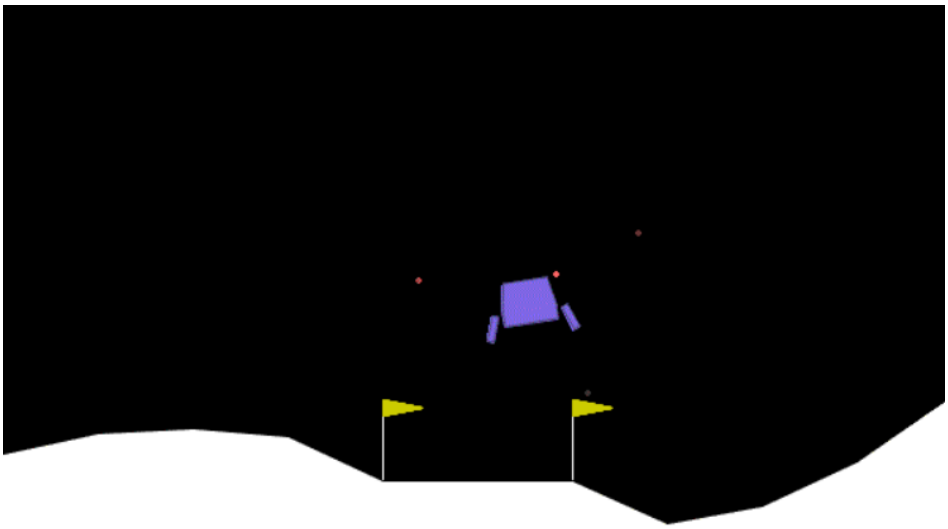

FIGURE 3.2.1 – Illustration de l'environnement LunarLander de Gym.

Paramètre	Valeur [-]
Id	LunarLander-v2
Type d'action	Continue
Gravité	-10.0
Vent	✓
Puissance du vent	5.0
Puissance de turbulence	0.5

Tableau 3.2.1 – Paramètres de l'environnement LunarLander utilisés.

Le réseau de valeurs utilisé est le même pour tous les agents qui sont comparés dans cette expérience afin de mettre en évidence les distinctions entre les différentes politiques. Plus concrètement, le réseau de valeurs est formé de trois couches linéaires séparées par des couches d'activation PReLU [131].

Dans l'objectif de comparer ce qui est fait classiquement et ce qui peut être fait avec Neuro-Torch, deux types de politiques sont donc utilisées. Le premier type est le "classique", celui-ci est composé de trois couches linéaires à 128 neurones (à l'exception de la dernière couche avec 1 seul neurone de sortie par action) séparées par des couches d'activation PReLU ainsi que des couches de *Dropout* [132] avec une probabilité de 10% d'être nulle. Le second type de politique est un réseau formé d'une des dynamiques suivantes avec 128 neurones : linéaire, LIF, ALIF, SpyLIF, SpyALIF, SpyLIF-LPF, SpyALIF-LPF et Wilson-Cowan. La couche de lecture du modèle est formée de la dynamique SpyLI avec 2 neurones de sortie (un pour chaque action). Finalement, le modèle possède une transformation constante (voir section 1.5.3) avec huit pas

de temps dans l'objectif de transformer l'entrée en série temporelle.

Pour tous les modèles, l'entraînement des différents réseaux avec PPO se fait avec les mêmes hyperparamètres sur 1000 itérations, 30 *epochs* et un `ReplayBuffer` qui se vide complètement entre chaque itération et sans utilisation de priorité pour les expériences. Cette méthode est utilisée afin de pouvoir bien isoler les différences entre les dynamiques et la fonctionnalité du pipeline de RL. Au total 90 modèles sont entraînés, ce qui donne 10 entraînements différents pour chaque type de politique. Finalement, l'entraînement de chaque modèle est accompagné d'un critère d'arrêt précoce qui suspend l'optimisation du modèle lorsque cela fait 10 itérations dont l'agent obtient une récompense cumulative moyenne supérieure à 230.

Afin d'évaluer la résilience des dynamiques précédentes, des expériences d'ablation ont été menées sur les modèles entraînés. Deux types d'ablation ont été considérés : une ablation de connexions hiérarchiques et une ablation de connexions aléatoires. L'ablation de connexions hiérarchiques implique la suppression des connexions ayant le moins d'impact sur le réseau, en supprimant les termes de plus petite valeur absolue de la matrice de connectivité par ordre croissant. En effet, considérant que $\frac{\mathrm{d}y}{\mathrm{d}w} \propto w$ où y est la sortie du modèle et w est le poids en question, il est possible de caractériser grossièrement l'importance d'un poids par sa valeur absolue. L'ablation de connexions aléatoires consiste à supprimer aléatoirement des éléments de la matrice de connectivité. Dans cette étude, 32 graines aléatoires (*seed*) ont été utilisées pour chaque point d'ablation aléatoire afin de garantir l'indépendance par rapport à la graine utilisée. Pour chaque méthode d'ablation, 32 niveaux d'éparsité ont été considérés, en ne supprimant que la couche présentant la dynamique. Ainsi, les poids de la couche de sortie ne sont pas affectés par l'ablation. Finalement, pour chaque test d'ablation, 64 trajectoires sont utilisées afin de calculer les moyennes et déviations standard des récompenses cumulatives.

3.3 Résultats

Le tableau 3.3.1 présente les récompenses cumulatives (CR) ainsi que le nombre d'itérations de convergence pour tous les modèles utilisés dans cette expérience. L'itération de convergence est ici définie comme le nombre d'itérations nécessaires pour que le modèle possède une récompense cumulative moyenne supérieure à 210 durant 10 itérations consécutives. Les récompenses cumulatives inscrites au tableau des résultats ont été calculées suite à 100 trajectoires de tests exécutés sur les modèles une fois entraînés. Il est donc possible de remarquer que le modèle ALIF obtient la meilleure récompense cumulative moyenne avec 241.28 ± 60.29. Du côté du nombre d'itérations de convergence, le modèle LIF converge plus rapidement en moyenne avec un nombre d'itérations moyen de 529.20 ± 26.13.

La figure 3.3.1 présente les courbes d'entraînement des différents modèles. Afin de mieux visualiser l'ensemble des courbes, la figure est divisée en quatre cadrans classifiant le type de

Modèle	Récompenses cumulatives [-]	Itérations de convergence [-]
ALIF	**241.28 ± 60.29**	543.50 ± 18.58
Classical	233.96 ± 66.12	535.20 ± 35.50
LIF	226.43 ± 73.54	**529.20 ± 26.13**
Linear	234.08 ± 78.21	555.10 ± 37.44
SpyALIF	229.58 ± 95.07	682.30 ± 216.45
SpyALIF-LPF	189.77 ± 132.28	646.70 ± 180.26
SpyLIF	213.92 ± 113.34	619.70 ± 167.77
SpyLIF-LPF	192.55 ± 127.94	660.60 ± 161.84
Wilson-Cowan	64.49 ± 142.49	679.30 ± 16.82

Tableau 3.3.1 – Moyenne et déviation standard de la somme cumulative de récompenses (CR) sur 100 trajectoires en test (après entraînement) pour l'environnement de LunarLander continue pour les différents types de modèles. Dans cette expérience l'environnement possédait une force de vent de 5 et une amplitude de turbulence de 0.5. Les résultats marqués en gras désignent les meilleurs résultats, donc le plus élevé pour CR et le plus petit pour le nombre d'itérations de convergence.

politique par le type de sortie de la dynamique utilisée. Chacune des courbes correspond à la moyenne de 10 entraînements distincts ainsi que sa déviation standard. Il est aussi possible de voir que dans certains cadrans, notamment les cadrans (a) et (c) que certaines courbes deviennent constantes. Cela signifie qu'à ce moment les entraînements de ce type de politique ont été arrêtés prématurément dû au critère d'arrêt. À première vue, cette figure montre qu'en général les différentes dynamiques sont optimisées aussi rapidement les unes que les autres à l'exception de la dynamique de Wilson-Cowan. En effet, cette dernière dynamique semble être optimisée avec une tendance linéaire au lieu d'exponentielle comme les autres, même si elle finit tout de même éventuellement par atteindre le critère d'arrêt.

La figure 3.3.2 présente les résultats de l'analyse de résilience des différentes dynamiques. Les figures en annexe B.0.1, B.0.2, B.0.3 et B.0.4 de leur côté présentent les mêmes données divisées par type de sortie avec un intervalle de confiance de 95% sur chacune des courbes. Celles-ci ont été séparées par souci de visibilité. De retour à la figure 3.3.2, dans le panneau **(a)**, des ablations de connexions hiérarchiques ont été réalisées en supprimant progressivement les connexions les plus petites des réseaux. Cette analyse permet d'observer l'éparsité effective des réseaux et les différences entre les dynamiques étudiées. Les résultats indiquent que la majorité des dynamiques doivent atteindre environ 0.6 d'éparsité avant de commencer à diminuer rapidement en performance. Tandis que la dynamique de Wilson-Cowan diminue très rapidement en performance dès 0.2 d'éparsité. Les dynamiques LPF semblent aussi diminuer plus rapidement que le reste du groupe en général. Ces résultats suggèrent que les dynamiques à impulsions et les modèles classiques peuvent avoir un niveau d'éparsité plus élevé sans compromettre leur efficacité par rapport à la dynamique Wilson-Cowan. Dans la sous-figure **(b)**, qui se concentre sur les ablations aléatoires de connexions, des connexions

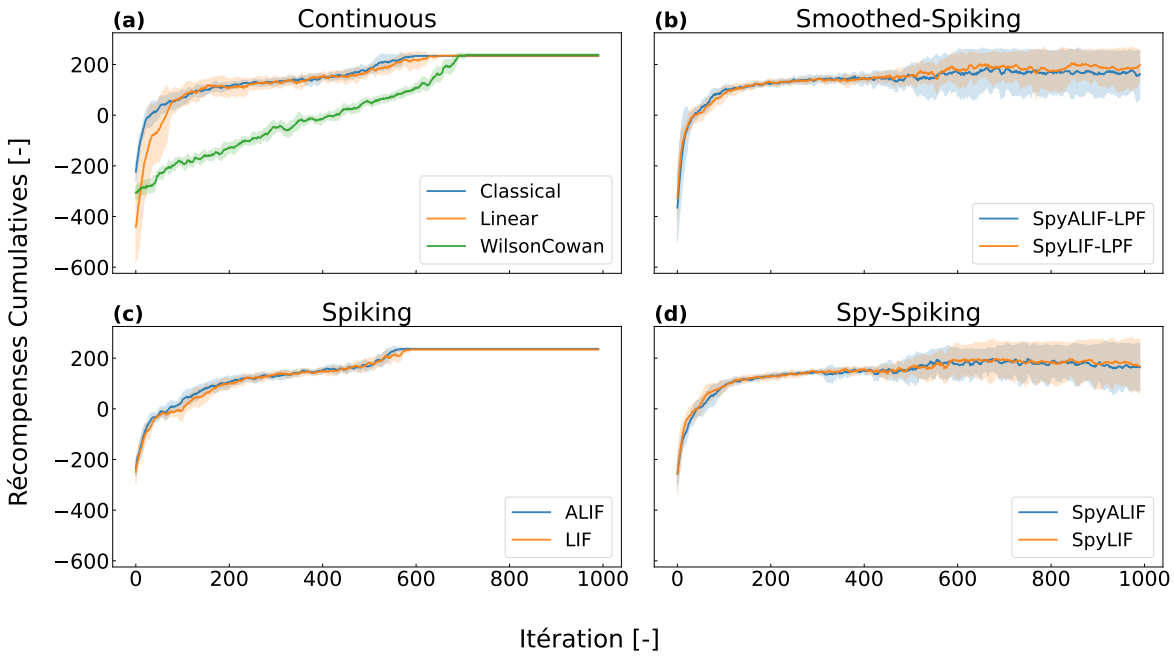

FIGURE 3.3.1 – Courbes d'entraînement des différents modèles dans l'environnement Lunar-Lander continu. **(a)** Les courbes d'entraînement des modèles continus. **(b)** Les courbes d'entraînement des modèles à impulsions filtrés par filtre passe-bas. **(c)** Les courbes d'entraînement des modèles à impulsions. **(d)** Les courbes d'entraînement des modèles à impulsions avec une équation différentielle supplémentaire pour le courant synaptique. Le moment où les courbes deviennent constantes est le moment où l'entraînement des modèles de cette courbe a été arrêté.

ont été supprimées aléatoirement des réseaux pour évaluer leur robustesse. À travers 32 simulations indépendantes pour chaque modèle, des distributions de résultats ont été obtenues. Initialement, toutes les courbes montrent une diminution rapide. Par la suite, il est possible de déceler quatre groupes en regardant l'intersection entre les courbes et l'axe des x. En effet, en ordre des moins résilients aux plus résilients il y a Wilson-Cowan, les dynamiques LPF, les dynamiques à impulsions et ensuite les modèles classiques. Les panneaux **(c)** et **(d)** présentent respectivement l'aire sous la courbe (AUC) pour chaque courbe dans les panneaux **(a)** et **(b)**. L'analyse du panneau **(c)** suggère que les dynamiques classiques et à impulsions sont plus résilientes contre l'ablation hiérarchique. Cependant, le panneau **(d)** indique que les modèles LPF et Spy sont plus résilients que LIF et ALIF contre l'ablation aléatoire.

3.4 Discussion

Les résultats présentés dans le tableau 3.3.1 montrent plusieurs tendances intéressantes. En effet, ce tableau suggère que les modèles de type ALIF sont plus performants en test que leur contrepartie LIF à l'exception des LPF. Par exemple, ALIF termine avec 14.85 CR de plus que LIF en moyenne, SpyALIF termine avec 15.66 CR de plus que SpyLIF en moyenne

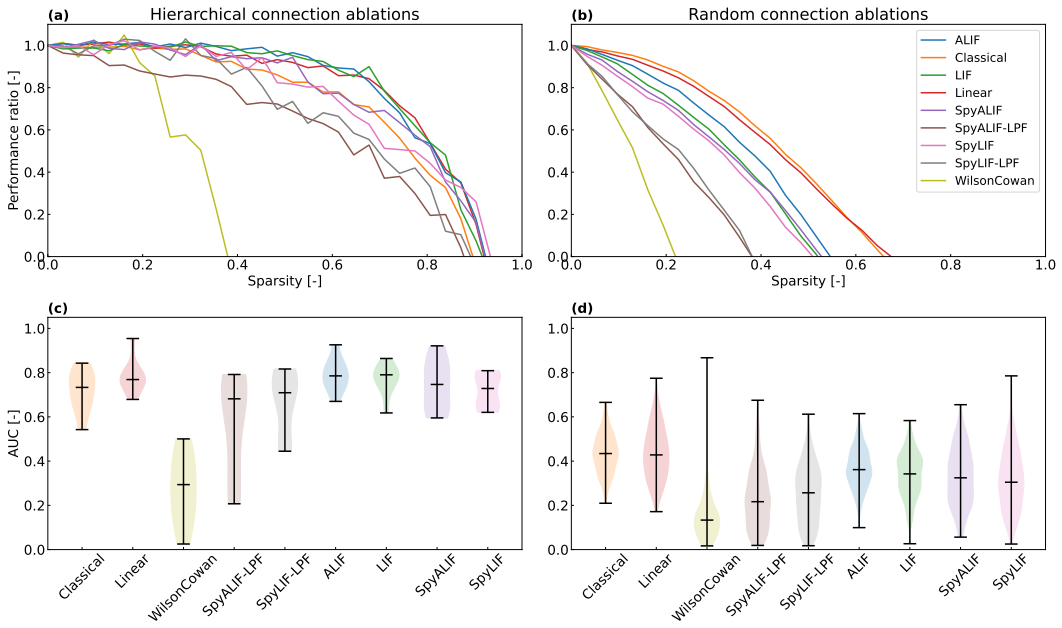

FIGURE 3.3.2 – Analyse de la résilience des modèles de réseaux basés sur les différentes dynamiques entraînées avec PPO. **(a)** Ablations de connexions hiérarchiques : à chaque étape, la connexion ayant le poids le plus faible en valeur absolue est supprimée du réseau. **(b)** Ablations de connexions aléatoires : des connexions aléatoires sont successivement supprimées du réseau. Ici, 32 graines aléatoires par modèle sont utilisées pour obtenir une distribution des ratios de performance pour chaque valeur d'éparsité. Chaque ligne solide suit la moyenne de la distribution correspondante. **(c, d)** Aire sous la courbe (AUC) pour chaque courbe dans (a) et (b) respectivement.

et SpyLIF-LPF termine avec 2.78 CR de plus que SpyALIF-LPF. Cette même tendance se maintient pour les itérations de convergence. Ce type de comportement semble suggérer que le seuil adaptatif des dynamiques ALIF aide à la performance pour les dynamiques à impulsions.

De plus, il est logique de croire que le modèle classique et la dynamique linéaire devraient surpasser les autres en termes de performance, toutefois, ce n'est pas tout à fait le cas. En effet, les dynamiques à impulsions ALIF et LIF atteignent non seulement les performances des modèles populaires, mais les dépassent. Ce qui est une bonne nouvelle pour les modèles neuromorphiques.

D'un autre côté, la dynamique Wilson-Cowan ne semble pas être adaptée à ce type de tâche puisqu'elle atteint un maigre 64.49 CR en moyenne. Ce résultat est non seulement le moins bon, mais ne réussit pas à dépasser le critère de réussite de 200 CR. Il est à noter cependant que ce résultat est surprenant considérant que les modèles possédant cette dynamique réussissent tout de même à atteindre le critère d'arrêt d'entraînement durant l'optimisation comme observé dans la figure 3.3.1. Cela suggère qu'un agent possédant la dynamique de Wilson-Cowan n'est pas fiable.

Pour ce qui est de la comparaison de la résilience, la figure 3.3.2 suggère qu'en général les dynamiques à impulsions et les modèles classiques sont plus robustes aux retraits de poids dans leurs matrices de connectivité. Toutefois, il est à noter que le modèle classique possède plus de couches, il en possède 3 avec la couche de sortie, incluant des couches de *dropout* qui consistent à retirer aléatoirement 10% des poids du réseau. Le modèle classique est donc d'une certaine façon optimisé pour être résilient à ce type de changement. Maintenant, en voyant que les dynamiques à impulsions sont en mesure d'atteindre une résilience comparable au modèle classique, il serait certainement intéressant de les comparer en utilisant des couches de *dropout* dans leur optimisation. Quoi qu'il en soit, la figure en question montre clairement dans son panneau (a) que la plupart des dynamiques peuvent avoir un niveau élevé d'éparsité sans trop affecter leur performance. Ce type de caractéristique est importante non seulement si l'objectif est de rendre les modèles plus efficaces au niveau énergétique, mais aussi pour rendre les modèles plus biologiquement plausibles.

En conclusion, les résultats sur l'environnement LunarLander confirment que l'implémentation du pipeline d'apprentissage par renforcement et de l'algorithme PPO fonctionne comme prévu. Les modèles ALIF ont surpassé les modèles LIF, à l'exception des LPF, suggérant que le seuil adaptatif des dynamiques ALIF améliore les performances. Cependant, la dynamique Wilson-Cowan n'est pas adaptée à cette tâche, n'atteignant pas le critère de réussite. Bien que les modèles basés sur cette dynamique aient atteint le critère d'arrêt d'entraînement, ce qui remet en question leur fiabilité pour ce type de tâche. En termes de résilience, les dynamiques à impulsions et les modèles classiques ont montré une plus grande robustesse aux retraits de poids. Les dynamiques à impulsions ont maintenu une performance comparable malgré leur éparsité, ce qui est important pour l'efficacité énergétique et la plausibilité biologique des modèles.

Conclusion

Le projet présenté dans ce mémoire avait pour objectif principal de développer la librairie NeuroTorch, dédiée aux neurosciences computationnelles, et d'intégrer différents modèles et dynamiques populaires dans ce domaine. Il visait également à fournir un pipeline d'apprentissage complet et intégré pour optimiser les modèles pour des tâches spécifiques. De plus, le projet comportait des objectifs secondaires, notamment l'implémentation de l'algorithme d'apprentissage *eligibility trace forward propagation* (e-prop) de Bellec et collab. [17], la comparaison de la résilience à l'ablation hiéarchique et l'ablation aléatoire entre les réseaux de neurones continus et les réseaux de neurones à impulsions, et enfin l'intégration d'un pipeline d'apprentissage par renforcement utilisant l'algorithme d'optimisation PPO à NeuroTorch.

Dans l'ensemble, ce projet a atteint ses objectifs de manière satisfaisante. La librairie NeuroTorch a été développée avec succès, comme présenté dans les chapitres 1 et 2, en intégrant différents modèles et dynamiques populaires en neurosciences computationnelles telles que LIF, ALIF et Wilson-Cowan. Cette librairie comble le besoin de disposer d'algorithmes d'apprentissage biologiquement plausibles et offre un pipeline d'apprentissage spécifiquement adapté aux exigences de la recherche en neurosciences computationnelles. NeuroTorch est disponible en libre d'accès sur GitHub au github.com/NeuroTorch.

De plus, l'objectif d'implémentation de l'algorithme d'apprentissage e-prop a été atteint, comme exposé dans le chapitre 2. Les dynamiques SpyLIF-LPF et Wilson-Cowan ont été testées avec succès sur des données neuronales expérimentales de la région ventrale de l'habenula du cerveau de poisson-zèbre, atteignant des pVar respectifs de 0.97 et 0.96. Ces résultats démontrent que l'algorithme e-prop peut bel et bien être utilisé pour optimiser des modèles sur des séries temporelles et prouve sa fonctionnalité dans NeuroTorch.

Une autre ambition était de comparer la résilience des réseaux de neurones continus et des réseaux de neurones à impulsions, afin de mieux comprendre les avantages et les limites de chaque approche en termes de défaillance neuronale et d'ablation hiérarchique. À cette fin, une comparaison de la résilience entre les dynamiques SpyLIF-LPF et Wilson-Cowan, avec et sans la loi de Dale, a été effectuée, comme présentée dans le chapitre 2. Les résultats suggèrent que l'application de la loi de Dale améliore la robustesse des modèles en termes d'ablation hiérarchique, mettant ainsi en évidence son importance dans les simulations en neurosciences

computationnelles. En effet, ces observations démontrent que l'utilisation de la loi de Dale pourrait permettre de créer des réseaux moins denses, réduisant ainsi la charge mémoire, tout en maintenant des performances comparables aux réseaux artificiels classiques. Finalement, ce concept ouvre de nouvelles perspectives quant à l'utilité de cette loi en neuroscience, notamment en tant qu'outil de régulation pour les réseaux endommagés.

Enfin, le dernier objectif principal de ce projet était l'intégration d'un pipeline d'apprentissage par renforcement utilisant l'algorithme d'optimisation PPO, comme exposé dans le chapitre 3. Cette intégration a été réalisée avec succès, permettant à différents types d'agents d'atteindre le critère de réussite dans l'environnement LunarLander. Cela ouvre de nouvelles possibilités d'implémenter d'autres algorithmes d'optimisations en RL tel que l'adaptation RL de e-prop tel que présenté dans l'article de Bellec et al. [17].

Les prochaines étapes du projet visent à renforcer et à étendre les fonctionnalités de NeuroTorch pour répondre aux besoins croissants de la recherche en neurosciences computationnelles. Voici quelques-unes des orientations prévues.

Tout d'abord, une étape importante consistera à assurer la compatibilité entre NeuroTorch et PyTorch Lightning [35]. Cette intégration permettra de combiner la flexibilité et la richesse des fonctionnalités de NeuroTorch avec les performances optimales offertes par PyTorch Lightning. Les chercheurs bénéficieront ainsi d'un environnement de développement plus fluide et efficace pour leurs travaux en neurosciences computationnelles.

Ensuite, une autre priorité sera de rendre NeuroTorch compatible avec *stable baselines 3* [41], une bibliothèque populaire d'algorithmes d'optimisation en renforcement learning. Cette intégration permettra aux utilisateurs de NeuroTorch de tirer parti de l'ensemble étendu et bien testé d'algorithmes de RL disponibles dans *stable baselines 3*, tout en profitant de la flexibilité et des fonctionnalités spécifiques à NeuroTorch pour modéliser des processus biologiquement plausibles.

Une autre direction envisagée est l'ajout de l'algorithme d'optimisation *Spike-Timing-Dependent Plasticity* (STDP), largement utilisé en neurosciences computationnelles. L'intégration de STDP dans NeuroTorch permettra aux chercheurs de modéliser et de simuler plus précisément les processus d'apprentissage synaptique dans les réseaux neuronaux, en alignant davantage les capacités de la librairie avec les mécanismes biologiques observés.

En parallèle, il est prévu de finaliser l'implémentation des algorithmes d'optimisation RLS présentés dans le chapitre 1 de manière à ce qu'ils soient pleinement fonctionnels dans NeuroTorch.

De plus, l'analyse de résilience des différents modèles présentée jusqu'à présent est encore sommaire, et il est prévu de pousser cette analyse plus en profondeur. Cela permettra de mieux

comprendre les différences entre les diverses dynamiques utilisées dans NeuroTorch et d'offrir des informations précieuses pour orienter le choix des modèles en fonction des spécificités des applications en neurosciences computationnelles.

Une autre application prometteuse de NeuroTorch réside dans la modélisation et la prédiction des comportements du poisson-zèbre. Grâce à la flexibilité de son pipeline et à la diversité de ses modèles, NeuroTorch pourrait être utilisé pour développer des réseaux de neurones capables de capturer les mécanismes sous-jacents des comportements observés chez ces poissons. En utilisant les séries temporelles d'activité neuronale en réponse à des stimuli spécifiques, il serait possible d'entraîner des modèles capables de prédire avec précision les réponses comportementales des poissons, offrant ainsi une application concrète et pratique à la librairie.

Enfin, une orientation prometteuse est l'utilisation de processeurs neuromorphiques pour exécuter les modèles développés avec NeuroTorch. L'intégration de ces processeurs avec la librairie permettra d'exploiter leur efficacité énergétique et leurs capacités de parallélisme massif pour accélérer les simulations de réseaux neuronaux.

En résumé, les prochaines étapes du projet visent à consolider et à enrichir NeuroTorch en renforçant son intégration avec d'autres bibliothèques, en étendant ses fonctionnalités avec des algorithmes spécifiques aux neurosciences computationnelles, en approfondissant l'analyse des modèles et en explorant les possibilités offertes par les processeurs neuromorphiques. Ces avancées permettront d'offrir aux chercheurs un outil puissant et adapté à leurs besoins pour leurs études en neurosciences computationnelles.

Bibliographie

[1] M. B. Ahrens, J. M. Li, M. B. Orger, D. N. Robson, A. F. Schier, F. Engert et R. Portugues, "Brain-wide neuronal dynamics during motor adaptation in zebrafish", Nature **485**, 471-477 (2012).

[2] J. Guilbert, A. Légaré, P. De Koninck, P. Desrosiers et M. Desjardins, "Toward an integrative neurovascular framework for studying brain networks", Neurophotonics **9**, 032211-032211 (2022).

[3] *Inferring brain-wide interactions using data-constrained recurrent neural network models*, 2021.

[4] D. L. Barabási, G. Bianconi, E. Bullmore, M. Burgess, S. Chung, T. Eliassi-Rad, D. George, I. A. Kovács, H. Makse, C. Papadimitriou, T. E. Nichols, O. Sporns, K. Stachenfeld, Z. Toroczkai, E. K. Towlson, A. M. Zador, H. Zeng, A.-L. Barabási, A. Bernard et G. Buzsáki, *Neuroscience needs Network Science*, 2023.

[5] D. S. Bassett et O. Sporns, "Network neuroscience", Nature neuroscience **20**, 353-364 (2017).

[6] L. K. Scheffer, C. S. Xu, M. Januszewski, Z. Lu, S.-y. Takemura, K. J. Hayworth, G. Huang, K. Shinomiya, J. Maitlin-Shepard, S. Berg *et al.*, "A Connectome and Analysis of the Adult Drosophila Central Brain", eLife **2020**, e57443 (2020).

[7] G. C. Vanwalleghem, M. B. Ahrens et E. K. Scott, "Integrative whole-brain neuroscience in larval zebrafish", Current opinion in neurobiology **50**, 136-145 (2018).

[8] M. Bahdine, "Simulation et inférence de réseaux de neurones à l'aide d'intelligence artificielle", mém. de mast. (Université Laval, 2020).

[9] A. M. Zador, "A critique of pure learning and what artificial neural networks can learn from animal brains", Nature Communications **10**, 3770 (2019).

[10] L. E. Suárez, B. A. Richards, G. Lajoie et B. Misic, "Learning function from structure in neuromorphic networks", Nature Machine Intelligence **3**, 771-786 (2021).

[11] F. Damicelli, C. C. Hilgetag et A. Goulas, "Brain connectivity meets reservoir computing", PLOS Computational Biology **18**, e1010639 (2022).

[12] D. Hassabis, D. Kumaran, C. Summerfield et M. Botvinick, "Neuroscience-inspired artificial intelligence", Neuron **95**, 245-258 (2017).

[13] A. H. Marblestone, G. Wayne et K. P. Kording, "Toward an Integration of Deep Learning and Neuroscience", Frontiers in Computational Neuroscience **10** (2016).

[14] E. M. Izhikevich, *Dynamical systems in neuroscience* (MIT Press, 2007), 522 p.

[15] B. Ermentrout et D. H. Terman, *Mathematical foundations of neuroscience*, t. 35 (Springer, 2010).

[16] T. Iakymchuk, A. Rosado-Muñoz, J. F. Guerrero-Martinez, M. Bataller-Mompeán et J. V. Francés-Villora, "Simplified spiking neural network architecture and STDP learning algorithm applied to image classification", EURASIP Journal on Image and Video Processing **2015**, 1-11 (2015).

[17] G. Bellec, F. Scherr, A. Subramoney, E. Hajek, D. Salaj, R. Legenstein et W. Maass, "A solution to the learning dilemma for recurrent networks of spiking neurons", Nature Communications **11**, 3625 (2020).

[18] T. P. Lillicrap et A. Santoro, "Backpropagation through time and the brain", Current opinion in neurobiology **55**, 82-89 (2019).

[19] G.-q. Bi et M.-m. Poo, "Synaptic modifications in cultured hippocampal neurons : dependence on spike timing, synaptic strength, and postsynaptic cell type", Journal of neuroscience **18**, 10464-10472 (1998).

[20] I. Goodfellow, Y. Bengio et A. Courville, *Deep learning* (MIT Press, 2016), 801 p.

[21] B. Chakraborty et S. Mukhopadhyay, *Brain-Inspired Spiking Neural Network for Online Unsupervised Time Series Prediction*, 2023.

[22] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang et H. Wang, "Loihi : A Neuromorphic Manycore Processor with On-Chip Learning", IEEE Micro **38**, 82-99 (2018).

[23] S. B. Furber, F. Galluppi, S. Temple et L. A. Plana, "The spinnaker project", Proceedings of the IEEE **102**, 652-665 (2014).

[24] T. DeWolf, "Spiking neural networks take control", Science Robotics **6**, eabk3268 (2021).

[25] G. Van Rossum et F. L. Drake, *Python 3 Reference Manual* (CreateSpace, 2009).

[26] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai et S. Chintala, "PyTorch : An Imperative Style, High-Performance Deep Learning Library", *Advances in Neural Information Processing Systems 32* (Curran Associates, Inc., 2019), p. 8024-8035.

[27] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu et Xiaoqiang Zheng, *TensorFlow : Large-Scale Machine Learning on Heterogeneous Systems*, 2015.

[28] C. Pehle et J. E. Pedersen, *Norse - A deep learning library for spiking neural networks*, version 0.0.7, 2021.

[29] E. O. Neftci, H. Mostafa et F. Zenke, "Surrogate Gradient Learning in Spiking Neural Networks : Bringing the Power of Gradient-Based Optimization to Spiking Neural Networks", IEEE Signal Processing Magazine **36**, 51-63 (2019).

[30] J. K. Eshraghian, M. Ward, E. Neftci, X. Wang, G. Lenz, G. Dwivedi, M. Bennamoun, D. S. Jeong et W. D. Lu, "Training spiking neural networks using lessons from deep learning", arXiv preprint arXiv :2109.12894 (2021).

[31] L. Deng, "The MNIST database of handwritten digit images for machine learning research", IEEE Signal Process. Mag. **29**, 141 (2012).

[32] Y. LeCun, C. Cortes et C. Burges, *MNIST handwritten digit database*, (1998)

[33] B. Cramer, Y. Stradmann, J. Schemmel et F. Zenke, "The Heidelberg Spiking Data Sets for the Systematic Evaluation of Spiking Neural Networks", IEEE Transactions on Neural Networks and Learning Systems, 1-14 (2022).

[34] D. L. Manna, A. Vicente-Sola, P. Kirkland, T. J. Bihl et G. D. Caterina, *Frameworks for SNNs : a Review of Data Science-oriented Software and an Expansion of SpykeTorch*, 2023.

[35] W. Falcon et The PyTorch Lightning team, *PyTorch Lightning*, version 1.4, 2019.

[36] F. Paradis, D. Beauchemin, M. Godbout, M. Alain, N. Garneau, S. Otte, A. Tremblay, M.-A. Bélanger et F. Laviolette, *Poutyne : A Simplified Framework for Deep Learning*, 2020.

[37] K. Arulkumaran, M. P. Deisenroth, M. Brundage et A. A. Bharath, "Deep Reinforcement Learning : A Brief Survey", IEEE Signal Processing Magazine **34**, 26-38 (2017).

[38] M. Botvinick, J. X. Wang, W. Dabney, K. J. Miller et Z. Kurth-Nelson, "Deep Reinforcement Learning and Its Neuroscientific Implications", Neuron **107**, 603-616 (2020).

[39] A. Subramanian, S. Chitlangia et V. Baths, "Reinforcement learning and its connections with neuroscience and psychology", Neural Networks **145**, 271-287 (2022).

[40] D. Lee, H. Seo et M. W. Jung, "Neural Basis of Reinforcement Learning and Decision Making", Annual Review of Neuroscience **35**, 287-308 (2012).

[41] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus et N. Dormann, "Stable-Baselines3 : Reliable Reinforcement Learning Implementations", Journal of Machine Learning Research **22**, 1-8 (2021).

[42] J. Gince, *NeuroTorch : A Python library for machine learning and neuroscience*, `https://github.com/NeuroTorch`, 2022.

[43] R. Siegwart, I. R. Nourbakhsh et D. Scaramuzza, "Kalman filter localization", *Introduction to Autonomous Mobile Robots*, 2nd (The MIT Press, 2011), p. 322-342.

[44] M. S. Al-Batah, N. A. Mat Isa, K. Z. Zamli et K. A. Azizli, "Modified recursive least squares algorithm to train the hybrid multilayered perceptron (HMLP) network", Applied Soft Computing **10**, 236-244 (2010).

[45] C. Zhang, Q. Song, H. Zhou, Y. Ou, H. Deng et L. T. Yang, *Revisiting Recursive Least Squares for Training Deep Neural Networks*, 2021.

[46] J. Schulman, F. Wolski, P. Dhariwal, A. Radford et O. Klimov, "Proximal Policy Optimization Algorithms", CoRR **abs/1707.06347** (2017).

[47] H. R. Wilson et J. D. Cowan, "Excitatory and inhibitory interactions in localized populations of model neurons", Biophysical journal **12**, 1-24 (1972).

[48] V. Painchaud, N. Doyon et P. Desrosiers, "Beyond Wilson-Cowan dynamics : oscillations and chaos without inhibition", Biological Cybernetics **116**, 527-543 (2022).

[49] T. P. Vogels, K. Rajan et L. Abbott, "NEURAL NETWORK DYNAMICS", Annual Review of Neuroscience **28**, 357-376 (2005).

[50] S. Grossberg, "Nonlinear neural networks : Principles, mechanisms, and architectures", Neural Networks **1**, 17-61 (1988).

[51] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang et W. Zaremba, *OpenAI Gym*, 2016.

[52] K. Hornik, M. Stinchcombe et H. White, "Multilayer feedforward networks are universal approximators", Neural networks **2**, 359-366 (1989).

[53] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli et J. Sohl-Dickstein, "On the expressive power of deep neural networks", international conference on machine learning (PMLR, 2017), p. 2847-2854.

[54] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser et I. Polosukhin, "Attention is all you need", Advances in neural information processing systems **30** (2017).

[55] K.-i. Funahashi et Y. Nakamura, "Approximation of dynamical systems by continuous time recurrent neural networks", Neural Networks **6**, 801-806 (1993).

[56] L. Jin, P. N. Nikiforuk et M. M. Gupta, "Approximation of discrete-time state-space trajectories using dynamic recurrent neural networks", IEEE Transactions on Automatic Control **40**, 1266-1270 (1995).

[57] N. Pavlidis, O. Tasoulis, V. Plagianakos, G. Nikiforidis et M. Vrahatis, "Spiking neural network training using evolutionary algorithms", Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005. T. 4 (2005), 2190-2194 vol. 4.

[58] C. Darwin, *The origin of species* (PF Collier & son New York, 1909).

[59] I. Galatzer-Levy, K. Ruggles et Z. Chen, "Data Science in the Research Domain Criteria Era : Relevance of Machine Learning to the Study of Stress Pathology, Recovery, and Resilience", Chronic Stress **2**, 247054701774755 (2018).

[60] Y. LeCun, Y. Bengio et G. Hinton, "Deep learning", nature **521**, 436-444 (2015).

[61] M. Lukoševičius, H. Jaeger et B. Schrauwen, "Reservoir computing trends", KI-Künstliche Intelligenz **26**, 365-371 (2012).

[62] P. Werbos, "Backpropagation through time : what it does and how to do it", Proceedings of the IEEE **78**, 1550-1560 (1990).

[63] F. Zenke et S. Ganguli, "SuperSpike : Supervised Learning in Multilayer Spiking Neural Networks", Neural Computation **30**, 1514-1541 (2018).

[64] M. R. Azimi-Sadjadi et R.-J. Liou, "Fast learning process of multilayer neural networks using recursive least squares method", IEEE Transactions on Signal Processing **40**, 446-450 (1992).

[65] M. R. Azimi-Sadjadi, S. Citrin et S. Sheedvash, "Supervised learning process of multi-layer perceptron neural networks using fast least squares", International Conference on Acoustics, Speech, and Signal Processing (IEEE, 1990), p. 1381-1384.

[66] O. Chapelle et L. Li, "An Empirical Evaluation of Thompson Sampling", Advances in Neural Information Processing Systems, t. 24, sous la dir. de J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira et K. Weinberger (2011).

[67] O.-C. Granmo, "Solving two-armed Bernoulli bandit problems using a Bayesian learning automaton", International Journal of Intelligent Computing and Cybernetics **3**, 207-234 (2010).

[68] B. C. May, N. Korda, A. Lee et D. S. Leslie, "Optimistic Bayesian sampling in contextual-bandit problems", Journal of Machine Learning Research **13**, 2069-2106 (2012).

[69] J. C. Eccles, "Chemical transmission and Dale's principle", Progress in Brain Research **68**, 3-13 (1986).

[70] P. Strata, R. Harvey *et al.*, "Dale's principle", Brain Research Bulletin **50**, 349-350 (1999).

[71] S. Zhou et Y. Yu, "Synaptic E-I Balance Underlies Efficient Neural Coding", Frontiers in Neuroscience **12** (2018).

[72] M. Xue, B. V. Atallah et M. Scanziani, "Equalizing excitation–inhibition ratios across visual cortical neurons", Nature **511**, 596-600 (2014).

[73] A. Bhatia, S. Moza et U. S. Bhalla, "Precise excitation-inhibition balance controls gain and timing in the hippocampus", eLife **8**, sous la dir. de R. L. Calabrese et C. D. Aizenman, e43415 (2019).

[74] B. G. Fenyves, G. S. Szilágyi, Z. Vassy, C. Sőti et P. Csermely, "Synaptic polarity and sign-balance prediction using gene expression data in the caenorhabditis elegans chemical synapse neuronal connectome network", PLOS Computational Biology **16**, e1007974 (2020).

[75] D. Bank, N. Koenigstein et R. Giryes, *Autoencoders*, 2021.

[76] C. Doersch, *Tutorial on Variational Autoencoders*, 2021.

[77] P. Baldi, "Autoencoders, Unsupervised Learning, and Deep Architectures", Proceedings of ICML Workshop on Unsupervised and Transfer Learning, t. 27, sous la dir. d'I. Guyon, G. Dror, V. Lemaire, G. Taylor et D. Silver, Proceedings of Machine Learning Research (2012), p. 37-49.

[78] L. McInnes, J. Healy et J. Melville, "UMAP : Uniform Manifold Approximation and Projection for Dimension Reduction", arXiv preprint arXiv :1802.03426 (2018).

[79] K. Pearson, "LIII. On lines and planes of closest fit to systems of points in space", The London, Edinburgh, and Dublin philosophical magazine and journal of science **2**, 559-572 (1901).

[80] L. Van der Maaten et G. Hinton, "Visualizing data using t-SNE.", Journal of machine learning research **9** (2008).

[81] I.-M. Comşa, L. Versari, T. Fischbacher et J. Alakuijala, "Spiking autoencoders with temporal coding", Frontiers in Neuroscience **15**, 712667 (2021).

[82] D. Hassabis, D. Kumaran, C. Summerfield et M. Botvinick, "Neuroscience-inspired artificial intelligence", Neuron **95**, 245-258 (2017).

[83] W. S. McCulloch et W. Pitts, "A logical calculus of the ideas immanent in nervous activity", The bulletin of mathematical biophysics **5**, 115-133 (1943).

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

[85] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities.", Proceedings of the national academy of sciences **79**, 2554-2558 (1982).

[86] R. S. Sutton et A. G. Barto, "Toward a modern theory of adaptive networks : expectation and prediction.", Psychological review **88**, 135 (1981).

[87] F. Pereira, T. Mitchell et M. Botvinick, "Machine learning classifiers and fMRI : a tutorial overview", Neuroimage **45**, S199-S209 (2009).

[88] M.-P. Hosseini, A. Hosseini et K. Ahi, "A review on machine learning for EEG signal processing in bioengineering", IEEE reviews in biomedical engineering **14**, 204-218 (2020).

[89] C. Stringer, T. Wang, M. Michaelos et M. Pachitariu, "Cellpose : a generalist algorithm for cellular segmentation", Nature Methods **18**, 100-106 (2021).

[90] F. Zhang, B. Pan, P. Shao, P. Liu, S. Shen, P. Yao, R. X. Xu, A. D. N. Initiative *et al.*, "A single model deep learning approach for Alzheimer's disease diagnosis", Neuroscience **491**, 200-214 (2022).

[91] J. Mei, C. Desrosiers et J. Frasnelli, "Machine learning for the diagnosis of Parkinson's disease : a review of literature", Frontiers in aging neuroscience **13**, 633752 (2021).

[92] J. Oh, B.-L. Oh, K.-U. Lee, J.-H. Chae et K. Yun, "Identifying schizophrenia using structural MRI with a deep learning algorithm", Frontiers in psychiatry **11**, 16 (2020).

[93] D. L. Yamins et J. J. DiCarlo, "Using goal-driven deep learning models to understand sensory cortex", Nature neuroscience **19**, 356-365 (2016).

[94] A. H. Marblestone, G. Wayne et K. P. Kording, "Toward an integration of deep learning and neuroscience", Frontiers in Computational Neuroscience, 94 (2016).

[95] A. M. Zador, "A critique of pure learning and what artificial neural networks can learn from animal brains", Nature communications **10**, 3770 (2019).

[96] M. P. Van den Heuvel, E. T. Bullmore et O. Sporns, "Comparative connectomics", Trends in cognitive sciences **20**, 345-361 (2016).

[97] R. F. Betzel et D. S. Bassett, "Specificity and robustness of long-distance connections in weighted, interareal connectomes", Proceedings of the National Academy of Sciences **115**, E4880-E4889 (2018).

[98] S. Vyas, M. D. Golub, D. Sussillo et K. V. Shenoy, "Computation through neural population dynamics", Annual Review of Neuroscience **43**, 249-275 (2020).

[99] D. Sussillo et L. F. Abbott, "Generating coherent patterns of activity from chaotic neural networks", Neuron **63**, 544-557 (2009).

[100] B. DePasquale, C. J. Cueva, K. Rajan, G. S. Escola et L. Abbott, "full-FORCE : A target-based method for training recurrent networks", PloS one **13**, e0191527 (2018).

[101] T. Miconi, "Biologically plausible learning in recurrent neural networks reproduces neural dynamics observed during cognitive tasks", eLife **6**, e20899 (2017).

[102] A. S. Andalman, V. M. Burns, M. Lovett-Barron, M. Broxton, B. Poole, S. J. Yang, L. Grosenick, T. N. Lerner, R. Chen, T. Benster *et al.*, "Neuronal dynamics regulating brain and behavioral state transitions", Cell **177**, 970-985 (2019).

[103] D. Hadjiabadi, M. Lovett-Barron, I. G. Raikov, F. T. Sparks, Z. Liao, S. C. Baraban, J. Leskovec, A. Losonczy, K. Deisseroth et I. Soltesz, "Maximally selective single-cell target for circuit control in epilepsy models", Neuron **109**, 2556-2572 (2021).

[104] A. Valente, J. W. Pillow et S. Ostojic, "Extracting computational mechanisms from neural data using low-rank RNNs", Advances in Neural Information Processing Systems **35**, 24072-24086 (2022).

[105] L. Ji-An, M. K. Benna et M. G. Mattar, "Automatic Discovery of Cognitive Strategies with Tiny Recurrent Neural Networks", bioRxiv, 1-33 (2023).

[106] K. J. Miller, M. Eckstein, M. M. Botvinick et Z. Kurth-Nelson, "Cognitive Model Discovery via Disentangled RNNs", bioRxiv, 1-12 (2023).

[107] J. G. Orlandi, M. Abdolrahmani, R. Aoki, D. R. Lyamzin et A. Benucci, "Distributed context-dependent choice information in mouse posterior cortex", Nature Communications **14**, 192 (2023).

[108] T. A. Machado, I. V. Kauvar et K. Deisseroth, "Multiregion neuronal activity : the forest and the trees", Nature Reviews Neuroscience **23**, 683-704 (2022).

[109] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier et A. Maida, "Deep learning in spiking neural networks", Neural networks **111**, 47-63 (2019).

[110] A. Taherkhani, A. Belatreche, Y. Li, G. Cosma, L. P. Maguire et T. M. McGinnity, "A review of learning in biologically plausible spiking neural networks", Neural Networks **122**, 253-272 (2020).

[111] A. McKenzie, D. W. Branch, C. Forsythe et C. D. James, "Toward exascale computing through neuromorphic approaches", Sandia Report SAND2010-6312, Sandia National Laboratories (2010).

[112] W. Maass, C. H. Papadimitriou, S. Vempala et R. Legenstein, "Brain computation : a computer science perspective", Computing and Software Science : State of the Art and Perspectives, 184-199 (2019).

[113] A. Siddique, M. I. Vai et S. H. Pun, "A low cost neuromorphic learning engine based on a high performance supervised SNN learning algorithm", Scientific Reports **13**, 6280 (2023).

[114] R. Siegwart, I. R. Nourbakhsh et D. Scaramuzza, *Introduction to Autonomous Mobile Robots*, 2nd (The MIT Press, 2011).

[115] H. Xiao, K. Rasul et R. Vollgraf, *Fashion-MNIST : a Novel Image Dataset for Benchmarking Machine Learning Algorithms*, 2017.

[116] E. Becht, L. McInnes, J. Healy, C.-A. Dutertre, I. W. Kwok, L. G. Ng, F. Ginhoux et E. W. Newell, "Dimensionality reduction for visualizing single-cell data using UMAP", Nature biotechnology **37**, 38-44 (2019).

[117] S. Herculano-Houzel, "The remarkable, yet not extraordinary, human brain as a scaled-up primate brain and its associated cost", Proceedings of the National Academy of Sciences **109**, 10661-10668 (2012).

[118] S. Herculano-Houzel *et al.*, "Energy supply per neuron is constrained by capillary density in the mouse brain", Frontiers in Integrative Neuroscience **16**, 760887 (2022).

[119] G. A. Wildenberg, M. R. Rosen, J. Lundell, D. Paukner, D. J. Freedman et N. Kasthuri, "Primate neuronal connections are sparse in cortex as compared to mouse", Cell Reports **36**, 109709 (2021).

[120] S. Loomba, J. Straehle, V. Gangadharan, N. Heike, A. Khalifa, A. Motta, N. Ju, M. Sievers, J. Gempt, H. S. Meyer *et al.*, "Connectomic comparison of mouse and human cortex", Science **377**, eabo0924 (2022).

[121] T. P. Peixoto, *The Netzschleuder network catalogue and repository – Connectome data*, `https://networks.skewed.de/?search=connectome`, 2020.

[122] R. C. Gerum, A. Erpenbeck, P. Krauss et A. Schilling, "Sparsity through evolutionary pruning prevents neuronal networks from overfitting", Neural Networks **128**, 305-312 (2020).

[123] N. Brunel et M. C. Van Rossum, "Lapicque's 1907 paper : from frogs to integrate-and-fire", Biological cybernetics **97**, 337-339 (2007).

[124] G. Bellec, F. Scherr, E. Hajek, D. Salaj, A. Subramoney, R. Legenstein et W. Maass, "Eligibility traces provide a data-inspired alternative to backpropagation through time", Real Neurons & Hidden Units : Future directions at the intersection of neuroscience and artificial intelligence@ NeurIPS 2019 (2019).

[125] M. London et M. Häusser, "Dendritic computation", Annu. Rev. Neurosci. **28**, 503-532 (2005).

[126] T. P. Lillicrap, D. Cownden, D. B. Tweed et C. J. Akerman, "Random synaptic feedback weights support error backpropagation for deep learning", Nature Communications **7**, 13276 (2016).

[127] H. Dale, "Pharmacology and Nerve-endings", Proceedings of the Royal Society of Medicine (1934).

[128] R. E. Blaser et Y. M. Peñalosa, "Stimuli affecting zebrafish (Danio rerio) behavior in the light/dark preference test", Physiology & Behavior **104**, 831-837 (2011).

[129] S. Levine, A. Kumar, G. Tucker et J. Fu, *Offline Reinforcement Learning : Tutorial, Review, and Perspectives on Open Problems*, 2020.

[130] A. Nagabandi, C. Finn et S. Levine, *Deep Online Learning via Meta-Learning : Continual Adaptation for Model-Based RL*, 2019.

[131] B. Ding, H. Qian et J. Zhou, "Activation functions and their characteristics in deep neural networks", 2018 Chinese Control And Decision Conference (CCDC) (2018), p. 1836-1841.

[132] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever et R. R. Salakhutdinov, *Improving neural networks by preventing co-adaptation of feature detectors*, 2012.

[133] *Papers with code - fashion-MNIST benchmark (image classification).*

Annexe A

Réduction dimensionnelle éparse

Dans cette expérience, la fonctionnalité du pipeline d'entraînement de NeuroTorch ainsi que la viabilité des dynamiques à impulsions est démontrée en exécutant l'auto-encodage des ensembles de données MNIST [31, 32] et Fashion-MNIST [115]. En effet, dans un premier temps un auto-encodeur à impulsion sera utilisé pour simplement encoder et décoder les images pour ensuite y ajouter une couche de classification afin de montrer qu'il est possible d'effectuer un entraînement multitâches avec NeuroTorch et que les dynamiques à décharges sont en mesure de produire un espace latent riche et utile pour des tâches subséquentes.

Le jeu de données MNIST [31, 32] est constitué d'images en noir et blanc, 28 pixels par 28 pixels, de chiffres écrits à la main. Chacune des images possède un seul chiffre appartenant à une des 10 classes désignant le chiffre écrit de 0 à 9. Ce jeu de données est séparé en deux ensembles distincts de 60 000 images d'entraînements et 10 000 images de test. Des exemples de ces images sont affichés à la figure A.0.1. L'état de l'art des perceptrons multicouches (PMC) de cette base de données est bien documenté. Pour un perceptron multicouche, la précision maximale trouvée est de 95.3% tandis que pour un réseau à convolution la précision maximale est de 98.9% selon [32]. De nos jours, ces précisions se rapprochent plus de 100%, surtout pour les réseaux à convolutions, mais cette dernière référence est suffisante pour savoir si le pipeline de NeuroTorch est fonctionnel.

FIGURE A.0.1 – Exemples de données du jeu de données MNIST où chaque ligne est une classe distincte. Image tirée de Wikipédia [1].

Le jeu de données Fashion-MNIST [115] est composé de 60 000 images d'entraînements et 10 000 images de test. Tout comme MNIST, ces images sont en noir et blanc et 28 pixels par 28 pixels. Les données sont divisées en 10 classes distinctes de vêtements ou accessoires. Un exemple de ces données peut être visualisé à la figure A.0.2. L'état de l'art pour cette base de données est moins exhaustif que MNIST, puisqu'elle est beaucoup plus récente. Le réseau se rapprochant le plus de ce qui est implémenté dans cette expérience est le réseau à convolution *Tsetlin machine* avec une précision de 91.4% [133]. Il est à noter que ce dernier réseau n'est pas comparable aux réseaux à impulsions testés en termes de structure et de dynamique.

1. Wikipedia : https://en.wikipedia.org/wiki/MNIST_database

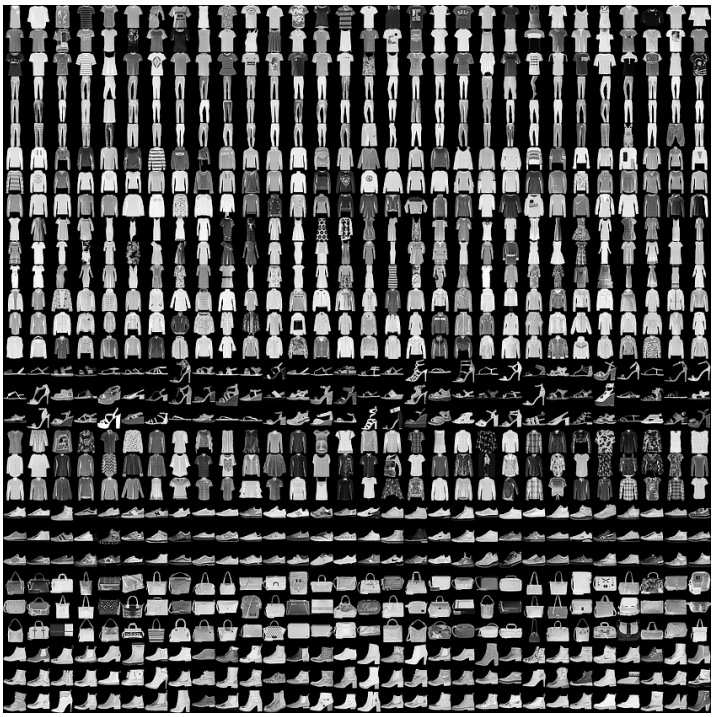

FIGURE A.0.2 – Exemples de données du jeu de données Fashion-MNIST où chaque ligne est une classe distincte. Image tirée de [115].

Pour effectuer l'auto-encodage des images des deux ensembles de données précédentes, le modèle de la figure A.0.3 est utilisé. Celui-ci possède exactement la même structure que celle présentée à la section 1.5.4 avec des fonctions de transformations supplémentaires en entrée et en sortie du réseau. En effet, afin d'être en mesure d'utiliser la dynamique SpyLIF comme encodeur, les images d'entrées sont converties en séries temporelles en utilisant la transformation présentée à la section 1.5.3 et 4 pas de temps. La taille de l'espace latent est de 128 ce qui donne des données de taille (4, 128) dans l'espace latent. Considérant que les images d'entrée sont de taille (1, 784), c'est un facteur de compression théorique d'environ 35%. Pour le décodage de l'espace latent, une dynamique SpyLI est utilisée et moyennée dans la dimension du temps pour reconstruire l'image d'entrée. Ce choix de dynamique et de transformation est complètement arbitraire et pourrait être changé pour augmenter les performances de ce modèle. Toutefois, cette architecture est suffisante pour faire la démonstration d'auto-encodeur avec NeuroTorch. Le code afin de régénérer les résultats reliés à cette expérience est dans le répertoire github relié à ce projet dans le script «SpikingAutoEncoder/auto_encoder_script.py».

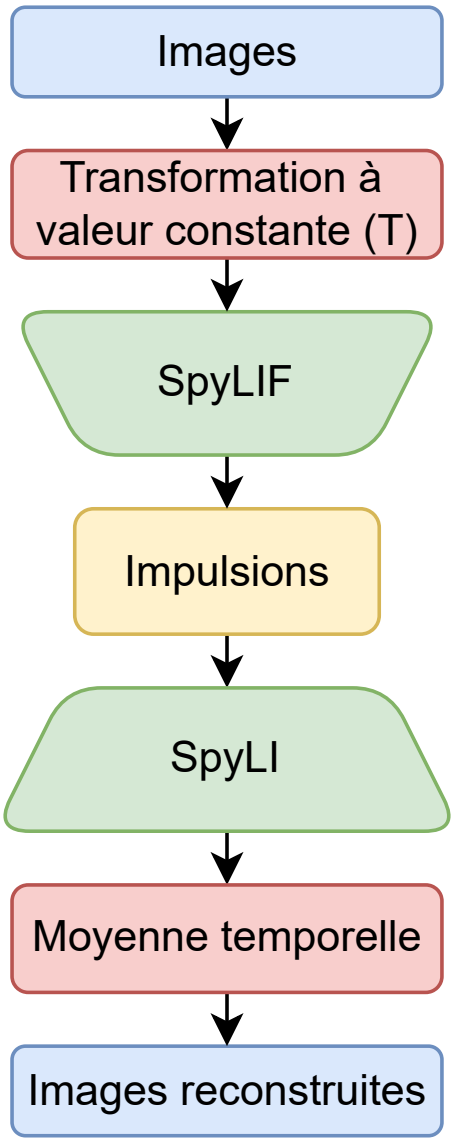

FIGURE A.0.3 – Illustration d'un auto-encodeur SpyLIF.

Le tableau A.0.1 présente les résultats de l'entraînement du réseau à impulsion pour l'auto-encodage des données d'MNIST et de Fashion-MNIST.

Données [-]	Méthode [-]	Taille d'encodage [octets]	pVar [-]	Compression [%]
MNIST	Encode	2451.00 ± 0.00	0.89 ± 0.04	25.43 ± 0.00
	Encode éparse	1347.76 ± 256.58		**59.00 ± 7.81**
Fashion-MNIST	Encode	2451.00 ± 0.00	0.87 ± 0.10	25.43 ± 0.00
	Encode éparse	1320.62 ± 263.54		**59.82 ± 8.02**

Tableau A.0.1 – Compressibilité des données d'MNIST et de Fashion-MNIST de 3287 octets pour les deux types de compressions. La pVar est la mesure de qualité de reconstruction des données.

Dans ce dernier tableau, la colonne de compression fait référence à la différence d'octets relative à la taille de l'image d'entrée ($100 \cdot \frac{\text{Taille[x] - Taille[h]}}{\text{Taille[x]}}$ où h est x transformé dans l'espace latent). Ces résultats montrent qu'il est possible de réduire l'espace mémoire des données d'environ 25% seulement en ayant un espace latent plus petit que l'espace initial. De plus en utilisant un encodage épars, il est possible de compresser jusqu'à 59% l'image d'entrée.

Une fois qu'il a été montré qu'il est possible d'effectuer un entraînement conventionnel d'auto-encodeur avec le pipeline de NeuroTorch, il est nécessaire de démontrer qu'il est possible de faire de l'entraînement multitâche utilisant le même pipeline. La figure A.0.4 illustre le modèle classifieur et auto-encodeur qui sera entraîné et testé. Ce modèle est pertinent dû à l'utilisation de la même couche SpyLIF pour créer un espace latent qui sera utile afin de recréer l'entrée du réseau et aussi pour classifier celle-ci. Le code afin de regénérer les résultats reliés à cette expérience est dans le répertoire github relié à ce projet dans le script «SpikingAutoEncoder/classifier_auto_encoder.py».

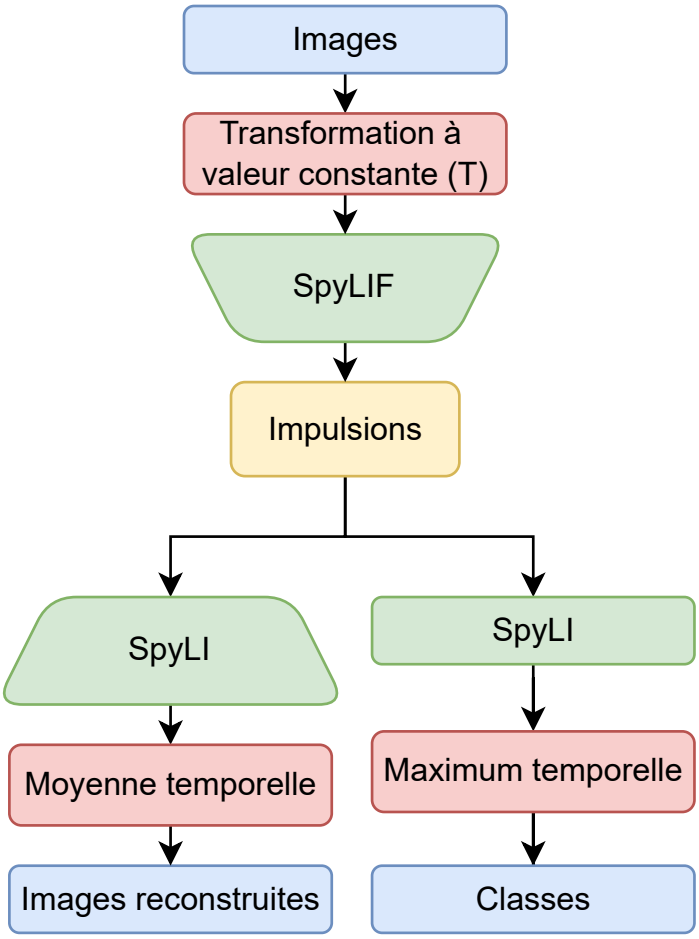

FIGURE A.0.4 – Illustration d'un auto-encodeur classifieur SpyLIF.

Le tableau A.0.2 présente les résultats de l'entraînement de l'auto encodeur et classifieur sur l'ensemble de données d'MNIST et de Fashion-MNIST. Premièrement, les exactitudes de 97.49% et 87.03% sont respectivement à +2.15% et -4.37% des exactitudes de références présentées plus tôt pour MNIST et Fashion-MNIST. Considérant que le modèle de référence de Fashion-MNIST est à convolutions, ces exactitudes sont très respectables pour une expérience où l'objectif n'est pas d'avoir les exactitudes les plus optimisées possibles, mais seulement pour montrer qu'il est possible de faire de l'entraînement multitâches avec NeuroTorch.

Données [-]	Méthode [-]	Exactitude [%]
MNIST	AE+CLS	97.45
	CLS	**97.49**
Fashion-MNIST	AE+CLS	86.64
	CLS	**87.03**

Tableau A.0.2 – Exactitudes en test pour l'ensemble de données d'MNIST et de Fashion-MNIST avec les deux types de réseaux CLS étant le réseau seulement classifieur et AE+CLS étant le réseau classifieur et auto-encodeur.

De plus, dans le tableau A.0.2, il est possible de remarquer que l'ajout d'un auto-encodeur au classifieur n'affecte presque pas les performances du modèle. En effet, les modèles de types AE+CLS ont une exactitude d'environ 1% moins que leur correspondant CLS.

Annexe B

Figures supplémentaires - Résilience

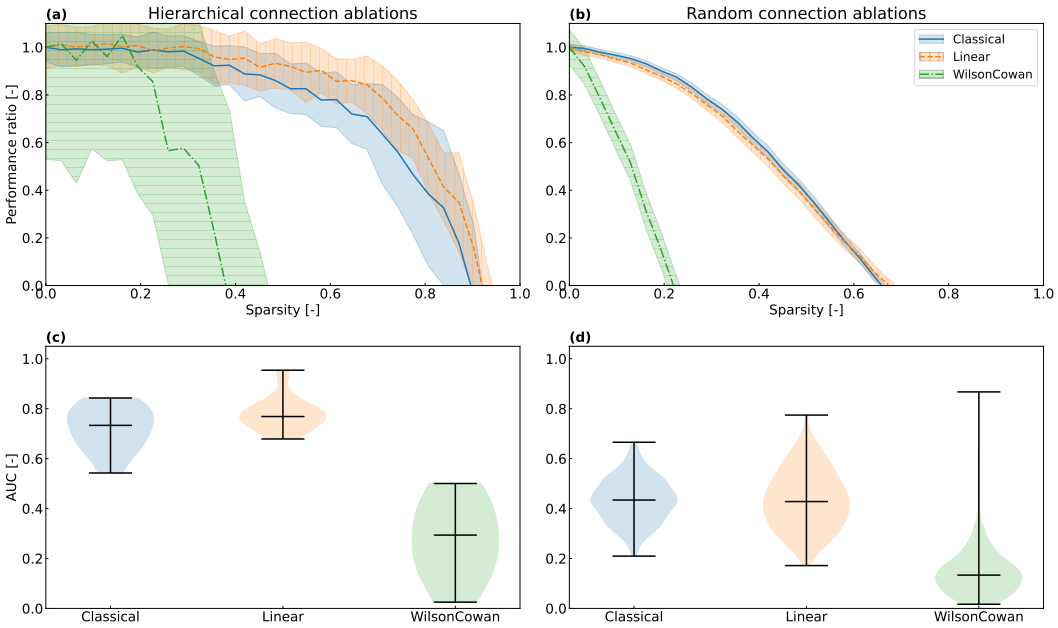

FIGURE B.0.1 – Analyse de la résilience du modèle classique ainsi que des modèles de réseau basés sur les dynamiques linéaires et Wilson-Cowan, entraînés avec PPO. **(a)** Ablations de connexions hiérarchiques : à chaque étape, la connexion ayant le poids le plus faible en valeur absolue est supprimée du réseau. **(b)** Ablations de connexions aléatoires : des connexions aléatoires sont successivement supprimées du réseau. Ici, 32 graines aléatoires par modèle sont utilisées pour obtenir une distribution des ratios de performance pour chaque valeur d'éparsité. Chaque ligne solide ou pointillées suit la moyenne de la distribution correspondante, tandis que chaque région ombrée autour d'une ligne représente un intervalle de confiance de 95%. **(c, d)** Aire sous la courbe (AUC) pour chaque courbe dans (a) et (b) respectivement.

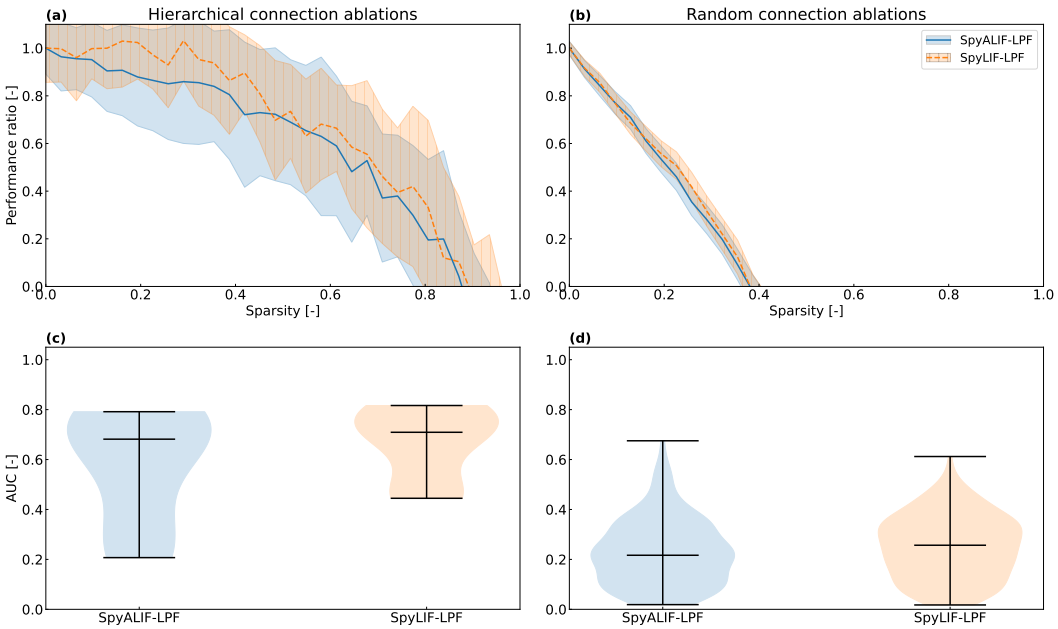

FIGURE B.0.2 – Analyse de la résilience des modèles de réseau basés sur les dynamiques SpyALIF-LPF et SpyLIF-LPF, entraînés avec PPO. **(a)** Ablations de connexions hiérarchiques : à chaque étape, la connexion ayant le poids le plus faible en valeur absolue est supprimée du réseau. **(b)** Ablations de connexions aléatoires : des connexions aléatoires sont successivement supprimées du réseau. Ici, 32 graines aléatoires par modèle sont utilisées pour obtenir une distribution des ratios de performance pour chaque valeur d'éparsité. Chaque ligne solide ou pointillées suit la moyenne de la distribution correspondante, tandis que chaque région ombrée autour d'une ligne représente un intervalle de confiance de 95%. **(c, d)** Aire sous la courbe (AUC) pour chaque courbe dans (a) et (b) respectivement.

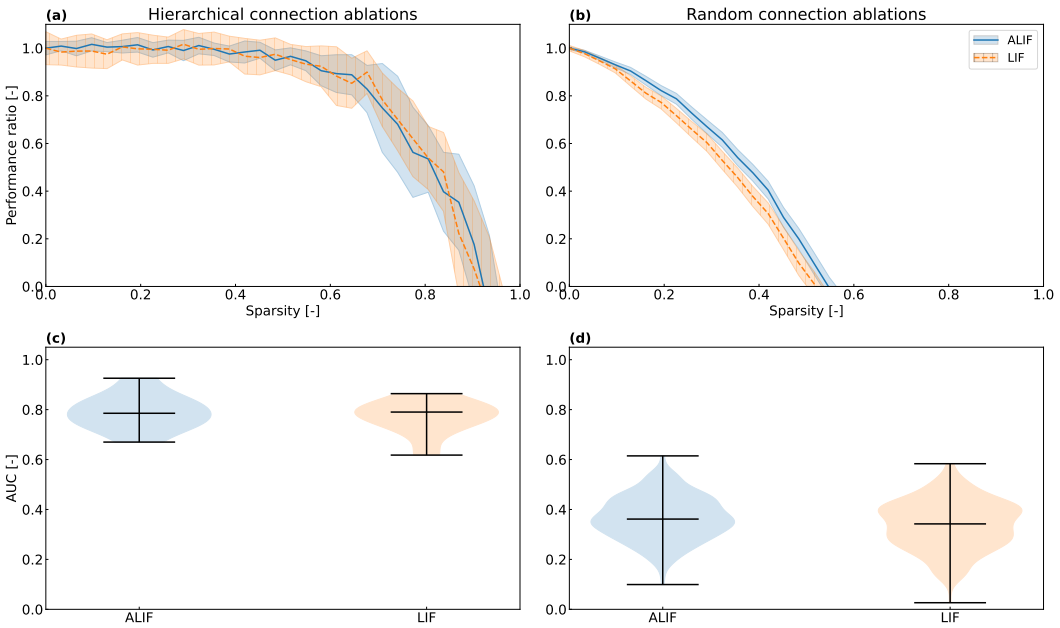

FIGURE B.0.3 – Analyse de la résilience des modèles de réseau basés sur les dynamiques ALIF et LIF, entraînés avec PPO. **(a)** Ablations de connexions hiérarchiques : à chaque étape, la connexion ayant le poids le plus faible en valeur absolue est supprimée du réseau. **(b)** Ablations de connexions aléatoires : des connexions aléatoires sont successivement supprimées du réseau. Ici, 32 graines aléatoires par modèle sont utilisées pour obtenir une distribution des ratios de performance pour chaque valeur d'éparsité. Chaque ligne solide ou pointillées suit la moyenne de la distribution correspondante, tandis que chaque région ombrée autour d'une ligne représente un intervalle de confiance de 95%. **(c, d)** Aire sous la courbe (AUC) pour chaque courbe dans (a) et (b) respectivement.

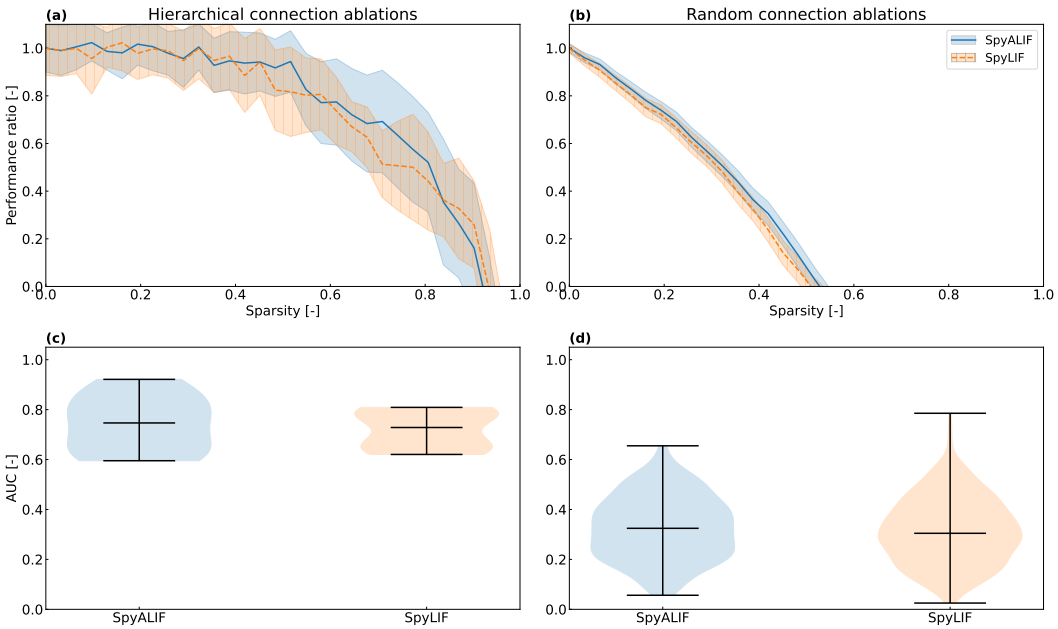

FIGURE B.0.4 – Analyse de la résilience des modèles de réseau basés sur les dynamiques SpyALIF et SpyLIF, entraînés avec PPO. **(a)** Ablations de connexions hiérarchiques : à chaque étape, la connexion ayant le poids le plus faible en valeur absolue est supprimée du réseau. **(b)** Ablations de connexions aléatoires : des connexions aléatoires sont successivement supprimées du réseau. Ici, 32 graines aléatoires par modèle sont utilisées pour obtenir une distribution des ratios de performance pour chaque valeur d'éparsité. Chaque ligne solide ou pointillées suit la moyenne de la distribution correspondante, tandis chaque région ombrée autour d'une ligne représente un intervalle de confiance de 95%. **(c, d)** Aire sous la courbe (AUC) pour chaque courbe dans (a) et (b) respectivement.