ÉCOLE CENTRALE LYON

Projet d'application de Recherche

Commandité par le Laboratoire LIRIS

Rapport PaR 160 : Incremental Learning

Auteurs :
M. Julien GAUBIL
M. Clément MARTINELLI

Encadrants :
M. Emmanuel DELLANDREA

Version du
28 octobre 2022

Résumé

L'Incremental learning est un scénario particulier d'entraînement de modèles de Machine Learning dans lequel le modèle est soumis à un flux continu de données. Il peut ainsi être amené à devoir apprendre de nouvelles tâches constamment. Un problème apparaît alors, particulièrement dans le cas des algorithmes de réseaux de neurones : les performances du réseau sur les précédentes tâches apprises chutent, c'est le Catastrophic Forgetting. Plusieurs méthodes ont été développées pour résoudre ce problème. Ce rapport effectue un état de l'art des algorithmes existants pour éviter le Catastrophic Forgetting dans les réseaux de neurones, puis approfondit un des modèles étudiés, Gradient Episodic Memory. Le choix du modèle sera justifié, puis une implémentation de GEM réalisée. Cette implémentation compare GEM à certains des autres modèles étudiés, sur les datasets MNIST rotations, MNIST permutations et CIFAR-100. Les résultats obtenus sont similaires aux résultats présentés dans l'état de l'art. Dans la configuration de l'implémentation, les résultats montrent que GEM est effectivement le modèle le plus efficace pour résoudre le problème du Catastrophic Forgetting parmi les modèles étudiés. GEM est ensuite approfondie au travers de ses variantes A-GEM et ϵ-soft-GEM. Des pistes d'amélioration des performances par amélioration de la sélectivité des mémoires sont enfin proposées.

Abstract

Incremental learning is a particular scenario for training Machine Learning models in which the model observes a continuous stream of data. It may thus be required to learn new tasks constantly. A problem then arises, particularly in the case of neural network algorithms : the performance of the network on the previously learned tasks drops, a phenomenon known as Catastrophic Forgetting. Several methods have been developed to overcome this issue. This document provides a literature review of the existing algorithms that aim at avoiding Catastrophic Forgetting in neural networks. It then further develops one of the models studied, Gradient Episodic Memory. The choice of the model will be justified. Afterwards, an implementation of GEM and the obtained results are discussed. This implementation compares GEM to some of the other models studied, on the MNIST rotations, MNIST permutations and CIFAR-100 datasets. The results obtained are similar to the results presented in the literature review.

In the configuration of the implementation, the results show that GEM indeed is the most efficient model to solve Catastrophic Forgetting among all the models studied. GEM is further developed by reviewing its variants A-GEM and ϵ-soft-GEM. Improvements are finally discussed by improving the selection of examples in episodic memories.

Remerciements

Nous tenons à adresser nos remerciements aux personnes qui nous ont aidé dans la réalisation de ce projet.

Dans un premier temps, nous remercions M. DELLANDREA, tuteur scientifique de notre projet, pour ses multiples conseils et son suivi tout au long de l'année. Son aide et son soutien précieux auront été d'une grande valeur pour nous.

Nous souhaitons aussi remercier notre conseiller en gestion de projet, M. DESSOMBZ, qui, par ses conseils nous aura permis de planifier et prévoir au mieux notre projet.

Nous remercions également M. PETIT, qui nous a guidé et apporté ses connaissances durant le début de notre projet.

Pour finir, nous remercions grandement l'*École Centrale de Lyon* pour nous avoir permis l'accès à toutes les installations nécessaires et nous avoir laissés dans les meilleures dispositions face à la réalisation de ce projet.

Table des matières

Table des figures

Glossaire

batch sous-groupe d'un dataset constitué d'instances de celui-ci (batch d'entraînement, de validation ...).

catastrophic Forgetting phénomène inhérent à l'incremental learning où les performances du réseau pour reconnaître des anciennes classes chutent lorsqu'il en apprend des nouvelles.

couche d'entrée première couche d'un réseau de neurones.

couche de sortie dernière couche d'un réseau de neurones. Dans un problème de classification, elle fournit une estimation de la classe à laquelle appartient l'instance en entrée.

couches cachées Comprise entre la couche d'entrée et de sortie, chacune contient N_i neurones. Les connexions entre les neurones de deux couches consécutives sont représentées par une matrice dont les éléments sont les poids du modèle. Chaque couche possède une fonction d'activation qui s'applique aux valeurs en entrée de chacun des neurones de la couche. Plusieurs choix pour la fonction d'activation sont possibles.

dataset ensemble de données (images, textes . . .) donné en entrée d'un réseau de neurones. Généralement, il est séparé en trois jeux de données : le set d'entraînement, le set de validation, et le set de test.

distillation transfert du savoir d'un grand réseau à un réseau plus petit. Le plus petit est entraîné en utilisant une fonction de coût modifiée qui fait en sorte que le grand et le plus petit réseau aient les mêmes résultats.

feature extraction processus de réduction de dimension d'un dataset. Par exemple, les instances du dataset sont données en entrée d'un réseau convolutif, et on extrait les features vectors des entrées, c'est à dire les sorties des couches de convolution.

fine-tuning elle modifie les paramètres d'un CNN déjà entraîné pour l'entrainer sur une nouvelle tâche. La couche de sortie est étendue (agrandie) avec des poids initialisés aléatoirement pour la nouvelle tâche. Le taux d'apprentissage utilisé dans la méthode de descente du gradient pour adapter les paramètres existants à la nouvelle tâche est petit. Parfois les poids des couches de convolutions sont fixés pour empêcher l'overfitting. Le Fine-tuning adapte les shared parameters θ_s pour les rendre plus discriminants sur la nouvelle tâche, tout en conservant le savoir acquis

lors de l'entraînement sur les tâches précédentes grâce au faible taux d'apprentissage.

fonction d'activation fonction d'un neurone qui prend en entrée la somme pondérée des entrées connectées au neurones et des poids des liasons, et fournit la sortie du neurone.

fully-connected chaque neurone d'une couche fully-connected est connecté à tous les neurones de la couche précedente.

hyperparamètre paramètre dont la valeur est utilisée pour contrôler le processus d'apprentissage (profondeur du réseau, coefficients de regularisation, taux d'apprentissage ...) Il peuvent être réglés par apprentissages, mais parfois ce n'est pas possible.

Incremental learning scénario particulier d'entraînement de modèles de Machine Learning dans lequel le modèle est soumis à un flux continu de données.

instance données pour lesquelles le réseau va faire uneprédiction de la classe à laquelle elles appartiennent.

label classes auxquelles appartiennent effectivement les instances.

Machine Learning l'utilisation et le développement de systèmes informatiques capables d'apprendre et de s'adapter sans suivre des instructions explicites, en utilisant des algorithmes et des modèles statistiques pour analyser et tirer des inférences à partir de modèles dans les données.

overfitting surapprentissage. En statistique, le surapprentissage est une analyse statistique qui correspond trop précisément à une collection particulière d'un ensemble de données. Ainsi, cette analyse peut ne pas correspondre à des données supplémentaires ou ne pas prévoir de manière fiable les observations futures. Le problème existe aussi en apprentissage automatique. De par sa trop grande capacité à capturer des informations, une structure dans une situation de surapprentissage aura de la peine à généraliser les caractéristiques des données. En d'autres termes, le modèle se souvient d'un grand nombre d'exemples au lieu d'apprendre à remarquer des fonctionnalités.

poids valeur permettant de chiffrer la valeur d'une liaison entre deux neurones.

problème de classification un certain nombre de classes est donné (par exemple pour un dataset d'images, les classes chiens, chats. . .). Il s'agit pour le réseau de neurones d'apprendre à associer chaque élément du dataset (souvent appeléinstance) à une classe, puis de généraliser, c'est-à-dire classifier des instances dudatasets qui ne lui ont jamais été fournies.

Relu fonction d'activation d'unité de rectification linéaire, elle est très utilisée. Sa formule est : $R(z) = max(0, z)$.

transfer learning champ de recherche de l'apprentissage automatique qui vise à transférer des connaissances d'une ou plusieurs tâches sources vers une ou plusieurs tâches cibles. Il peut être vu comme la capacité d'un système à reconnaître et appliquer des connaissances et des compétences, apprises à partir de tâches antérieures, sur de nouvelles tâches ou domaines partageant des similitudes..

Vision par ordinateur branche de l'intelligence artificielle dont le principal but est de permettre à une machine d'analyser, traiter et comprendre une ou plusieurs images prises par un système d'acquisition.

weight-decay ajout d'un terme pénalisant à la fonction de coût pour éviter l'overfitting.

Introduction

Les techniques utilisant des réseaux de neurones, et particulièrement les techniques dites d'apprentissage profond (deep learning) ont permis ces dernières années des avancées spectaculaires dans de nombreux domaines tels que la Vision par ordinateur par ordinateur.

Une grande application de ces techniques est la classification d'images, qui consiste à fournir au réseau des images appartenant chacune à des catégories différentes appelées classes. L'objectif du réseau de neurones est alors de classer les images qu'on lui donne en entrée dans un ensemble de classes prédéfini.

Dans un contexte réel d'application, il peut être nécessaire de permettre au modèle de continuer son apprentissage avec de nouvelles données disponibles en continu, et également d'apprendre de nouvelles classes. Ce scénario particulier d'apprentissage est appelé Incremental learning.

Dans les algorithmes de réseaux de neurones, cela entraîne l'apparition d'un phénomène qui se traduit lors de l'apprentissage d'une nouvelle tâche, par une chute brutale des performances sur les tâches précédemment apprises. Ce phénomène est appelé Catastrophic Forgetting.

Afin de tenter de résoudre le problème posé par le Catastrophic Forgetting, plusieurs algorithmes dits d'Incremental learning ont été mis au point. Ils ont pour but de permettre l'apprentissage d'un réseau de neurones dans un scénario d'Incremental learning tout en essayant de se rapprocher au plus près des performances d'un réseau de neurones entraîné dans un scénario classique.

Le but de ce projet est d'étudier les techniques pour pallier à ce problème d'Incremental learning, et d'en proposer des améliorations. La démarche comprend un état de l'art des méthodes existantes, puis une implémentation des méthodes jugées les plus prometteuses. Ensuite, une phase de recherche d'amélioration à apporter aux méthodes existantes sera menée. Ce projet est encadré par M. Emmanuel DELLANDREA, chercheur au laboratoire LIRIS. La gestion de projet est encadrée par M. Olivier DESSOMBZ, chercheur au LTDS.

Le plan de ce document suivra les grandes phases du projet : une première partie présentera les réseaux de neurones, l'Incremental learning et le Catastrophic Forgetting, puis l'état de l'art réalisé. Les résultats des implémentations des méthodes existantes déterminées comme intéressantes seront ensuite exposés. La troisième partie se consacrera à la phase d'amélioration des méthodes d'Incremental learning existantes, tandis qu'une dernière partie évoquera les perspectives à l'issue de ce projet.

1 Prérequis et état de l'art de l'Incremental learning

La sous-section suivante se consacre à une rapide introduction aux réseaux de neurones artificiels. Le lecteur familier de ces notions pourra directement se rendre à la sous-section qui traite de l'Incremental learning.

1.1 Introduction aux réseaux de neurones artificiels

Les réseaux de neurones sont des algorithmes de Machine Learning qui ont pour vocation de reproduire le fonctionnement de l'apprentissage par un cerveau biologique. Pour cela, les réseaux de neurones sont composés d'un élément de base que sont les neurones articiels, qui visent à imiter le fonctionnement des neurones biologiques.

Sur les figures qui suivent sont représentés les schémas d'une synapse biologique et d'un neurone artificiel.

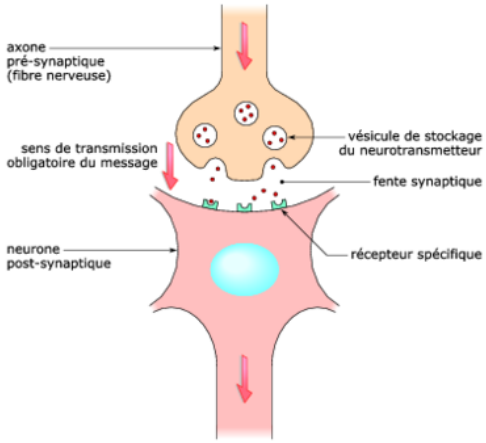


FIGURE 1 – Schéma d'une synapse biologique

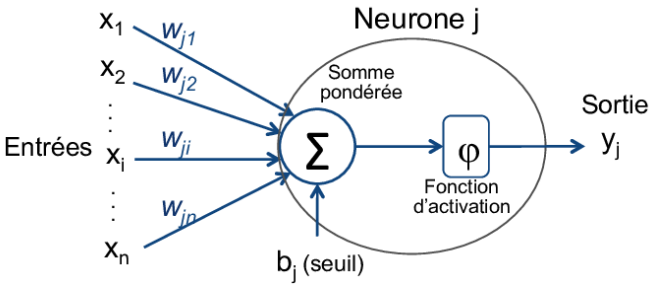


FIGURE 2 – Schéma d'un neurone artificiel

Sur la figure 1, l'espace entre les deux composants représente l'espace entre deux neurones. La connexion entre les deux neurones se fait via des transmetteurs chimiques qui propagent l'information d'un neurone à l'autre. Sur la figure 2, un neurone artificiel est représenté. Les signaux qu'il reçoit (symbolisés par les flèches bleues) sont les informations qui lui sont transmises depuis d'autres

neurones. La connexion entre deux neurones artificiels est modélisée par un poids (les $w_{i,j}$ sur la figure) : c'est un nombre réel qui pondère le signal reçu par le neurone j depuis le neurone i.

Les poids modélisent la synapse et permettent donc de propager l'information d'un neurone à un autre. Les signaux pondérés reçus par un neurone en provenance d'autres neurones sont ensuite sommés puis on leur applique une fonction d'activation propre au neurone.

Les neurones sont ensuites rassemblés en couches. Les neurones d'une même couche ont alors tous la même fonction d'activation. On distingue 3 types de couches :

— *Couche d'entrée* : en bleu sur l'image 3, elle possède autant de neurones que les données en entrée ont de variables. Exemple : images 20x10 en entrée = 200 neurones sur la couche d'entrée.

— *Couche de sortie* : en vert sur l'image 3. Elle contient la contient K neurones, où K est le nombre de classes du problème.

— *Couches cachées* : en orange sur l'image 3. Comprise entre la couche d'entrée et de sortie, chacune contient N_i neurones. Les connexions entre les neurones de deux couches consécutives sont représentées par une matrice dont les éléments sont les poids du modèle. Chaque couche possède une fonction d'activation qui s'applique aux valeurs en entrée de chacun des neurones de la couche. Plusieurs choix pour la fonction d'activation sont possibles.

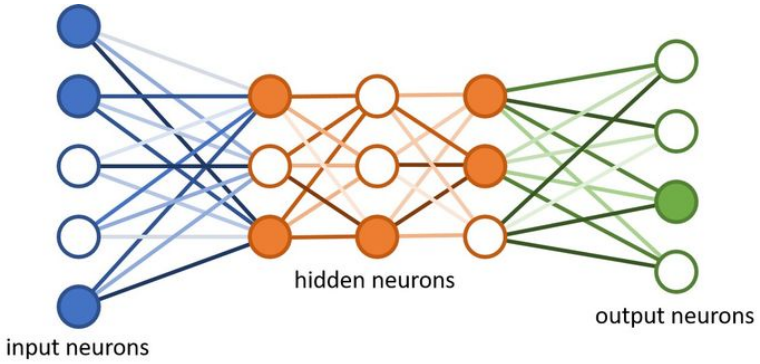


FIGURE 3 – Schéma simplifié d'un réseau de neurones

Les paramètres d'un réseau de neurones sont donc les poids qui représentent les connexions entre ces neurones. Les autres paramètres tels que le nombre de couches ou le nombre de neurones sur chaque couche sont appelés les hyperparamètres du modèle.

D'un point de vue technique, il est également possible de distinguer deux types de couches :

— les couches dites *fully-connected*, représentées sur la figure 4. Chaque neurone d'une couche fully-connected est connecté à tous les neurones de la couche précedente ;
— les couches pour lesquelles chaque neurone n'est connecté qu'à une partie des neurones de la couche précédente. C'est le cas des couches dites de convolution, représentées figure 5, utilisées dans les réseaux de neurones convolutifs[1].

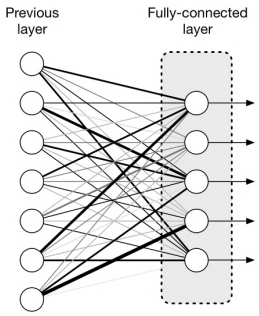


FIGURE 4 – Couche fully-connected dans un réseau de neurones

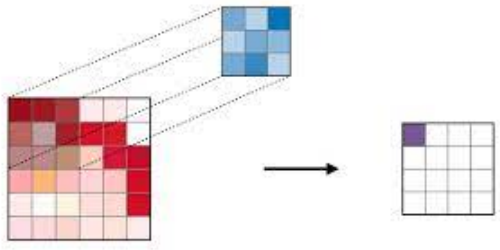


FIGURE 5 – Couche de convolution dans un réseau de neurones

Les réseaux de neurones sont souvent appliqués à des problèmes de classification : pour un dataset, un certain nombre de classes est donné (par exemple pour un dataset d'images, les classes chiens, chats…). Il s'agit pour le réseau de neurones d'apprendre à associer chaque élément du dataset (souvent appelé instance) à une classe, puis de généraliser, c'est-à-dire classifier des instances du datasets qui ne lui ont jamais été fournies.

Dans le cadre d'un problème de classification, le dataset se compose de deux types de données :

1. Le principe d'un réseau de neurones convolutif est détaillé en annexe

— les instances, qui sont les données pour lesquelles le réseau va faire une prédiction de la classe à laquelle elles appartiennent.
— les labels, qui sont les classes auxquelles appartiennent effectivement les instances.

Avec ces deux types de données, une fonction appelée fonction de coût est définie pour évaluer le modèle. Cette fonction a pour paramètre les poids du réseau et a pour propriété d'augmenter lorsque le réseau commet des erreurs de classification. Elle se met typiquement sous cette forme :

$$L(\theta) = \sum_{instance \in dataset} l(\theta, y)$$

où θ représente les poids du modèle, y est le label d'une instance donnée, et l une fonction appelée *loss* qui compare la prédiction du réseau pour une instance donnée à son label (*ie* sa vraie classe). Plusieurs choix de loss sont possibles pour composer une fonction de coût.

Dans le cadre d'un problème de classification classique, l'apprentissage se découpe en 2 phases :

1. **l'entraînement** : pour les instances du dataset, le réseau prédit une classe. Les prédictions du réseau et les labels des instances permettent de calculer la fonction de coût. Avec l'algorithme de descente du gradient, les poids du réseau sont modifiés jusqu'à converger vers des valeurs qui minimisent la fonction de coût sur le dataset.
2. **la généralisation** : une fois entraîné, le réseau est prêt à faire des prédictions sur des instances dont on ne connaît pas le label.

1.2 Incremental learning & Catastrophic Forgetting

L'**Incremental learning** est un type particulier d'algorithme de Machine Learning. Au lieu d'avoir 2 phases (entraînement puis généralisation) comme un algorithme classique, le dataset est subdivisé en plusieurs sous ensembles appelés batchs. Le modèle de Machine Learning sera d'abord entraîné sur le premier batch uniquement, puis il sera entraîné sur le deuxième batch uniquement, et ainsi de suite.
La phase d'apprentissage est donc décomposée en plusieurs temps au lieu d'être effectuée en une seule fois sur l'ensemble du dataset. Il est cependant souhaitable qu'à tout instant (même lorsque le modèle n'a pas été entraîné sur tous les batchs du dataset), le modèle présente des performances correctes, au moins pour les

batchs sur lesquels il a déjà été entraîné. L'image 6 représente l'ajout de nouvelles classes à chaque nouveau batch.

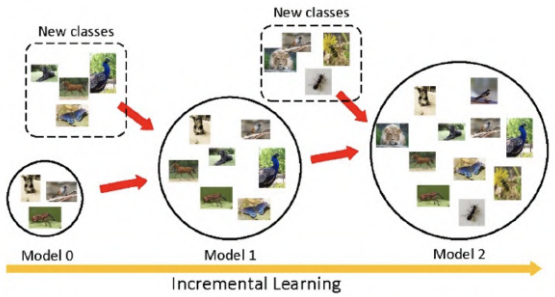


FIGURE 6 – Le principe de l'Incremental learning

Le cadre de notre étude sera l'application de l'Incremental learning aux réseaux de neurones, pour des problèmes de classification.

L'Incremental learning est un scénario intéressant car il se rapproche des conditions d'utilisation réelles d'un modèle de Machine Learning. Toutes les données sont rarement disponibles dès la phase d'entraînement du modèle, on voudrait donc pouvoir entraîner de nouveau un modèle si de nouvelles données devenaient disponibles.

Par exemple, pour un problème de classification d'images, des données sur un nouvel objet inconnu du réseau pourraient devenir disponibles. Cela constituerait alors une nouvelle classe. On souhaiterait entraîner de nouveau le réseau afin qu'il soit capable de reconnaître à la fois des objets appartenant aux classes sur lesquelles il a déjà été entraîné, et les objets de la nouvelle classe. Les applications de l'Incremental learning sont donc nombreuses, en particulier dans la reconnaissance d'image pour la robotique.

Pourtant, entraîner un réseau de neurones sur les différents batchs du dataset de la même manière que dans un problème classique de Machine Learning mène à un phénomène important : le **Catastrophic Forgetting**. En effet, en voulant apprendre au moins les nouvelles classes/instances, le réseau va petit à petit oublier les premières classes/instances apprises. Cela arrive particulièrement quand des poids importants pour la reconnaissance des anciennes classes sont modifiés pour minimiser la fonction de coût sur les nouvelles classes.

Comme nous pouvons le voir sur l'image 7, lors de l'apprentissage d'une deuxième tâche, les poids optimisés par le premier entraînement pour l'exécution de la première tâche sont modifiés. L'image de gauche représente le cheminement de la modification des poids au cours des deux apprentissages. Les cercles concentriques pour les tâches A et B représentent les zones pour lesquelles les poids mènent à une fonction de coût faible pour chacune des tâches. On observe bien que l'on quitte la zone de coût faible pour la tâche A lors de l'entraînement sur la tâche B. Cela se traduit dans le graphique de droite par une chute de performances pour la tâche A au profit de la tâche B, avec une augmentation de la fonction de coût pour A et une diminution pour B.

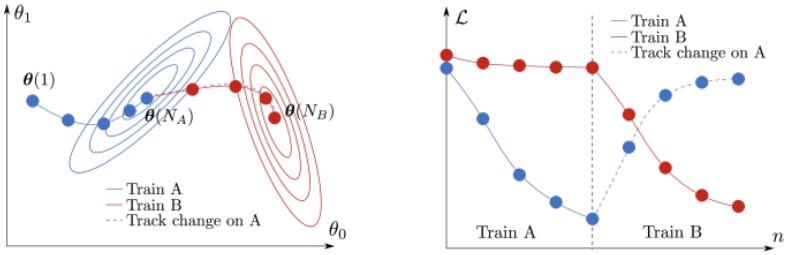


FIGURE 7 – Illustration du Catastrophic Forgetting

Il s'agira donc de déterminer des modèles qui permettent de réaliser un apprentissage dans le cadre donné par l'Incremental learning en évitant le Catastrophic Forgetting. Le modèle devra pouvoir être entraîné sur de nouveaux batchs sans que cela n'entraîne de chute de performances sur les classes/instances des batchs sur lesquelles il a déjà été entraîné. Dans le même temps, les performances sur les nouveaux batchs appris devront être bonnes.

1.3 Etat de l'art des méthodes existantes

Il existe de nombreuses méthodes ayant été créées pour résoudre le problème du Catastrophic Forgetting et améliorer la performance des algortihmes d'Incremental learning. Ces méthodes peuvent être classées en 3 catégories selon la figure 12 :

— **Architectural strategies :** ce sont des stratégies qui affectent la structure des réseaux de neurones pour adresser le problème de Catastrophic Forgetting. Des couches, des fonctions d'activation et / ou des stratégies de blocages de poids spécifiques sont utilisées.

— **Regularization strategies :** la fonction de coût est étendue avec des termes consolidant des poids importants pour conserver les précédents apprentissages : $L^{new} = L^{old} + \lambda * L^{regul}$

— **Reharsal strategies :** les instances des batchs précédents sont périodiquement réinjectées dans le modèle, pour renforcer les connexions des précédents apprentissages. Une approche simple consiste à stocker une partie des données d'entraînement précédentes et les mixer avec de nouvelles instances pour un futur apprentissage.

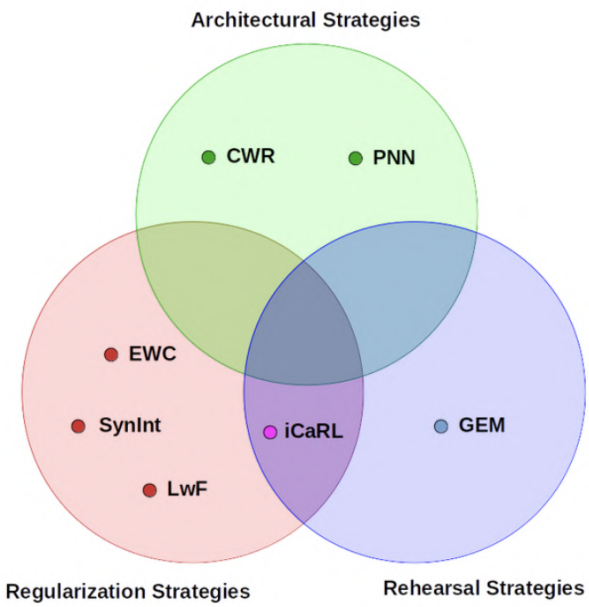


FIGURE 8 – Différents types de méthodes en Incremental learning

Chaque méthode a été étudiée afin d'avoir une vision globale, de leurs points positifs et négatifs, afin de les comparer et de sélectionner la (ou les) méthode à apprendir.

1.3.1 Présentation de quelques datasets

Avant la présentation des articles scientifiques et méthodes qui en découlent, il convient de présenter quelques datasets utilisés pour entraîner des modèles d'Incremental learning.

Un dataset est un ensemble de données (images, textes ...) fournit en entrée d'un réseau de neurones. Généralement, il est séparé en trois jeux de données : le set d'entraînement, le set de validation, et le set de test.
Dans le cas d'un apprentissage supervisé, les set d'entraînement et de validation contiennes des instances (objets) et des labels (les classes auxquelles ils appartiennent). Les datasets sont notamment utilisés pour tester et comparer les performances des différentes méthodes.

— *CIFAR 100* : dataset d'images composé de 20 superclasses (insectes, fleurs, poissons...) et de 100 classes (apples, bees...). Le dataset est subdivisé en un nombre donné de batchs (par défaut 10), qui représentent chacun une tâche à apprendre pour le modèle. Les batchs sont appris successivement (apprentissage sur le batch 1, puis on ré-entraîne sur le batch 2 et ainsi de suite...).

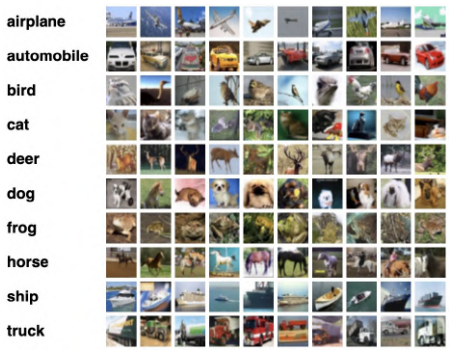


FIGURE 9 – Extrait du dataset CIFAR100

— *MNIST rotations* : dataset d'images issu de MNIST qui est composé d'images de digits de 0 à 9. MNIST rotations est composé d'un nombre donné de batchs qui représentent les tâches apprises successivement par le modèle. Chaque batch contient toutes les images de MNIST auxquelles on a appliqué une rotation d'un même angle $\theta \in [0, 90]^\circ$ déterminé de manière aléatoire. Chaque batch ne diffère donc des autres que par le degré de la rotation qui est appliqué aux images de MNIST.

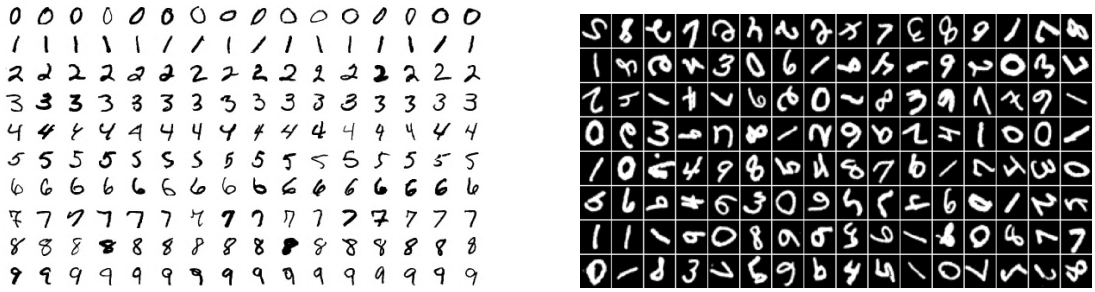


FIGURE 10 – Extraits du dataset MNIST (à gauche) et MNIST rotations (à droite)

— *MNIST permutations* : dataset d'images issu de MNIST qui est composé d'images de digits de 0 à 9. MNIST permutations est composé d'un nombre donné de batchs qui représentent les tâches apprises successivement par le modèle. Chaque batch contient toutes les images de MNIST auxquelles on a appliquée une même permutation des pixels. Chaque batch ne diffère donc des autres que par la permutation qui est appliquée aux pixels des images de MNIST.

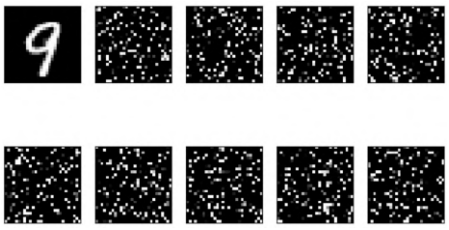
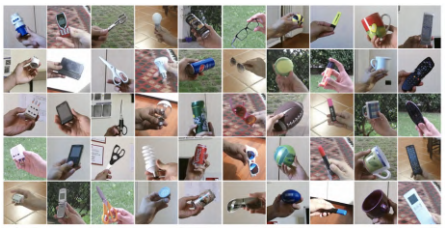


FIGURE 11 – Extrait du dataset MNIST permutations

— *CoRe50* : il est composé d'images acquises par des sessions vidéo sur 50 objets domestiques regroupés en 10 classes d'objets.



1.3.2 Copy Weights with Re-init (CWR)

Le modèle suivant est présenté dans l'article "CORe50 : a New dataset and Benchmark for Continuous Object Recognition" par Vincenzo Lomonaco, Davide Maltoni paru en 2017 [1].
En plus d'introduire CWR, il introduit un formalisme propre à l'Incremental learning ainsi que le dataset CORe50.

En considèrant un dataset d'Incremental learning constitué de N batchs $\{1, ..., N\}$. Le modèle va être entraîné de manière incrementale, d'abord sur le batch 1, puis le batch 2, jusqu'au batch N.

3 scénarios d'Incremental learning sont définis :
— **New instances (NI)** : de nouveaux exemples de classes connues sont présentés au réseau ;
— **New classes (NC)** : des exemples appartenant à de nouvelles classes sont présentés au réseau ;
— **New classes and instances** : des nouveaux exemples appartenant à d'anciennes classes et des exemples appartenant à de nouvelles classes sont présentés.

Les modèles CWR s'appliquent à des réseaux de neurones convolutifs [1]. Les poids des dernières couches (fully-connected) sont dupliqués en deux versions appelées **tw** et **cw**. Les poids **cw** sont des copies des poids **tw**, qui sont utilisés pour l'entraînement.
— Au premier batch d'entraînement, les **cw** sont initialisés à 0 et les **tw** sont initialisés aléatoirement. Les poids des couches de convolution sont initialisés aléatoirement de la même manière. Le réseau est entraîné sur le premier batch. A la fin de l'entraînement, les poids **tw** sont copiés dans **cw**. Les performances du réseau sont alors évaluées sur le dataset de test.
— Ensuite, à chaque nouveau batch, les poids **tw** et les poids des couches de convolution sont réinitialisés aléatoirement. Le réseau est alors entraîné sur le nouveau batch, puis les poids **tw** ayant des valeurs significatives sont copiés dans **cw**.

Les performances de ce modèle sont comparées avec des méthodes cumulatives et naïves. La méthode naïve consiste à entraîner un réseau de neurones sur les batchs les uns après les autres, sans stratégie d'Incremental learning. La

1. Voir annexe des CNN

méthode cumulative consiste à entraîner le réseau avec le nouveau batch ajouté à tous ceux qui lui ont déjà été présentés.

Les résultats sur le dataset CoRe50 sont présentés sur la figure 12 :

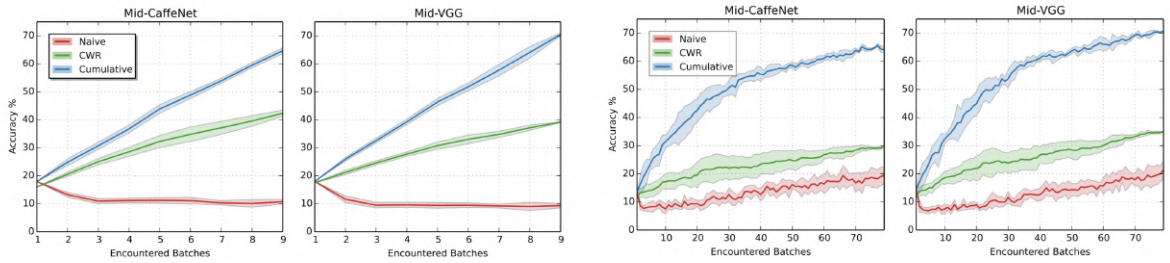


FIGURE 12 – Résultats de CWR en comparasion avec une méthode naïve et cumulative, pour le cas NC à gauche et NIC à droite

Pour le cas NC, la méthode naïve obtient 10% de précision moyenne, tandis que CWR obtient 40% la méthode cumulative obtient 65%.
Pour le cas NIC : 20% de précision moyenne en méthode naïve, 35% en CWR et 70% en cumulative.

CWR est donc plus efficace qu'une méthode naïve, mais ses performances restent assez faibles en comparaison d'une méthode cumulative présentée comme une borne supérieure. CWR a donc vocation à être un point de référence pour les modèles d'Incremental learning plus évolués qui seront développés suite à sa création.

1.3.3 Progressive Neural Networks (PNN)

Le modèle suivant est présenté dans l'article "Progressive Neural Networks" par Andrei A. Rusu, Neil C. Rabinowitz & al, paru en 2016 [2].
Cet article introduit les Progressive Neural Networks, une méthode de type architectural. Elle permet d'éviter le Catastrophic Forgetting et d'apprendre une nouvelle tâche plus rapidement par transfer learning[1] avec les tâches déjà apprises L'objectif visé est que le modèle soit capable de résoudre K tâches indépendantes à l'issue de son entraînement.

1. Voir lexique

Le réseau dispose d'un ensemble de tâches à apprendre successivement. A la première tâche, un réseau de neurones est initialisé aléatoirement et entraîné sur la première tâche. A la tâche suivante, un nouveau réseau de neurones identique en termes de structure au premier est créé et initialisé aléatoirement. Il est alors entraîné sur la deuxième tâche.

Chaque couche i du réseau de la 2^{eme} tâche est connectée à la sortie de la couche précédente $i-1$, mais aussi à la sortie de la couche $i-1$ du réseau de la première tâche. Ces dernières connexions sont appelées connexions latérales, représentées sur la figure 13.

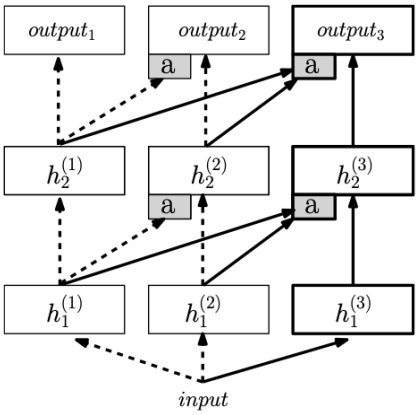


FIGURE 13 – Structure d'un modèle PNN à 3 réseaux retenus

Les PNN retiennent donc un ensemble de réseaux de neurones entraînés successivement, chacun sur une tâche spécifique. Pour un réseau entraîné à une tâche, les connexions latérales sont réalisées avec les réseaux entraînés sur les tâches précédentes afin d'incorporer leur « savoir » au modèle en cours d'entraînement. Il y a alors, pour une couche donnée d'un des réseaux entraîné, deux types de connexions réalisées par la couche. D'une part, les connexions de la couche avec sa couche précédente dans le même réseau. D'autre part, les connexions latérales réalisées avec sa couche précédente dans les autres réseaux précédemment entraînés.

Cette méthode comporte plusieurs avantages : il n'est pas nécessaire de stocker les exemples d'entraînement, puisque la même entrée est appliquée à tous les réseaux retenus (cf figure 13). De plus, cette structure permet de consolider les connaissances déjà acquises lors de l'apprentissage des nouvelles tâches.

Cependant, cette solution paraît coûteuse en mémoire et calculs, le nombre de paramètres à stocker ainsi que les calculs à réaliser augmentant beaucoup à chaque tâche apprise, du fait du double type de connexions réalisées par chacunes des couches.

1.3.4 Elastic Weight Consolidation (EWC)

Le modèle suivant est présenté dans l'article "Overcoming Catastrophic Forgetting in neural networks" par James Kirkpatrick, Razvan Pascanu & al, paru en 2018 [3].

Cet article introduit EWC, une des premières méthodes à avoir significativement réduit le Catastrophic Forgetting. Cette méthode utilise une stratégie de régularisation. Elle a pour objectif de renforcer les poids importants pour les précédentes classes apprises sans les figer toutefois, d'où le terme "elastic". En considérant un dataset d'Incremental learning constitué de N batchs $\{1, ..., N\}$, le modèle va être entraîné de manière incrémentale, d'abord sur le batch 1, puis le batch 2, jusqu'au batch N.

Un modèle EWC détermine pour chaque batch les poids qui sont importants dans son apprentissage. Il détermine aussi la manière de les prendre en compte, dans une approche probabiliste.
Lors de l'entraînement sur un batch B, afin de ne pas oublier l'entraînement déjà effectué sur un batch A, une pénalité quadratique pour retenir les poids importants pour A est ajoutée à la fonction de coût :

$$L(\theta) = L_B(\theta) + \frac{\lambda}{2} \sum_i F_i(\theta_i - \theta^*_{A,i})^2$$

où L_B la fonction de coût pour le batch B, et le terme de droite représente l'écart quadratique entre les poids actuels du réseau et les poids idéaux pour le batch A, notés $\theta^*_{A,i}$[1]. λ est un hyperparamètre qui représente l'importance de la tâche A comparée à la tâche B.

Cette forme de fonction de coût permet donc d'apprendre sur le batch B en pénalisant les écarts forts pour les poids importants à l'apprentissage effectué sur le batch A.

1. Ce sont les poids obtenus à l'issue de l'entraînement sur le batch A

Sur la figure 14, l'ellipse grise représente l'espace des poids qui permettent d'obtenir un coût faible sur un batch A. L'ellipse beige représente l'espace des poids permettant d'obtenir un coût faible sur un batch suivant, B. Avec la méthode EWC, la modification des poids est faite de manière à ce qu'à l'issue de l'entraînement sur B, les poids se situent toujours dans l'espace permettant d'obtenir un coût faible sur A

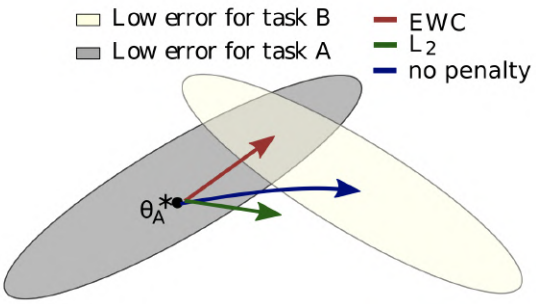


FIGURE 14 – EWC permet d'apprendre la tâche B sans oublier la tâche A

Les résultats seront présentés dans la partie 2.2, sur les datasets CIFAR 100, MNIST permutations et MNIST rotations. Cette méthode (datant de 2016) ne sera pas retenue par la suite du fait de ses faibles performances en comparaison d'autres méthodes introduites entre 2016 et 2017. Cette méthode peut être utilisée en complément d'autres méthodes plus efficaces et d'un autre type (reharsal, architectural..), pour limiter les interférences.

1.3.5 Synaptic Intelligence (SynInt)

Le modèle suivant est présenté dans l'article "Continual Learning Through Synaptic Intelligence" par Friedemann Zenke, Ben Poole & al, paru en 2017 [4].

Cet article introduit la méthode SynInt de type régularisation, qui s'inspire de la biologie. L'apprentissage pour un cerveau utilise des synapses [1]. Chaque synapse est un mécanisme complexe qui tire parti de la complexité moléculaire pour résoudre plusieurs tâches à la fois. Les synapses des réseaux de neurones artificiels, en revanche, sont paramétrées par un scalaire unique (le poids).

Le raisonnement est donc de complexifier la modélisation de la synapse qui est faite pour qu'elle soit capable de calculer sa propre importance dans

1. voir l'introduction aux réseaux de neurones

l'apprentissage des tâches passées, et donc sa contribution locale à la fonction de coût. Lorsqu'une nouvelle tâche est apprise, le changement des paramètres des synapses artificielles importantes pour la conservation de l'apprentissage précédent est pénalisé, maintenant ainsi l'apprentissage des tâches passées.

La fonction de coût du modèle SynInt pour une tâche μ est définie par :

$$L^*_\mu = L_\mu + c \sum_k \Omega^\mu_k (\theta^{\mu-1}_k - \theta^\mu_k)^2 \quad (1)$$

Le terme de gauche correspond à une fonction de coût classique pour la tâche μ. Le second terme est le terme de regularisation. Il comprend notamment c, le coefficient de régularisation (hyperparamètre), et Ω^μ_k, la mesure de l'importance du poids θ_k pour la tâche μ. Les changements trop élevés des valeurs des poids sont donc pénalisés dans cette formulation, afin de ne pas modifier significativement les poids importants à la réalisation de la tâche μ

.

Les résultats de cette SynInt sur le dataset MNIST permutations sont présentés sur la figure 15.

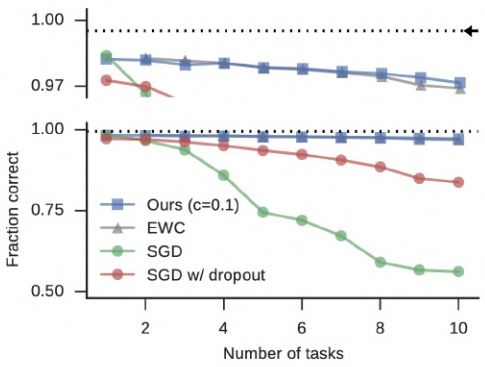


FIGURE 15 – Résultats de Synint (en bleu) sur le dataset MNIST permutations, en comparaison avec EWC (gris), Descente du gradient classique (vert) et Descente du gradient avec regularisation des poids (rouge)

Synint obtient bien des résultats comparables à EWC, ce qui s'explique par la similarité des deux approches. Pour les mêmes raisons qu'EWC, SynInt ne sera donc pas approfondie dans la suite de l'étude (résultats trop faibles par rapport à d'autres méthodes introduites la même année, en 2017).

1.3.6 Learning without Forgetting (LwF)

Le modèle suivant est présenté dans l'article "Learning Without Forgetting" par Zhizhong Li, Derek Hoiem paru en 2017 [5].

Cet article introduit le modèle LwF, qui utilise une stratégie de régularisation. Il s'applique à des réseaux convolutifs déjà entraînés sur de grands datasets, pour apprendre de nouvelles tâches sur d'autres datasets moins fournis. Lors du nouvel entraînement sur le petit dataset, les instances du gros dataset sur lequel le CNN a déjà été entraîné ne sont plus disponibles. LwF vise alors à préserver les performances sur le gros dataset tout en obtenant de bonnes performances sur le petit dataset. Ce modèle est surtout appliqué en computer vision et tracking.

Le modèle LwF introduit 3 types de poids :

— Les poids des couches de convolution, appelés shared parameters θ_s ;
— Les paramètres spécifiques aux tâches précédemment apprises, θ_0, situés dans les couches fully-connected, particulièrement la couche de sortie du réseau ;
— Les paramètres spécifiques aux nouvelles tâches à apprendre, θ_n. Ce sont des poids initialisés aléatoirement correspondant à des neurones ajoutés à la couche de sortie.

Les poids θ_0 et θ_n agissent donc comme des classifieurs qui opèrent sur les features[1] générées par les couches de convolution dont les poids sont les shared parameters θ_s .

En annexe sont présentées les notions de *weight-decay*, *distillation*, *feature extraction*, *fine-tuning* qui sont utilisées dans cette méthode. L'entraînement d'un modèle LwF se déroule selon les étapes suivantes :

1. Enregistrer les sorties pour les instances du nouveau dataset du CNN précédemment entraîné ;
2. Ajouter des neurones à la couche de sortie du CNN pour créer les poids θ_n initialisés aléatoirement. Le réseau est d'abord entraîné en figeant les poids θ_s et θ_0 ;
3. Le réseau est ensuite entraîné avec tous les poids θ_n, θ_s et θ_0 jusqu'à la convergence.

1. Voir annexe des CNN

La fonction de coût pour l'entraînement du réseau est ainsi définie :

$$L = L_{new} + \lambda_0 L_{old} + R$$

où L_{new} est une fonction de coût définie sur les instances du nouveau dataset, L_{old} est une fonction de coût définie sur les sorties du CNN originel pour les instances du nouveau dataset, et R est un terme de régularisation fonction des poids θ_0, θ_n et θ_s[1].

LwF performe mieux ou de manière similaire que les méthodes d'Incremental learning avec lesquelles elle est comparée (fine-tuning, Feature Exctraction...). Cependant, une différence importante est observée entre les performances sur le grand dataset et le petit dataset, qui traduit la différence qui peut exister entre les données de ces deux datasets.

Il est intéressant que LwF propose une méthode pour tenter d'entraîner un réseau convolutif sur deux datasets pouvant comporter des différences notables, car cela correspond à un cas d'utilisation proche de la réalité. Néanmoins, l'absence de code disponible pour l'implémentation nous a orienté vers d'autres approches qui traitent de problèmes plus restreints (car ils traitent de problèmes où un même dataset est subdivisé en plusieurs batchs). Ces modèles présentent de meilleurs résultats en précision sur des problèmes moins difficiles que celui traité par LwF. L'utilisation de tels modèles nécessite donc une phase de data engineering plus importante que LwF, qui peut s'appliquer à deux datasets significativement différents (avec une baisse de performance).

1.3.7 Gradient Episodic Memory (GEM)

Le modèle suivant est présenté dans l'article "Gradient Episodic Memory for Continual Learning" par David Lopez-Paz, Marc'Aurelio Ranzato, paru en 2017 [6].

L'article introduit la méthode GEM, qui est une méthode de type Reharsal. ELle est obtient des résultats très performants, c'est pour cela que nous allons plus la détailler que les précédentes.
Le modèle GEM introduit 3 métriques pertinentes pour l'évaluation d'un modèle

1. λ_0 est un hyperparamètre du modèle, coefficient de l'importance de l'apprentissage sur l'ancien dataset par rapport au nouveau.

d'Incremental learning.

La précision

La précision est la métrique classiquement utilisée pour évaluer les modèles :

$$ACC = \frac{1}{T}\sum_{i=1}^{T} acc_{T,i}$$

où T est le nombre de batchs du dataset, et $\forall i \in \{1, ..., T\}$, $acc_{T,i}$ est la précision sur le batch i à la fin de l'entraînement sur tous les batchs. Cette métrique calcule la précision moyenne du modèle.

Backward Transfer

Le *Backward Transfer* mesure l'influence de l'apprentissage de chaque batch i sur la précision sur les batchs k précédemment appris ($k \leq i$). Cette métrique traduit donc la présence de Catastrophic Forgetting, caractérisé par un Backward Transfer très négatif. Pour cela, le Backward Transfer est défini par la moyenne sur l'ensemble des batchs de la différence entre la précision sur le batch calculée dans deux conditions différentes :

— à l'issue de l'entraînement sur tous les batchs
— à l'issue de l'entraînement sur le batch en question.

$$BWT = \frac{1}{T-1}\sum_{i=1}^{T-1} (acc_{T,i} - acc_{i,i})$$

où T est le nombre de batchs du dataset, et $\forall i \in \{1, ..., T\}$, $acc_{T,i}$ est la précision sur le batch i à la fin de l'entraînement sur tous les batchs, $acc_{i,i}$ est la précision sur le batch i à la fin de l'entraînement sur ce batch.

De par sa conception, la méthode GEM minimise le *Backward Transfer* à chaque itération de l'algorithme de Descente du Gradient.

Forward Transfer

Le *Forward Transfer* mesure l'influence de chaque batch i sur la précision sur les batchs k pas encore appris ($k > i$). Cela correspond à l'influence de chaque

batch sur le *zero-shot learning*, qui est la capacité d'un modèle à effectuer une tâche pour laquelle il n'a pas été entraîné. Le Forward Transfer est défini par la moyenne sur chaque batch i de la différence entre la précision sur ce batch calculée dans deux conditions différentes :

— précision calculée sur le modèle juste avant qu'il ne soit appris (*ie* au moment où le batch $i-1$ est le dernier appris)
— précision calculée pour un modèle entraîné à aucune tâche et initialisé aléatoirement.

$$FWT = \frac{1}{T-1}\sum_{i=2}^{T}(acc_{i-1,i} - acc_{al_i})$$

où T est le nombre de batchs du dataset, et $\forall i \in \{1, ..., T\}$, $acc_{i-1,i}$ est la précision sur le batch i juste avant qu'il ne soit appris (à la fin de l'entraînement sur le batch $i-1$). acc_{al_i} est la précision sur le batch i pour un modèle initialisé aléatoirement.

Le modèle GEM

On considère un dataset d'Incremental learning constitué de N batchs $\{1, ..., N\}$. Le modèle va être entraîné de manière incrementale, d'abord sur le batch 1, puis le batch 2, jusqu'au batch N.

GEM est un modèle d'Incremental learning de type Rehearsal : son principe est de constituer des mémoires $\{M_1, ..., M_N\}$ au cours de l'apprentissage, pour chaque batch appris. Pour un batch $k \in \{1, ..., N\}$, la mémoire M_k est un sous-ensemble des instances du batch k sur lequel le modèle aura été entraîné. La taille des mémoires est fixée à l'avance, c'est donc un hyperparamètre du modèle. Dans le modèle GEM, pour une taille de mémoire donnée T et un batch donné $k \in \{1, ..., N\}$, les T instances du batch k conservées dans la mémoire M_k sont les T dernières instances du batch k fournies au modèle lors de l'apprentissage sur le batch k.

L'algorithme de descente du gradient permet de minimiser la fonction de coût L de paramètre $\theta = (\theta_1, ..., \theta_n)^t \in \mathbb{R}^n$ où les $\theta_1, ..., \theta_n$ sont les poids du modèle. A chaque itération de l'algorithme, les poids du modèles sont mis à jour

simultanément [1] en suivant la formule :

$$\forall i \in \{1, ..., n\}, \theta_i = \theta_i - \alpha \frac{\partial L}{\partial \theta_i}(\theta)$$

où α est un paramètre fixé (hyperparamètre du modèle), appelé taux d'apprentissage (*learning rate*).

Il est donc nécessaire de calculer, à chaque itération de l'algorithme, le gradient g de la fonction de coût [2] afin d'utiliser ses composantes (les $\frac{\partial L}{\partial \theta_i}(\theta)$) pour mettre à jour les poids du modèle et minimiser la fonction de coût L.

Le principe de GEM est de fixer une contrainte sur le gradient g à chaque étape de l'itération de la descente du gradient. Si g ne satisfait pas cette contrainte, il sera alors remplacé par un vecteur $\widetilde{g}$ qui la satisfait.

Lors de l'entraînement sur le batch $k \in \{1, ..., N\}$, la condition imposée à chaque itération de l'algorithme de descente du gradient est :

1. minimiser la fonction de coût calculée sur le batch k
2. diminuer la fonction de coût calculée sur chacune des mémoires des batchs $i < k$ (sur lesquels le modèle a déjà été entraîné), par rapport à l'itération précédente de l'algorithme de descente du gradient

Cette condition permet de s'assurer que l'entraînement sur un batch ne perturbe pas les résultats de l'entraînement effectué sur les batchs précédents. Avec cette condition, le Backward Transfer du modèle est nécessairement positif ou nul. Alors, cela empêche l'apparition du Catastrophic Forgetting, qui se caractérise par un Backward Transfer très fortement négatif.

On considère maintenant l'entraînement sur un des batchs $k \in \{1, ..., N\}$. On note, $\forall i \in \{1, ..., k-1\}, g_i$ le gradient de la fonction de coût du modèle calculé sur la mémoire M_i du batch déjà appris $i < k$. La reformulation mathématique de la contrainte imposée donne le problème :

$$\min_{\widetilde{g}} \frac{1}{2} \|g - \widetilde{g}\| \text{ où } \forall i \in \{1, ..., k-1\}, \quad \langle \widetilde{g}, g_i \rangle \geq 0$$

Le calcul donne :

$$\widetilde{g} = g + G^t v^*$$

1. jusqu'à convergence du modèle vers le minimum de L
2. par une méthode dite de rétropropagation du gradient

où $G = -(g_1, ..., g_{k-1})$ et v^* est la solution du problème :

$$\min_{v \geq 0} \frac{1}{2}(G^t v)^t (G^t v) + (Gg)^t v$$

Les mémoires sont donc utilisées à chaque itération de l'algorithme de descente du gradient pour calculer le gradient de la fonction de coût sur chacune des mémoires. Ces gradients seront utilisés dans la mise à jour des poids du modèle dans l'algorithme, si le gradient g calculé sur le batch en cours d'apprentissage ne permet pas de respecter la condition imposée.

Les résultats des expériences menées sur GEM seront étudiés dans la partie 2/

1.3.8 incremental Classifier and Representation Learning (iCarl)

Le modèle suivant est présenté dans l'article "iCaRL : Incremental Classifier and Representation Learning" par Sylvestre-Alvise Rebuffi, Alexander Kolesnikov & al, paru en 2017 [7].

Cet article introduit la méthode iCarl, une méthode très intéressante de type Reharsal et Regularisation.

On considère un dataset d'Incremental learning constitué de N batchs $\{1, ..., T\}$, tel que chaque batch soit composé d'une unique classe. C'est un dataset de type class-incremental, les. classes sont vues par le modèle les unes après les autres. Le modèle iCaRL va être entraîné de manière incrementale, d'abord sur le batch 1, puis le batch 2, jusqu'au batch T. Son entraînement consistera à classifier ces entrées parmi T classes, numérotées de 1 à T. La configuration class-incremental des datasets sur lesquels est entraîné iCaRL permet de faire l'analogie entre classe et batch (1 classe par batch).

Le modèle incremental Classifier and Representation Learning (iCaRL) est un modèle qui utilise à la fois une stratégie Regularization et une stratégie Rehearsal. En particulier, ce modèle utilise de la distillation et du Representation Learning. C'est une méthode qui se distingue des autres car elle a vocation à être efficace sur des problèmes de Deep Learning là où les méthodes proposées jusqu'ici ont des performances qui s'effondrent rapidement dans ce cadre. Elle s'applique sur des réseaux de neurones convolutifs (CNN).

Une des particularités d'un modèle iCaRL est qu'il apprend à la fois le classifieur (ce qui permet la classification) et la représentation des features [1].

Mémoires dans iCaRL

On rappelle [2] que pour une instance donnée x en entrée d'un réseau convolutif, le *feature vector* de l'instance x est la sortie des couches de convolution du réseau pour l'entrée x. Sur la figure 16, les features vectors sont donc les sorties des couches dans la zone bleue.

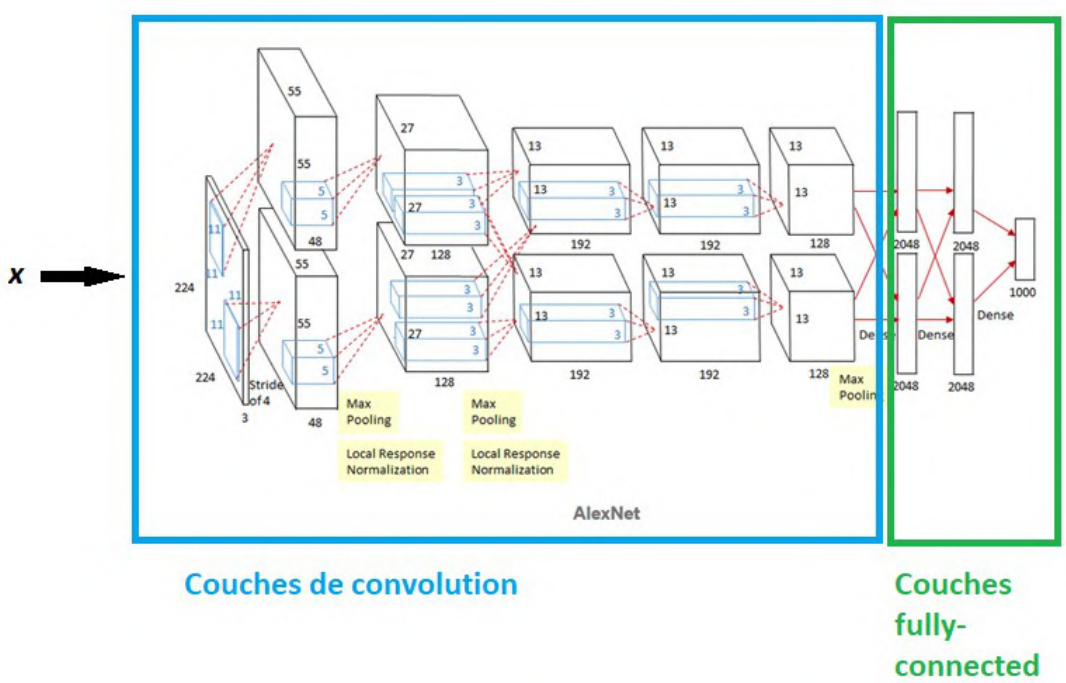


FIGURE 16 – Structure d'un réseau convolutif (AlexNet)

La fonction qui, à une instance x, associe son feature vector sera notée ϕ.
De même que pour GEM, iCaRL constitue des mémoires lors de l'apprentissage. Cependant, pour iCaRL, un batch contient une unique classe, et chaque classe n'est donc présente que dans un batch. Si le modèle est entraîné sur T classes, il va alors créer $\{M_1, ..., M_T\}$ mémoires au cours de l'apprentissage. Pour une classe $s \in \{1, ..., T\}$, sa mémoire M_s est un sous-ensemble des features vectors des instances appartenant à la classe s. Lors de l'entraînement sur un batch

1. *cf* notes sur les réseaux convolutifs en annexe
2. *cf* annexe

$s \in \{1, ..., T\}$, les features vectors des instances qui sont conservés dans les mémoires ont été choisis parmi les instances sur lesquelles le modèle aura été entraîné lors de l'entraînements sur les batchs précédents $i < s$.

Dans iCaRL, le nombre total d'instances conservées en mémoire, appelé K est fixé à l'avance. C'est donc un hyperparamètre du modèle. A un instant donné de l'entraînement, si le modèle a rencontré t classes lors de son entraînement, le nombre de features vectors d'instances contenu dans chaque mémoire est de $\lfloor \frac{K}{t} \rfloor$. La taille d'une mémoire décroît donc au cours de l'entraînement sur différents batchs, puisqu'il décroît lorsque le nombre de classes rencontrées, t, augmente.

Classification Nearest-Mean-of-Examplars

La stratégie de classification qu'utilise iCaRL est une stratégie dite Nearest-Mean-of-Exemplars. Pour chaque classe rencontrée lors de l'entraînement, $s \in \{1, ..., T\}$, de mémoire M_s, on définit le feature vector moyen de la mémoire M_s :

$$\widetilde{\mu_s} = \frac{1}{card(M_s)} \sum_{x \in s / \phi(x) \in M_s} \phi(x)$$

Pour classifier une instance x dans une des classes $\{1, ..., T\}$, le modèle iCaRL calcule le feature vector de x, $\phi(x)$, et assigne à x la classe s pour laquelle le feature vector moyen associé à sa mémoire, μ_i, est le plus proche de $\phi(x)$:

$$s = \underset{i \in \{1,...,T\}}{\operatorname{argmin}} \|\phi(x) - \widetilde{\mu_i}\|$$

Utiliser cette règle de classification permet de ne pas avoir à utiliser la moyenne des feature vectors de chaque classe, qui nécessiterait de stocker toutes les instances. En effet, en notant S le batch associé à la classe s, le feature vector moyen d'une classe $s \in \{1, ..., T\}$ est défini par :

$$\mu_s = \frac{1}{card(S)} \sum_{x \in S} \phi(x)$$

Le feature vector moyen d'une classe s dépend donc de ϕ, la fonction qui modélise l'action des couches de convolution sur une entrée x. Or les poids des couches de convolution sont modifiés au cours de l'apprentissage. Il faudrait lors de l'entraînement sur une classe $k \in \{1, ..., T\}$ recalculer $\mu_s, \forall s < k$ avec la fonction ϕ mise à jour, et donc disposer de toutes les instances des batchs $s < k$.

A la place, le feature vector moyen de la mémoire associée à la classe est utilisé, un vecteur qu'il est possible calculer à partir des features vectors conservés en mémoire uniquement. Il est donc important que la mémoire d'une classe représente bien la classe dans le sens où le feature vector moyen de la mémoire doit approximer le mieux possible le feature vector moyen de la classe.

Representation Learning

Le modèle iCaRL tire également parti de la feature extraction. [1]

Lors de l'entraînement sur un nouveau batch $k \in \{1, ..., T\}$, une routine est systématiquement mise en place :

1. Pour chacune des mémoires associées aux classes déjà observées dans les batchs $i < k$, on calcule les prédictions du modèle avec, en entrée, les features vectors stockés dans les mémoires (directement insérés dans les couches fully-connected)

2. On calcule les sorties du réseau avec en entrée les instances du batch k

Ces résultats seront utilisés pour calculer la fonction de coût à minimiser par l'algorithme de descente du gradient. Cette fonction de coût intègre en effet un terme de régularisation [2], qui s'ajoute au terme classiquement utilisé [3]. La fonction de coût est alors de la forme :

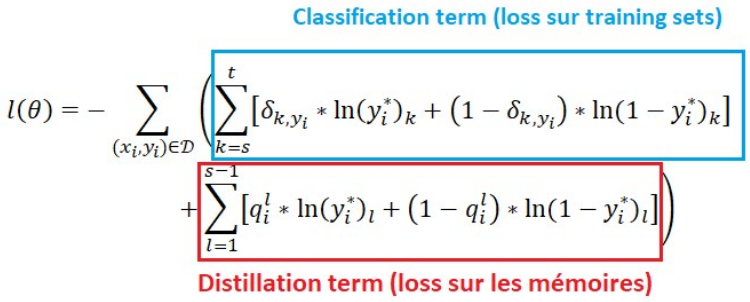


FIGURE 17 – Fonction de coût utilisée par le modèle iCaRL

1. *cf* lexique
2. terme de *distillation loss*
3. terme de *classification loss*

La fonction de coût ainsi définie permet d'inciter le modèle à reproduire les résultats obtenus lors de l'entraînement des batchs précédents $i < k$, et à classifier correctement les instances du nouveau batch k. Ensuite, l'algorithme de descente du gradient est appliqué sur cette fonction de coût pour entraîner le modèle sur le batch k.

Gestion des mémoires

Lors de l'entraînement sur un nouveau batch $s \in \{1, ..., T\}$, la routine suivante est mise en place pour gérer les mémoires :

1. On crée une mémoire M_s pour la classe présente dans le batch s.

2. On réduit les mémoires $M_1, ..., M_{s-1}$ déjà crées, celles des classes rencontrées dans les batchs précédents.

Ces deux actions ont un objectif : faire en sorte que la mémoire d'une classe représente au mieux cette classe, dans le sens où le feature vector moyen de la mémoire doit être le plus proche possible du feature vector moyen de la classe.

Pour atteindre cet objectif, l'ajout de feature vectors à une mémoire se fait de manière ordonnée : on ajoute en premier dans la mémoire M_s les features vectors des instances de la classe s dont la moyenne $\widetilde{\mu}_s$ approxime le mieux la moyenne des features vectors de la classe, μ_s. Pour chaque instance x de classe s sur laquelle le modèle est entraîné, on calcule :

$$\epsilon_x = \left\| \mu_s - \frac{1}{card(M_s) + 1} \left(\phi(x) + \widetilde{\mu}_s \right) \right\|$$

Alors, la mémoire M_s, de taille m, est constituée des m features vectors des instances de la classe s pour lesquels la quantité ϵ_x est la plus petite. Les features vectors sont ordonnés dans la mémoire dans l'ordre croissant de ϵ_x. Puisqu'une classe n'est présente que dans un batch, la mémoire ordonnée est créée lorsque le modèle est entraîné sur la classe/batch. Lors de l'entraînement sur les batchs suivants, le modèle n'aura qu'à diminuer la taille des mémoires déjà créées en supprimant les derniers features vectors qui y ont été ajoutés. L'ordre des mémoires permet donc une gestion facile de la suppression sans avoir à refaire de calculs.

1.4 Choix des méthodes à développer et améliorer

La méthodologie suivie pour l'état de l'art a été d'explorer les 3 grandes stratégies d'Incremental learning, Architectural, Regularization et Rehearsal. Pour cela, les années 2016 et 2017 ont marqué l'apparition des papiers qui présentaient les premières méthodes classiques de ces 3 grandes familles.

Le choix a été fait de se concentrer sur les méthodes classiques datant de 2017 dans l'état de l'art. Beaucoup des modèles qui ont des performances élevées au moment de l'état de l'art du début de ce projet [1] sont en effet des variantes évoluées des modèles de 2017.

Plusieurs facteurs sont entrés en compte dans le choix du modèle à approfondir. Le premier a été l'efficacité à empêcher le Catastrophic Forgetting. Pour des raisons pratiques, les modèles pour lesquels une implémentation était disponible ont été privilégiés. De plus, les différentes approches ont également été comparées. Le choix s'est porté sur les modèles dont l'approche se calquait sur l'utilisation réelle des algorithmes de Machine Learning, puisque l'Incremental learning vise à approcher ces conditions. Enfin, les résultats des modèles ont été pris en compte, en privilégiant les modèles qui présentent les meilleurs résultats.

La méthode **iCaRL** aurait été intéressante à développer. Ses performances sont bonnes parmi les modèles de 2017, et sa configuration d'apprentissage vise à être une configuration d'Incremental learning réaliste. Le papier souligne en effet que ce n'est pas le cas des méthodes présentées jusqu'à sa création. Néanmoins, iCaRL ne s'applique qu'aux réseaux de neurones convolutifs, ce qui réduit la classe d'algorithmes considérés.

Le modèle **Gradient Episodic Memory** (GEM) a alors été identifié comme le modèle à approfondir. Il présente des similarités avec iCaRL, les deux modèles adoptent notamment tous deux une stratégie Rehearsal[2]. Les performances présentées dans le papier de GEM en faisaient le modèle le plus efficace pour la précision en 2017. GEM est ultérieur à iCaRL et est notamment comparée avec iCaRL dans son papier, obtenant de meileurs résultats.

De plus, GEM est le modèle qui résolvait le mieux le Catastrophic Forgetting en garantissant un Backward Transfer positif. Son implémentation et ses résultats seront présentés dans la partie suivante, en reprenant l'implémentation de GEM avec celle de iCaRL et et EWC. Leurs résultats seront alors comparés dans la configuration d'apprentissage définie pour GEM.

1. Octobre 2020
2. iCaRL adopte également une stratégie de Régularisation

2 Implémentation de la méthode GEM

2.1 Détails de l'implémentation

Les expérimentations sont menées sur 3 datasets : CIFAR 100, MNIST permutations et MNIST rotations.

2.1.1 Architecture des modèles utilisés

Un réseau de neurones à couches fully-connected et à 2 couches cachées est utilisé pour les expérimentations sur les datasets MNIST permutations et MNIST rotations. Les couches cachées possèdent 100 neurones, et leur fonction d'activation est la fonction Relu.

Un réseau de neurones qui est une version plus petite de ResNet18 [8] (5 couches de convolution, une couche fully-connected, une couche softmax) est utilisé pour les expérimentations Sur CIFAR100. Cette version plus petite contient 3 fois moins de features maps que ResNet18 sur toutes les couches de convolution.

Plusieurs modèles sont comparés :

— *Single*, un seul classifieur entraîné sur tous les batchs.

— *Independant*, un classifieur par batch. Même architecture que *Single* mais avec 20 fois moins de neurones sur les couches cachées. Chaque classifieur est initialisé aléatoirement ou en clonant le dernier prédicteur entraîné (décidé par grid-search).

— *Multimodal*, même architecture que *Single*, mais chaque batch a une couche d'entrée dédiée. Evalué sur MNIST uniquement.

— *EWC*, Elastic Weight Consolidation (Regularization).

— *iCaRL*, méthode combinant Rehearsal et Regularization. Uniquement évaluée sur CIFAR100.

— *GEM*, Gradient Episodic Memory (Rehearsal).

Les méthodes EWC, iCaRL et GEM ont la même architecture que Single, avec les mémoires (dans le cas de GEM) qui s'y ajoutent. En ce qui concerne les hyperparamètres d'entraînement, les Datasets sont constitués de 20 batchs. Chacun des batchs est assimilé à une tâche que le réseau apprend à faire. Chaque modèle apprend successivement chaque batch et chaque exemple une seule fois. Les modèles sont entraînés avec l'algorithme SGD de Descente du Gradient Stochastique, pour des mini-batches de 10 exemples. Les hyper-paramètres ont été optimisés par grid-search, les résultats reportés sont donc les meilleurs.

2.2 Résultats

Dans cette section sont présentés les graphes tracés en sortie de l'exécution de chacun des modèles sur chacun des datasets. Les résultats seront comparés entre le papier GEM [6], et l'implémentation reproduite. En annexe sont reportés dans des tableaux récapitulatifs les résultats pour l'implémentation reproduite des métriques Précision, Backward Transfer et Forward Transfer.

2.2.1 CIFAR 100

Ci-dessous la légende utilisée pour les expérimentations sur le dataset CIFAR100 :

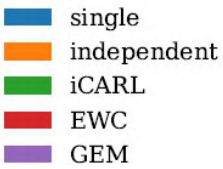


FIGURE 18 – Légende des couleurs utilisées CIFAR100

La figure ci-dessous représente l'évolution de la précision des modèles Single, Independant, iCaRL, EWC et GEM sur le premier batch en fonction du nombre de batchs appris :

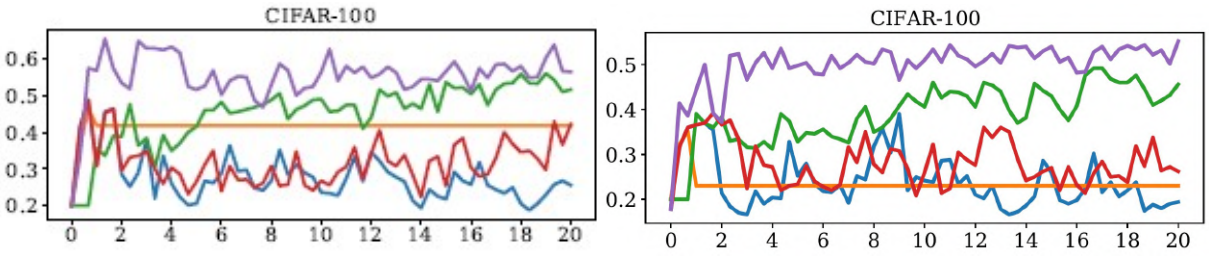


FIGURE 19 – Précision sur le premier batch en fonction du nombre de batchs appris sur CIFAR100 (gauche : papier, droite : implémentation reproduite)

L'échelle en ordonnées n'est pas exactement la même entre les résultats du papier et ceux de l'implémentation reproduite, il faut donc regarder les valeurs que prennent les courbes pour les comparer (entre implémentation du papier et implémentation reproduite).

L'allure de l'évolution de la précision sur le premier batch pour GEM (violet) concorde pour les résultats du papier et les résultats de la reproduction de cette implémentation. C'est également le cas pour les courbes des modèles Single, iCaRL, et EWC, bien que des différences d'allures locales existent.

Neanmoins, il est possible d'observer localement des différences de valeurs entre ces deux courbes. Globalement, les valeurs pour l'implémentation reproduite semblent toujours être inférieures à celles du papier. La différence d'allure des courbes correspondant, ainsi que de leurs positions relatives, s'assimile à une dilatation des courbes de Single, GEM, iCaRL et EWC de l'ordre de 8 à 15%.

En revanche, le modèle Independant a, à partir du 2^{me} batch appris, des valeurs de précision sur le premier batch qui ne concordent pas avec le papier. L'allure globale concorde : un pic de précision après l'apprentissage du deuxième batch puis immédiatement après une précision qui diminiue puis reste constante. Cependant, la valeur du plateau constant est très inférieure pour la reproduction de l'implémentation (environ 25 % contre environ 40 % pour le papier).

Plusieurs explications aux différences observées sont possibles :

— une version du code différente entre le code disponible sur le GitHub d'où est tirée la reproduction de l'implémentation, et la version du code utilisée pour obtenir les résultats du papier. C'est une hypothèse plausible au vu notamment de la légère différence d'échelle d'ordonnées entre les 2 courbes.

— un paramètre initialisé aléatoirement dans les modèles qui varierait entre chaque exécution des modèles. Cela pourrait expliquer les différences locales d'allures, mais n'explique pas que tous les modèles reproduits suivent la même allure que dans le papier en ayant des valeurs dilatées quasiment uniformément. De plus, les paramètres aléatoires de GEM sont initialisés par seeding, qui permet de générer une suite aléatoire à chaque fois identique en lui fournissant un même premier terme. Si une différence est dûe à un paramètre aléatoire qui varie entre chaque exécution, il s'agit d'un paramètre sous-jacent aux modèles (dans les classes implémentées...).

— une évolution des classes du module PyTorch de Python utilisé pour ces implémentations qui conduirait à des performances légèrement différentes.

— une modification du dataset CIFAR100 (peu probable).

Malgré ces différences entre le papier et l'implémentation reproduite, dans les deux cas l'efficacité de GEM est illustrée. GEM est le modèle qui a la précision la plus importante sur le premier batch au cours de son apprentissage. Seul iCaRL (Rehearsal+Regularization) voit ses performances sur le premier batch conservées lorsqu'il apprend de nouveaux batchs. Néanmoins, la précision de GEM sur ce premier batch est meilleure que celle d'iCaRL tout au long de l'apprentissage et à la fin de l'entraînement (environ 5% d'écart). Les autres modèles connaissent une chute des performances sur le premier batch après l'abscisse 1 (moment où est entraîné le premier batch).

Sur les diagrammes suivants est présentée l'évaluation des modèles Single, Independant, iCaRL, EWC et GEM sur les 3 métriques définies (Précision, Backward Transfer, Forward Transfer), sur le dataset CIFAR100 :

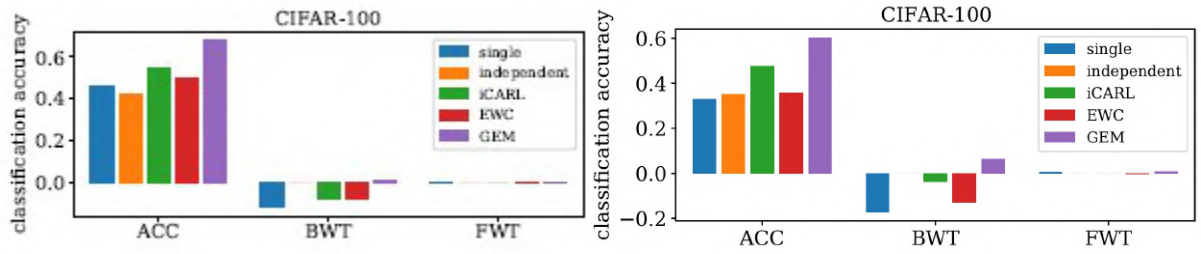


FIGURE 20 – Précision, Backward Transfer et Forward Transfer sur CIFAR100 (gauche : papier, droite : implémentation reproduite)

La précision moyenne sur l'ensemble des batchs (ACC) est cohérente avec ce qui a été observé pour le premier batch. Les allures et positions relatives des précisions des modèles concordent entre les résultats du papier et les résultats de l'implémentation reproduite. La différence observée est ce qui semble être une dilatation des précisions de l'ordre de 8 à 15% en diminution.

Seule la position de la courbe du modèle Independant par rapport à celle des autres modèles diffère dans la reproduction de l'implémentation, comme c'était le cas pour la précision sur la première tâche.

En ce qui concerne le Backward Transfer (BWT), les résultats semblent dilatés de la même manière pour l'implémentation reproduite, tout en suivant une position relative généralement cohérente par rapport aux résultats du papier.

Le Backward Transfer de GEM est bien le seul qui soit positif dans les deux implémentations, en accord avec la théorie. GEM est donc bien efficace pour minimiser le Catastrophic Forgetting qui se traduit par des valeurs très négatives de Backward Transfer. Dans l'implémentation reproduite, seul le modèle iCaRL a un Backward Transfer qui se positionne différemment par rapport aux autres modèles, en comparaison avec les résultats du papier.

Enfin, le Forward Transfer (FWT) est semblable entre l'implémentation reproduite et les résultats du papiers, bien que sa faible valeur par rapport à l'échelle rende difficile la comparaison. Pour tous les modèles, dans les deux cas, le Forward Transfer est très légèrement positif. Le *zero-shot learning* (évaluation d'un modèle sur une tâche qu'il n'a pas été entraîné à faire) est donc en moyenne très légèrement meilleur pour les modèles entraînés sur plusieurs tâches que pour un modèle initialisé aléatoirement et pas entraîné.

Il apparaît alors que, lorsque les modèles apprennent à faire certaines tâches, ils s'améliorent à la fois à la réalisation de ces tâches, mais aussi à la réalisation de tâches qu'ils n'ont pas encore appris à faire. Intuitivement, une condition pour

que ce soit possible est que les tâches qu'ils n'ont pas apprises sur lesquelles ont les évalue doivent être proches de celles qu'ils ont apprises. Ici, l'apprentissage d'une tâche est assimilée à l'apprentissage d'un batch, chaque batch de CIFAR100 contenant des images avec un nombre de classes données. Les classes désignent des types d'objets et chaque classe n'est présente que dans un batch.

2.2.2 MNIST permutations

Ci-dessous la légende utilisée pour les expérimentations sur le dataset MNIST permutations :

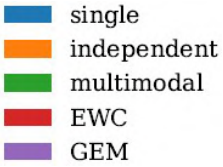


FIGURE 21 – Légende des couleurs utilisées MNIST permutations

Sur la figure ci-dessous sont présentées l'évolution de la précision des modèles Single, Independant, Multimodal, EWC et GEM sur le premier batch en fonction du nombre de batchs appris, sur MNIST permutations :

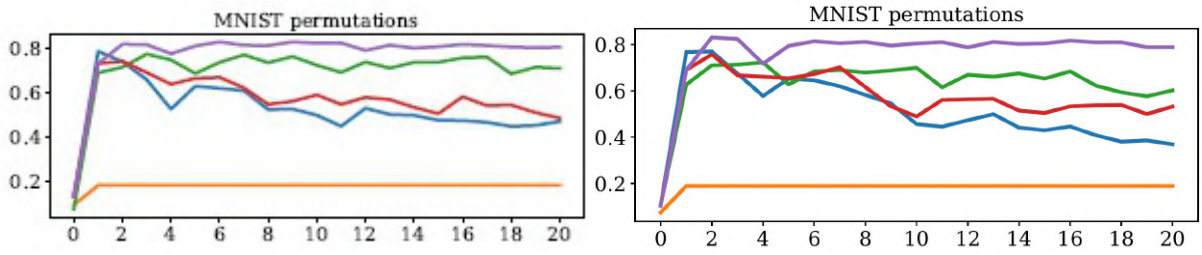


FIGURE 22 – Précision sur le premier batch en fonction du nombre de batchs appris sur MNIST permutations (gauche : papier, droite : implémentation reproduite)

Ici, pour tous les modèles, les allures et les valeurs de précision sur la première tâche correspondent entre le papier GEM et l'implémentation reproduite. Les positions relatives des courbes correspondent également. Localement apparaissent quelques différences minimes, qui pourraient s'expliquer par les facteurs mentionnés pour CIFAR (aléatoire, évolution du code, évolution du langage...).

La figure ci-dessous représente l'évaluation des modèles Single, Independant, Multimodal, EWC et GEM sur les 3 métriques définies (Précision, Backward Transfer, Forward Transfer), sur le dataset MNIST permutations :

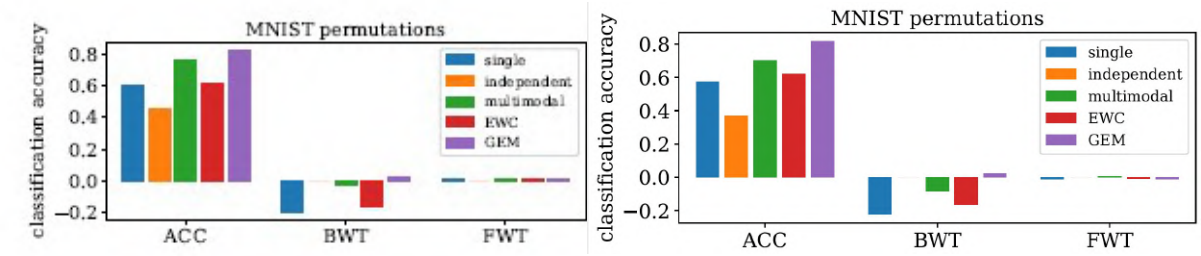


FIGURE 23 – Précision, Backward Transfer et Forward Transfer sur MNIST permutations (gauche : papier, droite : implémentation reproduite)

De la même manière que pour la précision sur le premier batch, le Backward Transfer, le Forward Transfer et la précision correspondent entre les résultats du papier et ceux de l'implémentation reproduite. Seul le modèle Independant, possède une valeur de précision sensiblement plus faible lors de l'implémentation reproduite. Il se distinguait des autres pour CIFAR100 par la grande différence entre les résultats du papier et ceux de l'implémentation reproduite.

La précision du modèle GEM sur le dataset MNIST permutations est donc meilleure que celle de tous autres les modèles avec lesquels il est comparé. GEM est également le seul modèle à présenter sur MNIST permutations (et CIFAR100) un Backward Transfer positif, qui témoigne du fait que l'apprentissage de nouveaux batchs ne perturbe pas les performances sur les batchs déjà appris, il le renforce même en moyenne légèrement. Il n'y a donc pas de Catastrophic Forgetting pour GEM sur MNIST permutations, au contraire de EWC et Single notamment.

2.2.3 MNIST rotations

Ci-dessous le code couleurs utilisé pour les expérimentations sur le dataset MNIST rotations :

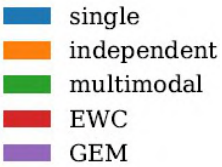


FIGURE 24 – Légende des couleurs utilisées MNIST rotations

La figure ci-dessous présente l'évolution de la précision des modèles Single, Independant, Multimodal, EWC et GEM sur le premier batch en fonction du nombre de batchs appris :

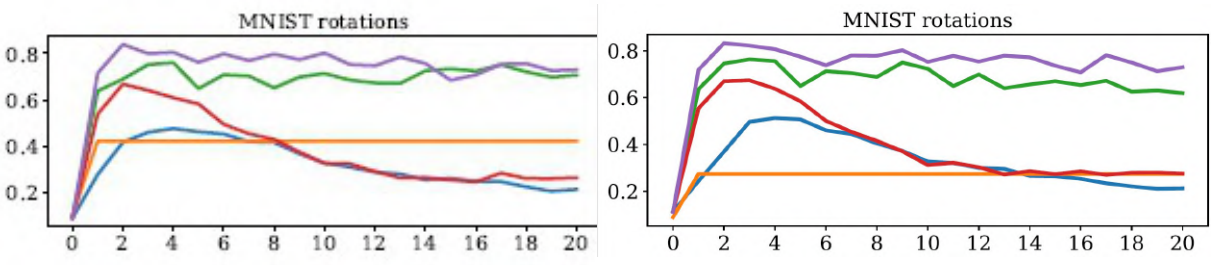


FIGURE 25 – Précision sur le premier batch en fonction du nombre de batchs appris sur MNIST rotations (gauche : papier, droite : implémentation reproduite)

De même que pour MNIST permutations, les résultats concordent entre l'implémentation reproduite et le papier, à l'exception du modèle Independant.

Sur les diagrammes suivants sont représentées l'évaluation des modèles Single, Independant, Multimodal, EWC et GEM sur les 3 métriques définies (Précision, Backward Transfer, Forward Transfer), sur le dataset MNIST rotations :

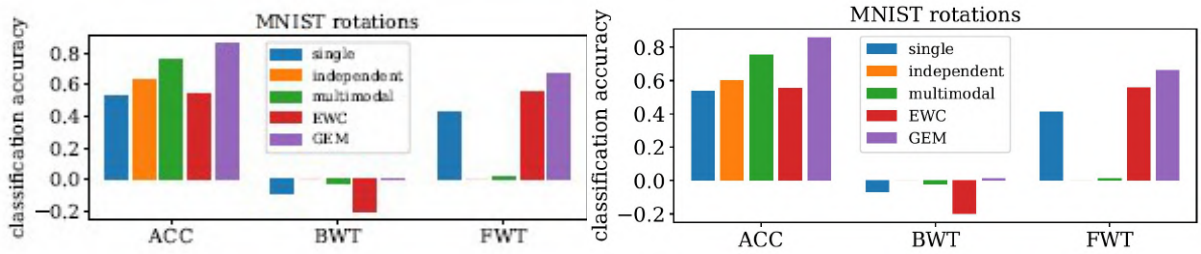


FIGURE 26 – Précision, Backward Transfer et Forward Transfer sur MNIST rotations (gauche : papier, droite : implémentation reproduite)

Comme sur MNIST permutations, les résultats concordent parfaitement entre le papier de GEM et l'implentation reproduite (avec une légère différence de Backward Transfer pour Independant).

De même que sur CIFAR100 et MNIST permutations, GEM est le modèle avec la précision la plus élevée en moyenne sur l'ensemble des 20 batchs, et est le seul à avoir un Backward Transfer positif.

Le Forward Transfer sur MNIST permutations est beaucoup plus élevé, pour tous les modèles, que sur les datasets MNIST rotations et CIFAR100. Cette métrique quantifie l'influence moyenne de l'apprentissage d'un batch sur le zero-shot learning, la réalisation de tâches non-apprises. Cela doit tenir à la configuration de MNIST permutations en comparaison des autres datasets, bien qu'il paraisse étonnant que le Forward Transfer soit élevé pour tous les modèles sur MNIST permutations et pas sur MNIST rotations au vu des similarités dans la construction des deux datasets.

2.3 Conclusions des implémentations reproduites

Les expériences reproduites depuis le GitHub du papier de GEM sont dans l'ensemble concordantes avec les résultats présentés dans le papier. Sur les datasets MNIST rotations et MNIST permutations, l'évolution de la précision des modèles sur le premier batch suit la même allure dans les deux expérimentations, les positions relatives des courbes des modèles sont les mêmes, et les valeurs également (sauf pour le modèle Independant). L'évaluation des modèles sur MNIST rotations et MNIST permutations sur les métriques propres à l'Incremental learning (Backward Transfer, Précision, Forward Transfer) est également quasiment identique entre le papier et la reproduction.

Pour l'évaluation des modèles sur le dataset CIFAR100, toutes les mesures de précisision (sur le premier batch et en moyenne) semblent être dilatées en diminution de 8 à 15% environ. Les évolutions et positions relatives des courbes des modèles correspondent entre le papier et l'implémentation reproduite, sauf pour le modèle Independant dont l'allure correspond mais pas les valeurs.

De manière générale, les résultats des implémentations reproduites montrent l'efficacité de GEM dans la configuration d'Incremental learning précisée, sur les datasets MNIST rotations, MNIST permutations et CIFAR100. Sa précision est meilleure que celle des autres modèles qui existaient en 2017 lors de sa création, et son Backward Transfer positif sur chacun des datasets témoigne de son efficacité

à éviter le Catastrophic Forgetting. Ce Backward Transfer positif pour GEM montre même que les performances sur les batchs déjà appris s'améliorent en moyenne (très légèrement) grâce à l'apprentissage des nouveaux batchs.

3 Améliorations des méthodes et expérimentations

La partie qui suit présente des pistes d'améliorations pouvant être apportées au modèle Gradient Episodic Memory présenté dans la partie précédente. Ces pistes se diviseront en deux catégories :

— des idées d'améliorations qui proviennent de la veille bilbiographique réalisée lors du projet. Ces idées visent surtout à agir sur la contrainte imposée au gradient par GEM à chaque itération de l'algorithme de descente du gradient ;

— des idées d'améliorations personnelles, qui visent à améliorer la sélection des exemples conservés dans les mémoires.

3.1 Averaged Gradient Episodic Memory (A-GEM)

Un article proposé par un des auteurs de GEM en propose une variante de GEM, Averaged-GEM (A-GEM) : *Efficient Lifelong Learning with A-GEM*, publié en 2019 par Arslan Chaudhry &al. [9]. Cette méthode propose un compromis entre coût de calcul et efficacité, en modifiant la contrainte appliquée au gradient sur GEM. La contrainte imposée par A-GEM consiste à diminuer, à chaque étape de l'algorithme de descente du gradient, le coût moyen sur les mémoires définies dans GEM. GEM imposait comme contrainte de diminuer le coût sur chacune des mémoires. Lors de l'entraînement sur un batch $k \in \{1, ..., N\}$, la contrainte de A-GEM est :

1. minimiser la fonction de coût calculée sur le batch k
2. diminuer la fonction de coût calculée sur l'ensemble des mémoires des batchs $i < k$ (sur lesquels le modèle a déjà été entraîné) par rapport à l'itération précédente de l'algorithme de descente du gradient. Ici se fait la différence avec GEM, qui impose de diminuer la fonction de coût calculée sur **chacune** des mémoires des batchs $i < k$ (sur lesquels le modèle a déjà été entraîné), par rapport à l'itération précédente de l'algorithme de descente du gradient

La condition de A-GEM permet de s'assurer que l'entraînement sur un batch ne perturbe pas les résultats de l'entraînement effectué sur les batchs précédents, tout en étant moins restrictive. Il résulte d'après le papier que A-GEM est environ 100 fois plus rapide à l'exécution que GEM, avec des besoins 10 fois moins élevés en terme de mémoire.

De plus, GEM garantit une meilleure protection contre l'oubli sur chacun des batchs sur lequel le modèle a déjà été entraîné, tandis que la contrainte introduite par A-GEM induit de meilleures garanties sur la précision moyenne du modèle.

La condition en termes de gradient [1] de la contrainte lors de l'entraînement sur un batch $k \in \{1, ..., N\}$ introduite par GEM est :

$$\begin{cases} \widetilde{g} = \min_{\widetilde{g}} \|g - \widetilde{g}\| \\ \text{où} \\ \langle \widetilde{g}, g_i \rangle \geq 0 \quad \forall i < k \end{cases}$$

La condition en termes de gradient[1] de la contrainte lors de l'entraînement sur un batch $k \in \{1, ..., N\}$ introduite par A-GEM est :

$$\begin{cases} \widetilde{g} = \min_{\widetilde{g}} \|g - \widetilde{g}\| \\ \text{où} \\ \langle \widetilde{g}, g_{ref} \rangle \geq 0 \end{cases}$$

où g_{ref} est un gradient calculé sur les instances conservées en mémoire, en utilisant un sous-ensemble tiré aléatoirement depuis l'union des mémoires $\bigcup_{i<k} M_i$.

La figure 27 présente les résultats obtenus pour l'implémentation de A-GEM sur les datasets MNIST permutations et MNIST rotations.

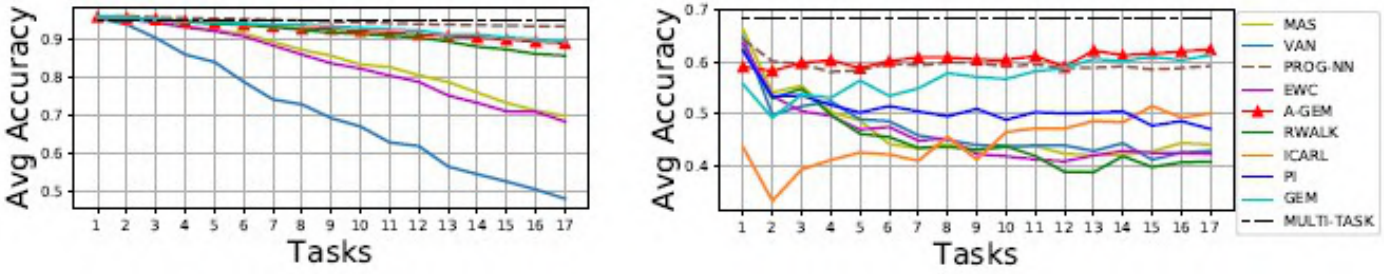


FIGURE 27 – Evolution de la précision moyenne au cours de l'apprentissage sur MNIST permutations (à gauche) et CIFAR-100 (à droite)

La précision moyenne sur ces deux datasets est donc similaire entre A-GEM et GEM, A-GEM obtenant même des résultats légèrement supérieurs au cours de l'apprentissage sur CIFAR-100. A-GEM performe parmi les meilleures méthodes

1. de la fonction de coût du modèle

d'Incremental learning avec lesquelles elle est comparée (la borne supérieure étant la méthode mutli-task qui n'est pas à proprement parler une méthode d'Incremental learning).

3.2 ϵ-soft-Gradient Episodic Memory

Une veille bibliographique a été réalisée avant de réfléchir à de possibles améliorations à apporter à GEM. Un papier, ultérieur à l'état de l'art effectué dans ce rapport, propose une variante de A-GEM avec des performances au niveau de l'état de l'art au moment de sa parution. Il s'agit de l'article *Gradient Episodic Memory with a Soft Constraint for Continual Learning* publié par Guanna Hu, Zhang Wu & al.[10].

La piste suivie par ce papier est également d'agir sur la contrainte imposée par GEM et A-GEM en introduisant un nouvel hyperparamètre au modèle, $\epsilon \in [0, 1]$. La reformulation en terme de gradient de la fonction de coût de ϵ-soft-GEM est (lors de l'apprentissage d'un batch k) :

$$\begin{cases} \widetilde{g} = \min_{\widetilde{g}} \|g - \widetilde{g}\| \\ \text{où} \\ \langle \widetilde{g}, g_{ref} \rangle \geq \epsilon \end{cases}$$

où g_{ref} est un gradient calculé sur les instances conservées en mémoire, en utilisant un sous-ensemble tiré aléatoirement depuis l'union des mémoires $\bigcup_{i<k} M_i$.

Le modèle ϵ-soft-GEM est alors comparé à plusieurs modèles étudiés tels que les A-GEM, Elastic-Weight Consolidation (EWC), et une version améliorée des Progressive Neural Networks (PNN),

La figure 28 présente la précision moyenne du modèle ϵ-soft-GEM (vert pâle) sur les datasets MNIST-permutations, CIFAR-100, split-CUB.

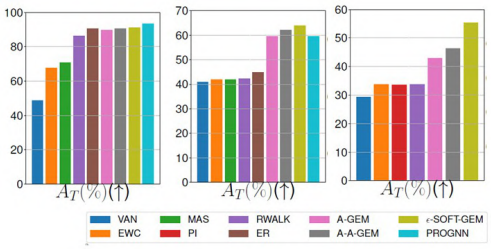


FIGURE 28 – Expérimentations sur ϵ-soft-GEM pour MNIST-permutations, CIFAR-100 (centre), split-CUB (droite)

Les résultats du modèle ϵ-soft-GEM sont bien les meilleurs en terme de précision moyenne, sur CIFAR-100 et split-CUB, dépassant les 85 % de précision moyenne sur MNIST dans la configuration des expérimentations réalisées par le papier.

Sur la figure 29, le Backward Transfer obtenu lors des expérimentations est présenté :

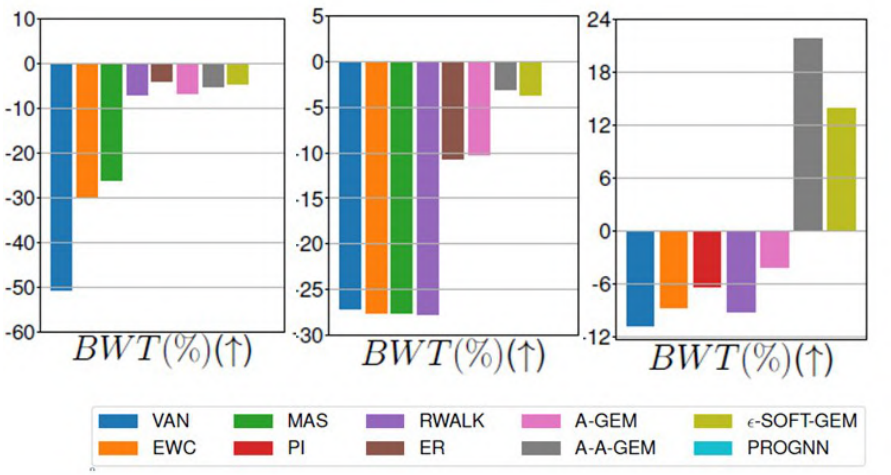


FIGURE 29 – Backward Transfer sur MNIST-permutations, CIFAR-100 (centre), split-CUB (droite)

Il est possible de noter que parmi tous les modèles comparés, ϵ-soft-GEM est celui qui obtient le meilleur Backward Transfer sur MNIST permutations, le deuxième meilleur Backward Transfer sur CIFAR-100 et split-CUB. Sur ces 3 datasets, le Backward Transfer de ϵ-soft-GEM n'est jamais fortement négatif, ce modèle ne subit donc pas de catastrophic forgetting au cours de l'apprentissage. Le Backward Transfer du modèle n'est pas toujours positif, au contraire de GEM.

C'est également le cas de A-GEM, ce qui montre que la contrainte imposée sur le gradient par A-GEM [1] est effectivement moins restrictive que celle imposée par GEM. Le Backward Transfer est donc privilégié par GEM, tandis que la précision moyenne du modèle est bien favorisée par A-GEM et ϵ-soft-GEM.

3.3 Amélioration de la sélection en mémoire

3.3.1 Intuition de l'amélioration imaginée

Dans le modèle de GEM, les instances conservées en mémoire sont, pour chaque batch, les dernières instances du batch fournies au modèle. Les mémoires sont ensuites utilisées dans l'algorithme de descente du gradient lorque le gradient de la fonction de coût du modèle ne respecte pas la contrainte imposée. Cette contrainte est rappelée :

1. minimiser la fonction de coût calculée sur le batch k,
2. diminuer la fonction de coût calculée chacun des batchs sur lesquels le modèle a déjà été entraîné par rapport à l'itération précédente de l'algorithme de descente du gradient.

Le postulat qui est fait est qu'améliorer la sélectivité des instances conservées dans les mémoires $M_k, \forall k \in \{1, ..., N\}$ permet d'imposer une contrainte permettant d'améliorer le Backward Transfer. Pour la respecter, il faudrait qu'à chaque itération de l'algorithme de descente du gradient, le coût diminue sur un échantillon vraiment représentatif de chacun des batchs sur lesquels le modèle a déjà été entraîné. Cela permettrait alors de contraindre le modèle à s'améliorer à la reconnaissance d'instances qui représentent le mieux chaque batch.

3.3.2 Idée liée à iCaRL

Il faudrait alors définir une mesure de la qualité de la représentation d'un batch par sa mémoire, afin de choisir de conserver en mémoire les éléments du batch qui minimisent cette mesure. Cette pratique s'approche de la solution proposée par iCaRL pour gérer les mémoires que le modèle crée lorsqu'il est entraîné sur un nouveau batch. Le modèle iCaRL propose de conserver en mémoire les instances du batch dont les features vectors permettent le mieux d'approcher le feature vector[2] moyen du batch.

1. qui sert de base à la contrainte imposée par ϵ-soft-GEM
2. Voir annexe sur les CNN.

Il pourrait être intéressant de procéder de la même manière pour les instances :

Pour chaque batch rencontré lors de l'entraînement, $k \in \{1, ..., N\}$, de mémoire M_k, on définit l'instance moyenne de la mémoire M_k :

$$\widetilde{x_k} = \frac{1}{card(M_k)} \sum_{x \in M_k} x$$

On définit également l'instance moyenne du batch k :

$$x_k = \frac{1}{card(k)} \sum_{x \in k} x$$

Il faut alors ajouter les instances du batch k en faisant en sorte que la mémoire M_k représente au mieux la classe, dans le sens où l'instance moyenne de la mémoire doit être la plus proche possible de l'instance moyenne de la classe.

Pour atteindre cet objectif, l'ajout d'instances à une mémoire M_k se fait de manière ordonnée : on ajoute en premier dans la mémoire les instances du batch k dont la moyenne $\widetilde{x_k}$ approxime le mieux la moyenne des features vectors de la classe, x_k. Pour chaque instance x du batch k sur laquelle le modèle est entraîné, on calcule :

$$\epsilon_x = \left\| x_k - \frac{1}{card(M_k) + 1} (x + \widetilde{x_k}) \right\|$$

Alors, la mémoire M_k, de taille m, est constituée des m instances du batch k pour lesquels la quantité ϵ_x est la plus petite. Les instances sont ordonnées dans la mémoire dans l'ordre croissant de ϵ_x. La mémoire M_k sera alors constituée du sous-ensemble du batch k qui minimise la distance de l'instance moyenne du batch k à l'instance moyenne de sa mémoire M_k. Selon le critère ϵ défini, on aura alors bien assuré la meilleure représentation du batch k possible.

Néanmoins, lors de l'implémentation pourrait apparaître un overfitting des instances conservées dans les mémoires.

La sélectivité des mémoires est notamment abordée pour GEM, dans le cas des petites mémoires, dans le papier *On Tiny Episodic Memories in Continual Learning* publié par Arslan Chaudhry & al. en 2019 [11]. Leur méthode de sélection des instances conservées en mémoire était alors une méthode hybride mêlant plusieurs méthodes : méthode de reservoir sampling, puis de ring buffer.

Les résultats obtenus par cette méthode sont bons, du même ordre que A-GEM. Les auteurs concluent que ce résultat est étonnant, car ils pensaient que la

généralisation du modèle serait mauvaise, à cause d'un overfitting des instances conservées dans les episodic memories.

Cela valide la piste qui consiste à améliorer la sélectivité des mémoires pour améliorer les performances du modèle GEM.

4 Perspectives

Une piste qui pourrait permettre d'améliorer à la fois GEM, mais aussi ses variantes plus récentes, consiste à améliorer la sélection des exemples conservés dans les mémoires du modèle. Les travaux mentionnés dans ce domaine laissent penser que cette piste est prometteuse et son implémentation sera tentée en vue de la soutenance finale.

Les perspectives de développement pourraient être de continuer à essayer d'améliorer la sélectivité des mémoires après l'implémentation de l'amélioration proposée dans ce rapport. La lecture d'articles comme [11] se rapportant à la gestion de ces mémoires pourrait fournir une bonne base théorique pour cette étude.

Conclusion

Pour conclure ce rapport, rappelons les objectifs fixés à l'occasion du RVP1, en Novembre 2020 :

1. Faire un état de l'art de l'Incremental learning pour étudier les différents types de modèles ;
2. Approfondir un des modèles existants, et proposer des pistes d'améliorations ;
3. Communiquer les résultats des travaux effectués.

Le premier objectif aura été rempli lors de ce projet. L'état de l'art effectué reprend les modèles classiques introduits en 2017, qui servent de base aux modèles actuels ayant les meilleures performances. L'approfondissement du modèle GEM aura même permis de restituer des articles parus au cours du projet et aux performances parmi les meilleures actuellement.

Le second objectif aura également été tenu. Le modèle Gradient Episodic Memory a pu être implémenté avec succès au regard de la comparaison des résultats obtenus et des résultats présentés dans la papier. L'approfondissement de GEM a été proposé dans la partie III, au travers des variantes A-GEM et $\epsilon - soft - GEM$. Une piste d'amélioration a également été proposée, qui pourrait permettre d'améliorer à la fois GEM, mais aussi ses variantes plus récentes. Cette piste consiste à améliorer la sélection des exemples conservés dans les mémoires du modèle. Les travaux mentionnés dans ce domaine laissent penser que cette piste est prometteuse et son implémentation sera tentée en vue de la soutenance finale.

Enfin, la communication des travaux effectués aura été effectuée, de manière exhaustive à travers ce rapport final, mais aussi grâce aux rendez-vous de gestion et les présentations régulières effectuées avec les tuteurs. La soutenance finale du projet viendra conclure ce dernier objectif en même temps que le projet.

Une des limites de ce projet aura été le manque de temps et d'expérience. Ce manque d'expérience a notamment considérablement rallongé la phase de reproduction de l'implémentation de GEM. Ce retard par rapport au calendrier espéré a empêché la présentation de résultats d'implémentations des pistes d'amélioration évoquées dans la partie III.

Les perspectives de développement pourraient être de continuer à essayer d'améliorer la sélectivité des mémoires après l'implémentation de l'amélioration proposée dans ce rapport.

Références

[1] Vincenzo LOMONACO et Davide MALTONI. "CORe50 : a New Dataset and Benchmark for Continuous Object Recognition". en. In : *Conference on Robot Learning*. ISSN : 2640-3498. PMLR, oct. 2017, p. 17-26. URL : http://proceedings.mlr.press/v78/lomonaco17a.html.

[2] Andrei A. RUSU et al. "Progressive Neural Networks". In : *arXiv :1606.04671 [cs]* (sept. 2016). arXiv : 1606.04671. URL : http://arxiv.org/abs/1606.04671.

[3] James KIRKPATRICK et al. "Overcoming catastrophic forgetting in neural networks". en. In : *Proceedings of the National Academy of Sciences* 115.11 (mar. 2018). Publisher : National Academy of Sciences Section : Letter, E2498-E2498. ISSN : 0027-8424, 1091-6490. DOI : 10.1073/pnas.1800157115. URL : https://www.pnas.org/content/114/13/3521.

[4] Friedemann ZENKE, Ben POOLE et Surya GANGULI. "Continual Learning Through Synaptic Intelligence". en. In : *International Conference on Machine Learning*. ISSN : 2640-3498. PMLR, juil. 2017, p. 3987-3995. URL : http://proceedings.mlr.press/v70/zenke17a.html.

[5] Zhizhong LI et Derek HOIEM. "Learning without Forgetting". In : *arXiv :1606.09282 [cs, stat]* (fév. 2017). arXiv : 1606.09282. URL : http://arxiv.org/abs/1606.09282.

[6] David LOPEZ-PAZ et Marc'Aurelio RANZATO. "Gradient Episodic Memory for Continual Learning". en. In : *Advances in Neural Information Processing Systems* 30 (2017), p. 6467-6476. URL : https://proceedings.neurips.cc/paper/2017/hash/f87522788a2be2d171666752f97ddebb-Abstract.html.

[7] Sylvestre-Alvise REBUFFI et al. "iCaRL : Incremental Classifier and Representation Learning". In : *arXiv :1611.07725 [cs, stat]* (avr. 2017). arXiv : 1611.07725. URL : http://arxiv.org/abs/1611.07725.

[8] Kaiming HE et al. "Deep Residual Learning for Image Recognition". In : *CoRR* abs/1512.03385 (2015). arXiv : 1512.03385. URL : http://arxiv.org/abs/1512.03385.

[9] Arslan CHAUDHRY et al. "Efficient Lifelong Learning with A-GEM". In : *arXiv :1812.00420 [cs, stat]* (jan. 2019). arXiv : 1812.00420. URL : http://arxiv.org/abs/1812.00420.

[10] Guannan HU et al. “Gradient Episodic Memory with a Soft Constraint for Continual Learning”. In : *arXiv :2011.07801 [cs]* (nov. 2020). arXiv : 2011.07801. URL : http://arxiv.org/abs/2011.07801.

[11] Arslan CHAUDHRY et al. “On Tiny Episodic Memories in Continual Learning”. In : *arXiv :1902.10486 [cs, stat]* (juin 2019). arXiv : 1902.10486. URL : http://arxiv.org/abs/1902.10486.

[12] Alex KRIZHEVSKY, Ilya SUTSKEVER et Geoffrey E HINTON. “ImageNet Classification with Deep Convolutional Neural Networks”. In : *Advances in Neural Information Processing Systems*. Sous la dir. de F. PEREIRA et al. T. 25. Curran Associates, Inc., 2012, p. 1097-1105. URL : https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[13] Vincenzo LOMONACO, Davide MALTONI et Lorenzo PELLEGRINI. “Rehearsal-Free Continual Learning over Small Non-I.I.D. Batches”. In : *arXiv :1907.03799 [cs, stat]* (avr. 2020). arXiv : 1907.03799. URL : http://arxiv.org/abs/1907.03799.

[14] Jiangpeng HE et al. “Incremental Learning In Online Scenario”. In : *arXiv :2003.13191 [cs]* (mar. 2020). arXiv : 2003.13191. URL : http://arxiv.org/abs/2003.13191.

A Annexe

A.1 Convolutional Neural Networks (CNN)

Cette partie présente des notes prises sur les réseaux de neurones convolutifs. Elle permettra de faciliter la compréhension de certaines méthodes utilisant des CNN.

Notes réseaux de neurones convolutifs (CNN)

Ressources utilisées :

- https://cs231n.github.io/convolutional-networks/ Cours de Stanford
- https://openclassrooms.com/fr/courses/4470531-classez-et-segmentez-des-donnees-visuelles/5082166-quest-ce-quun-reseau-de-neurones-convolutif-ou-cnn cours d'OpenClassrooms
- https://medium.com/coinmonks/paper-review-of-alexnet-caffenet-winner-in-ilsvrc-2012-image-classification-b93598314160 Article sur AlexNet/CaffeNet

I. Prérequis : le template-matching

a) Template et filtres :

⇨ **Filtre** : outil permettant de réduire le bruit dans une image mais aussi **retrouver des motifs** dans une image.

⇨ Motif : représenté par une petite sous-image appelé template.

⇨ **Template-matching** : retrouver des templates donnés dans une image. Réalisé par l'utilisation de l'opérateur de corrélation croisée.

⇨ Opérateur de corrélation croisée pour un template de représentation matricielle H : transforme l'image de représentation matricielle X en une nouvelle image Y de la façon suivante

$$Y_{i,j} = \sum_{u=-k}^{k} \sum_{v=-k}^{k} H_{u,v} X_{i+u,j+v}$$

Revient à faire glisser H sur l'image X, à multiplier les pixels qui se superposent et à sommer ces produits. Y appelée carte de corrélation (même dimensions que X). Plus une région de Y est de valeur élevée, plus la région correspondante de X ressemble au template recherché.

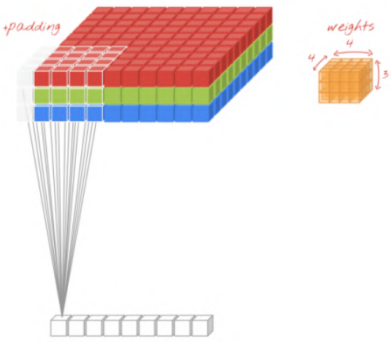
b) Produit de convolution :

⇨ Le **produit de convolution (2D)** entre un filtre de noyau H et une image X définit une nouvelle image Y ainsi :

$$Y_{i,j} = \sum_{u=-k}^{k} \sum_{v=-k}^{k} H_{u,v}. X_{i-u,j-v}$$

Revient à faire glisser H sur l'image X, à multiplier les pixels qui se superposent et à sommer ces produits. Idem que produit de corrélation mais H pivoté de 180° entre les 2.

⚠ Un noyau de produit de convolution est petit spatialement (typiquement 3x3 ou 5x5 de largeur x hauteur), mais prend s'étend selon toute la profondeur des données en entrée. Par ex, pour une image 224x224x3 (3 pour les pixels RGB), un filtre typique est de taille 5x5x3.



⇨ En **3D**, pour une image de dimensions WxHxD où **D** est la profondeur (typiquement D=3 pour une image RGB), le produit de convolution est :

$$Y_{i,j,z} = \sum_{l=1}^{z} \sum_{u=-k}^{k} \sum_{v=-k}^{k} H_{u,v,l} \cdot X_{i-u,j-v,l}$$

⇨ 2 paramètres au produit de convolution :

- *stride*, le pas, ie le nombre de pixels desquels se décale le noyau en glissant le long de l'image pour se déplacer.
- *zero-padding*, entier, entoure l'image d'un contour de zéros (d'épaisseur la valeur de zero-padding) pour pouvoir contrôler la taille de l'image de sortie.

⚠ Si on empile des couches avec zero-padding=0, les images se réduisent et on perd rapidement beaucoup d'informations.

⇨ Pour 1 filtre de noyau de dimensions $W_f \times H_f$ appliqué à une entrée de taille $W \times H$ avec une stride *S* et un zero-padding *P*, on obtient1 features map en sortie de taille :

$$\left(\frac{W-W_f+2P}{S}+1\right) \times \left(\frac{H-H_f+2P}{S}+1\right).$$

⚠ Choisir les paramètres S, P, W_f, H_f de sorte à obtenir des dimensions entières en sortie ! Souvent, avec *F* le côté d'un filtre carré, $P = (F-1)/2$ avec $S = 1$ est utilisé permet de conserver les dimensions de la feature map d'entrée.

c) Features d'une image :

⇨ Une (local) **feature** d'une image : partie de l'image (sous-image) correspondant à une zone intéressante. Elle participe à la classification d'une image. Même idée que template, mais plus générique (une feature peut être commune à plusieurs images même si les zones concernées ne sont pas strictement identiques, juste des zones qui partagent des motifs/paterns).

II. Les CNN

a) Principes généraux des CNN :

⇨ Le réseau est divisé en 2 blocs principaux, le premier étant caractéristique des CNN.

⇨ Un CNN est composé de 4 types de couches : couche de **convolution**, couche de **pooling**, couche de **correction** (ReLu), couche **fully-connected**. Il commence toujours par une couche de convolution et finit par une couche fully-connected.

⇨ En général, un réseau de neurones empile plusieurs couches de convolution et de correction ReLU, ajoute ensuite une couche de *pooling* (facultative), et répète ce motif plusieurs fois ; puis, il empile des couches *fully-connected*.

⇨ Performants principalement pour la classification d'images.

Notes réseaux de neurones convolutifs (CNN)

⚠ Sur les CNN initiaux on trouvait des couches de normalisation, c'était une normalisation dite « locale » (local response normalisation). Maintenant, on n'en utilise plus mais on normalise les batchs d'entraînement à la place.

b) La couche de convolution :

- ⇨ Première couche d'un CNN. But : étant donné un ensemble de features, repérer leur présence dans les images reçues en entrée.
- ⇨ Détection : Chaque ***feature*** définit un **noyau de convolution**, donc un filtre. On obtient pour chaque paire (image, filtre) une carte d'activation, ou ***feature map***, qui nous indique où se situent la *feature* du filtre dans l'image : plus la valeur est élevée, plus l'endroit correspondant dans l'image ressemble à la *feature*.

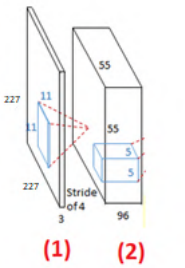
  Les noyaux des filtres désignent les poids de la couche de convolution. Ils sont initialisés puis mis à jour par **rétropropagation du gradient**.

- ⇨ En sortie d'une couche de convolution à N filtres, on a une couche de *N* features maps. Pour une feature map donnée, ses éléments sont des neurones. Couche qui suit la couche de convolution : a une profondeur de *N*, chaque feature map étant une couche de neurones.

  Chaque neurone en sortie de la couche de convolution n'est **relié qu'à une partie** de la de l'entrée de la couche de convolution, la partie qui a été convoluée par le noyau correspondant à la feature map pour l'obtenir.

  Ex : (1) Couche d'entrée du réseau d'une image 227x227 RGB (3 matrices, une par pixel), couche de convolution avec 96 features/noyaux de convolutions de taille 11x11x3, stride=4 en hauteur/largeur, padding=0.

  Pour chacun des 96 filtres : calcule le produit de convolution du filtre ➔96 features maps de taille 55x55 à la couche 2.

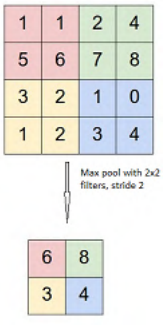


c) La couche de pooling :

- ⇨ Permet de réduire le nombre de calculs et le sur-apprentissage en rendant le réseau moins sensible à la position des features dans l'image. Souvent située entre 2 couches de convolution.

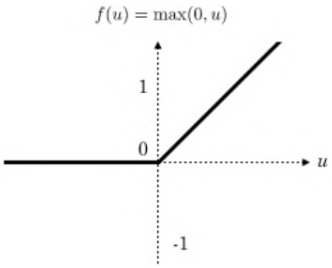
- ⇨ Principe : pour chaque feature map, en sortie une nouvelle feature map de taille réduite. On subdivise chaque feature map en entrée de la couche de pooling en de petites parties de même tailles (souvent carrées, 2x2, 3x3…). A chacune des petites parties est associée une valeur calculée grâce à une **fonction de pooling**. La feature map réduite en sortie est constituée de ces valeurs.

- ⇨ Plusieurs types de fonctions de pooling : la plus utilisée est la fonction **max** (maxpool), à chaque cellule associe la valeur maximale de ses éléments.



⇨ 2 possibilités souvent utilisées : diviser en cellules de 2x2 avec un pas de 2 (pas de chevauchement) ou cellules 3x3 avec pas de 2 (chevauchement de 1 à chaque fois).

d) Couche de correction (ReLu) :



⇨ Elle applique simplement à toutes les features map en entrée la fonction d'activation **ReLu**(x)= max(0,x). Induit une non-linéarité et supprime les valeurs négatives.

e) Couche fully-connected :

⇨ Couche des réseaux de neurones simples. Prend en entrée un vecteur et produit un vecteur en sortie. Chaque valeur en entrée est connectée avec toutes les valeurs en sortie.

⇨ Applique une **combinaison linéaire**, puis une **fonction d'activation** (softmax=max(0,x) , sigmoide ?). La combinaison linéaire est réalisée par multiplication du vecteur d'entrée par une matrice de poids, en prenant en compte le biais.

⇨ Dernière couche du CNN : couche fully-connected, sa sortie est le vecteur des probabilités d'appartenance à chaque classe.

III. Applications : AlexNet/ CaffeNet :

a) AlexNet :

i. Paramètres utilisés et traitement des données :

⇨ Local response normalization

⇨ Data augmentation :

- les images 224x224 sont extraites d'images 256x256. On multiplie le nombre d'exemples du dataset par 2048 en faisant ça.
- Changer l'intensité : Pour chaque image, additionner les matrices RGB de l'image à la matrice obtenue ainsi :

$$[\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3][\alpha_1\lambda_1, \alpha_2\lambda_2, \alpha_3\lambda_3]^T$$

where pi and λi are ith eigenvector and eigenvalue of the 3×3 covariance matrix of RGB pixel values, respectively, and αi is the random variable with mean 0 and standard variation 0.1.

⇨ Dropout : une couche utilisant le dropout a des chances définit une probabilité que chaque neurone soit entraîné. 2 premières couches fully connected utilisent un dropout de proba 0.5 pendant l'entraînement. Permet de ne pas trop dépendre sur un neurone « fort ».

Notes réseaux de neurones convolutifs (CNN)

- ⇨ Batch size: 128
  Momentum v: 0.9
  Weight Decay: 0.0005
  Learning rate ϵ: 0.01, reduced by 10 manually when validation error rate stopped improving, and reduced by 3 times.
- ⇨ Entraîné pendant environ 90 cycles.

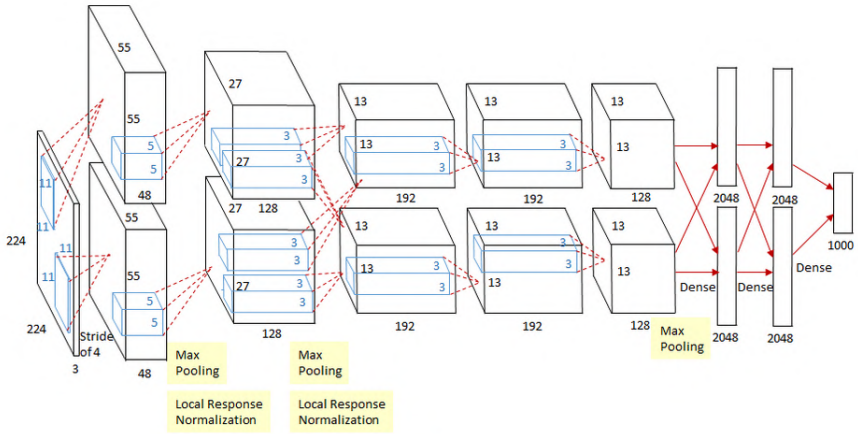
$$v_{i+1} := 0.9 \cdot v_i - 0.0005 \cdot \epsilon \cdot w_i - \epsilon \cdot \left\langle \frac{\partial L}{\partial w}\Big|_{w_i} \right\rangle_{D_i}$$
$$w_{i+1} := w_i + v_{i+1}$$

The update of momentum v and weight w

ii. Structure

- ⇨ Architecture à 8 couches.
- ⇨ 2 GPU étaient utilisés à l'époque (problèmes de mémoire), la structure est donc séparée en 2 à l'issue de la première couche.

⚠ L'architecture proposée avec des images 224x224x3 en entrée n'est pas possible (le calcul pour la première couche de convolution des dimensions de la sortie ne donne pas des entiers). Avec 227x227, fonctionnerait (sorte de padding-zero=3 juste sur côté gauche de l'image et la hauteur).



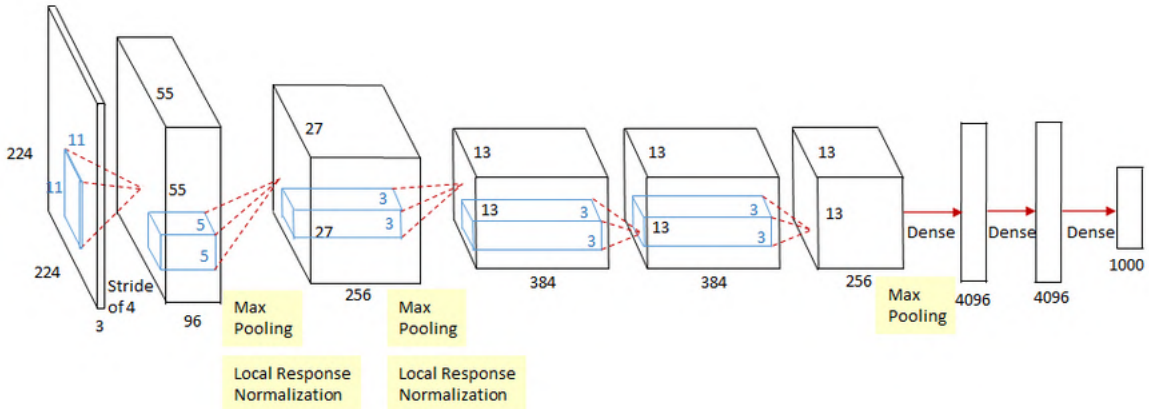
AlexNet

Opération/couche	1 **Conv**	2 **Conv**	3 **Conv**	4 **Conv**	5 **Conv**	6 **Full-con**	7 **Full-con**	8 **Full-con + ReLu**
Nombre & paramètres noyaux	96 noyaux 11x11x3 Stride : 4 Pad : 0 Features maps : 55×55×96	96 noyaux 11x11x3 Stride : 4 Pad : 0 Features maps : 55×55×96	384 noyaux 3×3×256 Stride : 1 Pad : 1 Features maps : 13×13×384	384 noyaux 13×13×384 Stride : 1 Pad : 1 Features maps : 13×13×384	96 noyaux 11x11x3 Stride : 4 Pad : 0 Features maps : 55×55×96	4096 neurones	4096 neurones	N neurones Softmax Sortie : 1000 classes
Paramètres Maxpooling	3×3 Stride : 2 Features maps : 27×27×96	3×3 Stride : 2 Features maps : 13×13×256			3×3 Stride: 2 Features maps : 6×6×256			

Paramètres Normalization	Features map : 27×27×96	Features map : 13×13×256						

b) CaffeNet :

⇨ Version à 1 GPU de AlexNet, structure identique mais avec un seul GPU.



A.2 Méthode de Descente du Gradient (SGD)

L'algorithme de descente du gradient permet de minimiser la fonction de coût L de paramètre $\theta = (\theta_1, ..., \theta_n)^t \in \mathbb{R}^n$ où les $\theta_1, ..., \theta_n$ sont les poids du modèle. A chaque itération de l'algorithme, les poids du modèles sont mis à jour simultanément [1] en suivant la formule :

$$\forall i \in \{1, ..., n\}, \theta_i = \theta_i - \alpha \frac{\partial L}{\partial \theta_i}(\theta)$$

où α est un paramètre fixé (hyperparamètre du modèle), appelé taux d'apprentissage (*learning rate*).

Il est donc nécessaire de calculer, à chaque itération de l'algorithme, le gradient g de la fonction de coût [2] afin d'utiliser ses composantes (les $\frac{\partial L}{\partial \theta_i}(\theta)$) pour mettre à jour les poids du modèle et minimiser la fonction de coût L.

École centrale de Lyon
36, Avenue Guy de Collongue
code du labo concerné
69134 Écully

1. jusqu'à convergence du modèle vers le minimum de L
2. par une méthode dite de rétropropagation du gradient