IDIAP RESEARCH REPORT

IMPLéMENTATION D'UN ALGORITHME DE RéDUCTION DE TAILLE DES RéSEAUX DE NEURONES

François Marelli

Idiap-RR-03-2018

MARCH 2018

Centre du Parc, Rue Marconi 19, P.O. Box 592, CH - 1920 Martigny
T +41 27 721 77 11 F +41 27 721 77 12 info@idiap.ch www.idiap.ch

UMONS
Université de Mons

Faculté Polytechnique

Implémentation d'un algorithme de réduction de taille des réseaux de neurones

Institut de recherche Idiap

Rapport de stage de MA2

François MARELLI

POLYTECH MONS

Superviseurs: Pr. Hervé BOURLARD, Dr. François FLEURET
Référent académique: Pr. T. DUTOIT

Année académique 2017-2018

ACADÉMIE UNIVERSITAIRE WALLONIE-BRUXELLES

idiap MARTIGNY SWITZERLAND

1 Introduction

Les réseaux de neurones artificiels constituent une technologie très actuelle. Leurs bonnes performances permettent de traiter des problèmes complexes dans des domaines très variés. Leurs applications vont de la reconnaissance vocale de nos téléphones à la compréhension de l'environnement pour les voitures autonomes, en passant par la vérification d'identité des systèmes de sécurité. C'est pourquoi cette technologie est au centre de nombreuses recherches qui tendent à mieux comprendre son fonctionnement et à l'améliorer.

La tendance actuelle est que des réseaux de neurones de plus en plus volumineux sont utilisés pour résoudre des problèmes de plus en plus complexes. Cela peut poser un souci quant à l'intégration de ces technologies dans des systèmes disposant de peu de mémoire et/ou de peu de puissance de calcul. C'est l'une des raisons qui justifie l'utilité des techniques de réduction de taille des réseaux de neurones artificiels. Celles-ci visent à réduire le volume des systèmes sans détériorer leurs performances.

Ce document décrit les résultats obtenus lors d'un stage de deux mois ayant pour objectifs d'implémenter et de valider une technique de réduction de réseaux de neurones décrite dans l'article [ZAP16]. D'abord, la problématique est exposée et analysée d'un point de vue technique. Ensuite les concepts théoriques sur lesquels ce travail est basé, sont rappelés brièvement. Enfin, les résultats techniques sont exposés et analysés.

2 Objectifs

L'objet de ce travail est d'implémenter un algorithme de réduction de taille de réseaux de neurones basé sur l'article [ZAP16]. Une fois l'implémentation fonctionnelle, celle-ci sera validée sur plusieurs bases de données et sur différentes structures de réseaux.

La première étape de validation consistera à reproduire les résultats exposés dans l'article, suite à quoi l'algorithme sera étendu à d'autres applications de complexité croissante. Une étude des paramètres et du comportement de l'algorithme sera aussi réalisée afin de mieux cerner son fonctionnement. Le délivrable attendu consiste en un outil de réduction de réseaux pouvant être réutilisé dans divers projets, sous forme d'un code source documenté.

L'utilisation de méthodes de réduction de réseaux de neurones artificiels a trois avantages principaux :

- *diminution de l'espace mémoire requis* : en réduisant la taille du réseau, et donc le nombre de paramètres qu'il contient, l'espace nécessaire pour stocker ceux-ci dans une mémoire est réduit proportionnellement. Cela permet d'intégrer le réseau sur des systèmes disposant de moins de mémoire disponible.
- *réduction des ressources de calcul nécessaires* : en réduisant la complexité du réseau, on réduit le nombre d'opérations mathématiques utilisées dans le système. Cela permet d'une part d'accélérer le calcul d'une solution par le réseau et d'autre part d'utiliser ce réseau sur des systèmes dont les capacités de calcul sont plus réduites.
- *accélération de l'entraînement du réseau :* le temps que prend l'entraînement d'un réseau de neurones augmente fortement avec sa taille. Cela est lié à la difficulté croissante d'optimisation d'un système plus complexe. En réduisant la taille du réseau, le temps d'entraînement de celui-ci est aussi diminué.

La complexité de la réduction de taille des réseaux de neurones consiste à conserver des performances correctes tout en utilisant un réseau plus petit. L'objectif est donc, partant d'un système complexe, de réduire celui-ci au maximum sans affecter son efficacité.

3 Concepts théoriques

Cette section reprend brièvement les différents concepts théoriques utilisés dans ce travail afin de fixer la nomenclature utilisée. Ceux-ci sont aussi abordés d'un point de vue technique par rapport à leur implémentation, en mettant en évidence les parties utilisées dans ce projet.

3.1 Les réseaux de neurones artificiels

Les réseaux de neurones artificiels sont des algorithmes qui tentent d'imiter le fonctionnement d'un cerveau biologique. Ils sont constitués d'unités de calcul simples, appelées neurones, connectées les unes aux autres pour former une structure de réseau. Les neurones possèdent plusieurs entrées et une sortie. Ils effectuent une combinaison linéaire de leurs entrées, suivie d'une fonction non linéaire appelée activation. Les neurones sont regroupés en couches, les sorties d'une couche constituant les entrées de la suivante, comme représenté à la figure 1. La profondeur d'un réseau représente le

nombre de couches qui le constitue, et la largeur d'une couche désigne son nombre de sorties. Ces systèmes sont entraînés pour des tâches spécifiques par l'optimisation d'une fonction de coût à minimiser. Cet entraînement se fait par époques, qui désignent le passage de tous les échantillons de la base de données d'entraînement à travers le réseau.

On distingue différents types de couches, en fonction de leur action et de leur structure. Celles utilisées au cours de ce projet sont listées ci-dessous :

— *la couche linéaire* : chaque neurone effectue un produit scalaire entre ses entrées et son vecteur de poids, et y additionne un biais. Une telle couche est implémentée sous forme d'une matrice reprenant les poids de tous les neurones, et d'un vecteur reprenant les biais. Ses entrées et sorties sont des vecteurs.

— *la couche de convolution* : chaque neurone effectue un produit de convolution entre chacune de ses entrées et le noyau de convolution qui lui est associé, et y additionne un biais. Une telle couche est implémentée sous forme d'une matrice 4D reprenant les neurones associés à chaque sortie, et d'un vecteur reprenant les biais. Ses entrées et sorties sont des matrices tridimensionnelles.

La théorie complète des réseaux de neurones est détaillée dans [GBC16]. L'utilisation des réseaux convolutionels est étudiée plus avant dans [SLJ+15].

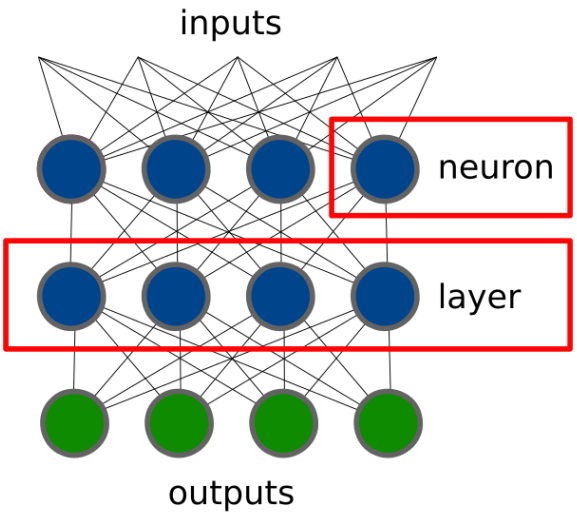

FIGURE 1 – Réseau de neurones artificiels

3.2 Les réseaux résiduels

Les réseaux résiduels visent à réduire la complexité d'optimisation des paramètres pour les réseaux très profonds. Ils utilisent pour cela une structure en blocs résiduels, comme représenté à la figure 2. Cette structure contient un ensemble de couches conventionnelles, en parallèle avec un chemin où l'entrée est simplement recopiée. Ces deux résultats sont additionnés pour obtenir la sortie du bloc. Un ensemble de blocs résiduels en série forme un réseau résiduel. L'implémentation des couches dans un réseau résiduel ne diffère pas de celle décrite précédemment.

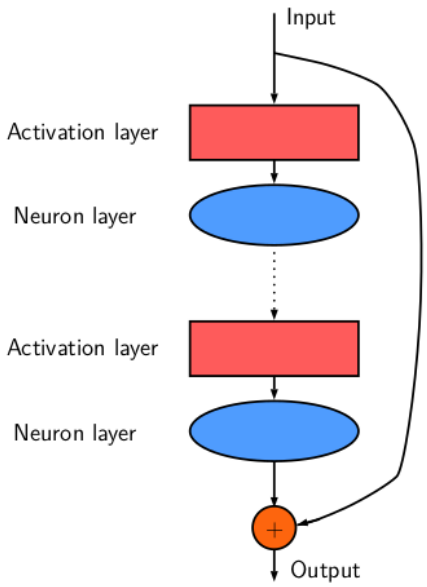

FIGURE 2 – Bloc résiduel

Une étude plus détaillée des réseaux résiduels et de leur fonctionnement est donnée dans [HZRS16].

3.3 Algorithme de réduction

L'algorithme de réduction décrit dans [ZAP16] a pour objectif de supprimer des neurones au sein du réseau afin de réduire sa taille. Cela est possible en ajoutant un terme de pénalité à la fonction de coût de l'optimiseur : si cette pénalité est proportionnelle à la somme des poids des neurones dans le réseau, ceux-ci tendront vers zéro après suffisamment d'époques. Il suffit alors de supprimer du réseau tous les neurones dont les poids sont nuls car ils n'apportent pas d'information.

Le développement mathématique de cette méthode est détaillé dans [ZAP16] et mène à la formule suivante :

$$w_{lj}^{k+} = \max\left\{\left\|w_{lj}^{k}\right\| - \tau, 0\right\} \frac{w_{lj}^{k}}{\left\|w_{lj}^{k}\right\|} \tag{1}$$

Avec :
— w_{lj}^{k} le neurone j de la couche l à l'époque k
— τ le seuil de réduction, paramètre à fixer
— $\|\cdot\|$ la norme L_2

Cette formule doit être appliquée à toutes les couches à réduire à la fin d'une époque d'entraînement. Elle consiste à normaliser tous les neurones de la couche, et à les supprimer si leur poids descend en-dessous d'un seuil. Cela peut s'interpréter comme un tri sélectif en fonction de la pertinence des neurones au sein du réseau. Un neurone de poids faible apporte peu d'information à la couche suivante, et peut donc être enlevé sans trop perdre de précision.

4 Résultats

Cette section décrit et analyse les différents résultats obtenus au cours du projet. Les choix techniques pris tout au long du travail sont exposés et justifiés.

4.1 Implémentation

Parmi les différents frameworks utilisés à l'Idiap, le choix pour l'implémentation s'est porté sur PyTorch, un ensemble de librairies liées au machine learning en python. La principale alternative est Tensorflow, qui offre des fonctionnalités équivalentes et propose une syntaxe similaire. PyTorch a été choisi pour deux raisons principales :

1. PyTorch est le framework le plus utilisé au sein du group de machine learning. Cela facilite donc l'intégration de l'outil aux autres projets et la comparaison des résultats obtenus.
2. En comparaison à Tensorflow, PyTorch permet plus facilement d'accéder aux algorithmes d'entraînement en raison de son implémentation. Modifier les procédures pour y intégrer la réduction y est donc plus direct.

Les outils de calcul tensoriel inclus dans PyTorch sont exploités afin d'optimiser le temps d'exécution de la réduction.

L'implémentation de l'algorithme prend la forme d'un ensemble de fonctions à ajouter dans le code source après la procédure d'entraînement d'une époque. Cela permet d'intégrer la réduction à des réseaux déjà existants, tout en conservant les implémentations déjà réalisées. Cette possibilité permet de tester rapidement l'algorithme sur des anciens projets, et facilite l'apprentissage de son utilisation.

L'algorithme s'applique en deux étapes : d'abord, l'application de la formule 1 aux différentes couches, et ensuite la suppression des neurones nuls au sein du réseau. Le fait d'enlever les neurones inutiles au fur et à mesure plutôt qu'à la fin de l'entraînement permet d'accélérer celui-ci. En effet, si on réduit la taille du réseau, le temps nécessaire pour entraîner une époque est lui aussi réduit (moins de calculs, moins d'espace mémoire, moins de paramètres pour l'optimiseur).

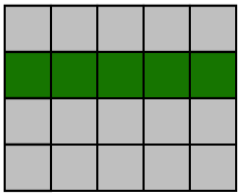

FIGURE 3 – Neurone dans une couche linéaire

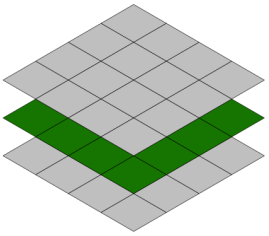

FIGURE 4 – Neurone dans une couche convolutionnelle

La représentation d'un neurone (w_{lj}^{k}) diffère selon le type de couche utilisé. Dans le cas d'une couche linéaire, ceux-ci constituent les lignes de la matrice des poids de la couche (voir figure 3). La norme à appliquer est donc une simple norme vectorielle L2. Dans le cas d'une couche convolutionnelle, les neurones sont représentés par leurs noyaux de convolution, soit un étage dans la matrice 4D des poids (voir figure 4). La norme à appliquer est alors la norme Frobenius. L'implémentation proposée détecte automatiquement le type de couche en fonction de sa matrice de poids et utilise automatiquement la norme correspondante.

Trois paramètres importants doivent être fixés pour utiliser l'algorithme

implémenté. Il s'agit du nombre d'époques à entraîner, du seuil de réduction et de l'intervalle laissé entre deux réductions. Leur influence respective est étudiée à la section 4.6.

4.2 Reproduction des résultats de l'article

La première étape de validation de l'implémentation consiste à reproduire des résultats similaires à ceux publiés dans [ZAP16]. Pour cela, l'algorithme a été appliqué à la base de données MNIST [LCB10]. Cette base de donnée contient des images 28x28 pixels représentant des chiffres écrits à la main, comme on peut le voir à la figure 5. La structure du réseau utilisé est l'architecture LeNet [LJB+95]. Il s'agit d'un réseau convolutionnel simple et peu profond, car le problème traité est très facile à résoudre.

Les résultats publiés dans l'article mentionnent une réduction de la taille du réseau de plus de 90%, avec un taux d'erreur restant sous la barre du pourcent.

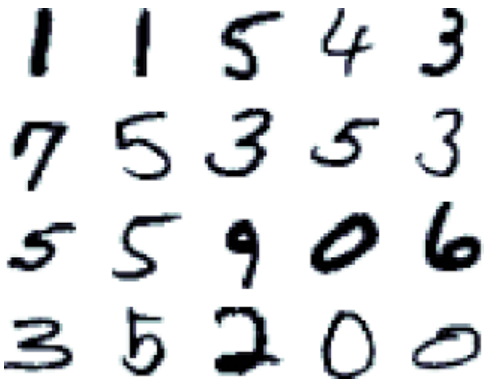

Figure 5 – Exemples d'échantillons MNIST

Afin de pouvoir mesurer et comparer l'efficacité de la méthode, une procédure de mesure est fixée. La réduction de la taille du réseau est calculée en comparant le nombre total de paramètres du réseau (poids des neurones, biais, etc.) avant et après la réduction, et est exprimée en pourcent. Le taux d'erreur du réseau est mesuré sur l'ensemble de la base de données de validation. Les différents résultats exposés sont moyennés sur 10 expériences pour atténuer l'effet de l'initialisation aléatoire du réseau.

Le réseau implémenté pour validation possède un taux d'erreur initial de 0.85%. Après ajout de l'algorithme de réduction, la taille du réseau est diminuée de 84% tandis que le taux d'erreur passe à 0.93%. Ces résultats, du

même ordre de grandeur que ceux mentionnés dans l'article, valident l'implémentation réalisée et confirment l'efficacité de la méthode de réduction utilisée.

Il est à noter que les résultats mentionnés ci-avant sont moins bons que ceux exposés dans l'article. Cela s'explique par le fait que le but principal de cette validation n'est pas de trouver l'optimum des performances de l'algorithme, mais d'avoir une preuve de son bon fonctionnement. Les tests ont donc été effectués avec un nombre limité de valeurs pour les différents paramètres afin de réduire le temps nécessaire pour faire tourner les mesures. Cette recherche grossière ne permet pas de trouver les meilleures performances possibles, et des meilleurs résultats peuvent être atteints en optimisant plus finement les paramètres de la méthode. Toutefois, ces mesures donnent une bonne idée des possibilités de l'algorithme implémenté.

4.3 Amélioration de la méthode initiale

Afin d'optimiser les performances de l'algorithme implémenté, plusieurs variations de celui-ci ont été proposées et comparées. Les méthodes envisagées sont les suivantes :

1. *l'algorithme de base* : appliquer l'algorithme présenté dans l'article sans modifications.
2. *l'algorithme renversé* : appliquer l'algorithme en groupant les poids vis-à-vis des entrées et non pas vis-à-vis des sorties. Cela revient à grouper les poids en colonnes plutôt qu'en lignes. Cette proposition part de l'hypothèse que supprimer les entrées non pertinentes permet aussi de réduire le réseau sans perdre trop d'information.
3. *l'algorithme découpé* : appliquer l'algorithme de base sur chaque couche séparément plutôt que sur tout le réseau à la fois. La procédure d'entraînement est découpée en autant de phases qu'il y a de couches dans le réseau. Cette méthode est inspirée d'une mention dans [ZAP16].
4. *l'algorithme relatif* : appliquer l'algorithme de base où le seuil de réduction est remplacé par un ratio. Les poids ne sont plus comparés à une constante, mais à un pourcentage du poids maximum de leur couche.

Les différentes méthodes ont été testées sur le même réseau qu'au point différent (LeNet - MNIST). Les résultats obtenus sont les suivants :

	réduction [%]	erreur[%]
Basique	84.0	0.93
Renversé	87.6	0.97
Découpé	87.2	1.31
Relatif	85.9	0.96

On constate que l'algorithme renversé donne des résultats similaires à ceux obtenus avec la version de base. Cependant, cette méthode se révèle beaucoup plus sensible aux variations des paramètres utilisés, et demande une recherche plus fine pour obtenir des performances correctes. Elle n'apporte donc aucune amélioration à l'algorithme.

L'algorithme découpé donne des résultats peu satisfaisants. En effet, il est impossible de descendre en-dessous de 1% d'erreur, ce qui est notre contrainte d'efficacité. Cette méthode dégrade donc les performances de l'algorithme initial.

L'algorithme relatif donne des résultats très similaires à la version initiale. Il ne montre pas de sensibilité accrue aux variations de paramètres. Par contre, il apporte une amélioration notable à l'algorithme initial. En effet, celui-ci implique de fixer le seuil de réduction de manière empirique et pour chacune des couches du réseau. Ce seuil peut varier en fonction de la taille et du type de couche utilisé, et choisir un seuil adéquat pour chaque couche du réseau peut s'avérer fastidieux. La version relative de l'algorithme demande de fixer un ratio pour l'ensemble du réseau. Cela permet au seuil de s'adapter automatiquement à chaque couche, quels que soient son type et sa taille. De plus, avec un seuil fixe trop élevé, il est possible qu'une couche disparaisse totalement du réseau, rendant son fonctionnement impossible. Ce problème n'apparaît pas avec le seuil relatif car le neurone de poids maximal de chaque couche ne peut pas être supprimé.

En raison des différents avantages de l'algorithme relatif, celui-ci est retenu comme implémentation principale dans le projet. Tous les résultats à suivre sont obtenus en utilisant cette méthode comme algorithme de réduction. Les autres variations envisagées ne sont pas inclues dans le délivrable final car elles n'apportent aucun avantage.

4.4 Application à Cifar-10

Pour mesurer l'efficacité de cette méthode appliquée à des problèmes plus complexes, l'algorithme a été utilisé sur des réseaux plus volumineux

entraînés sur la base de données Cifar-10 [KH09]. Cette base de données est constituée d'images en couleur 32x32 pixels triées en dix classes d'objets. La figure 6 montre les classes existantes ainsi que plusieurs échantillons.

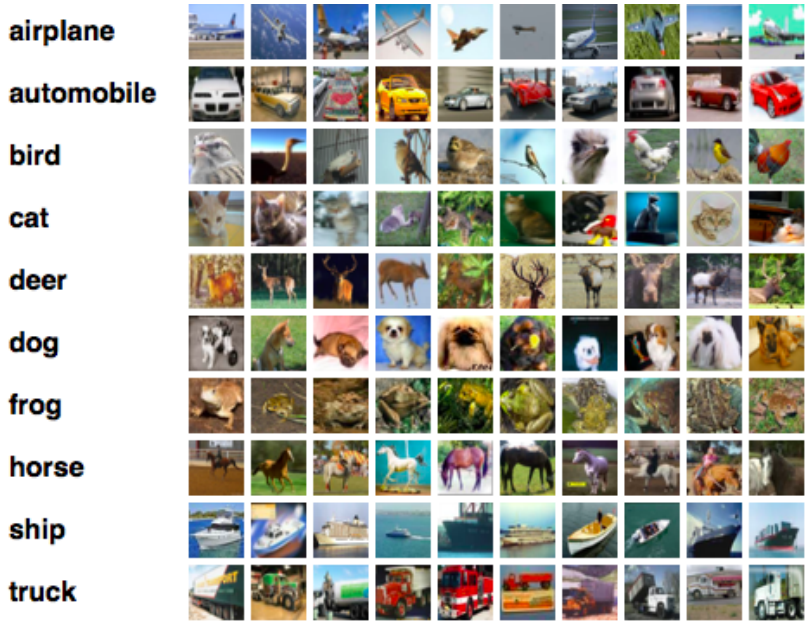

FIGURE 6 – Exemples d'échantillons Cifar-10

Il s'agit d'un problème plus complexe à résoudre que celui posé par MNIST, car les objets à détecter sont plus détaillés. La composante couleur des images ajoute une dimension supplémentaire aux données, qui sont représentées sous forme de matrices tridimensionnelles. La précision d'un humain devant reclasser les images est de l'ordre de 94%. Les erreurs sont dues à la faible résolution des images qui sont parfois illisibles. En raison de la complexité de la tâche, les réseaux de neurones artificiels employés sont plus volumineux et plus profonds, ce qui leur permet un plus grand degré de liberté pour trouver une solution adéquate.

L'algorithme de réduction a été appliqué à deux réseaux convolutionnels de structures différentes :

— *LeNet* : il s'agit du même type de réseau que celui utilisé pour MNIST. Il comporte deux couches de convolution et trois couches linéaires.
— *Cifar-quick* : il s'agit d'une structure comportant trois couches convolutionnelles larges suivies de deux couches linéaires.

Ces deux réseaux possèdent des structures encore simples et ne fournissent pas les meilleures performances de l'état de l'art actuel. Némanmoins, ils permettent d'évaluer l'effet de l'algorithme appliqué à des réseaux plus

volumineux quel le précédent. L'implémentation de l'algorithme de réduction a été directe pour ces réseaux, car ils utilisent une structure standard constituée de couches linéaires et convolutionnelles comme dans les cas déjà traités. Les résultats mesurés sur ces structures sont les suivants :

	Performances initiales	*Après réduction*	
	erreur [%]	réduction [%]	erreur [%]
LeNet	44.7	85.5	39.3
Cifar-quick	29.6	51.3	27.1

On constate qu'en effet, avec des erreurs initiales de 45 et 30% respectivement, ces réseaux ne sont pas extrêmement efficaces. L'utilisation de l'algorithme de réduction a permis de réduire leur taille de 86 et 51%, ce qui constitue un gain de mémoire significatif. Ce dernier résultat est par ailleurs comparable avec les chiffres mentionnés dans [ZAP16], où un réseau de type Cifar-quick a pu être réduit de 50% en conservant moins de 30% d'erreur.

Un effet secondaire intéressant de la réduction constaté sur ces réseaux est une amélioration de leur précision. En effet, pour la structure LeNet, l'erreur chute de 5.4%, et pour Cifar-quick elle diminue de 2.5%. Cela montre que la réduction a un effet régularisant sur le réseau. Les algorithmes courants de régularisation visent à réduire l'over-fitting (sur-entraînement du réseau qui détériore ses performances) en imposant des contraintes sur la fonction de coût de l'optimiseur. L'algorithme de réduction utilisé est basé sur un principe similaire, et peut donc être utilisé comme régulariseur pour certaines applications.

4.5 Application à un réseau résiduel

Afin de tester la généralisation de l'algorithme à des réseaux de structures différentes et pour mesurer son efficacité dans un cas d'application réaliste, celui-ci a été appliqué à un réseau résiduel entraîné sur Cifar-10. L'état de l'art actuel sur cette base de données est de 2.9% d'erreur, réalisé grâce à une régularization de type shake-shake [Gas17]. Le réseau résiduel utilisé pour le test est tiré de [ZK16], et offre de très bonnes performances avec seulement 3.9% d'erreur.

L'application de l'algorithme de réduction à un réseau de type résiduel sort du cadre de l'article initial [ZAP16]. En effet, seule l'application de la réduction à des couches standard linéaires et convolutionnelles y est envisagée. La structure d'un bloc résiduel pose un problème lors de la réduction

de la couche de sortie du bloc. En effet, la sortie de celle-ci est additionnée aux entrées du bloc résiduel (voir figure 2). Cela implique une cohérence pour la dimension des données à additionner. Cependant, l'algorithme de réduction modifie le nombre de neurones d'une couche, et donc la dimension de ses sorties. Il est donc possible de réduire directement les couches intermédiaires, mais la réduction de la couche de sortie de chaque bloc demande plus de réflexion.

La solution proposée consiste à remplacer l'addition par un bloc d'addition intelligente. Ce bloc garde en mémoire la trace des sorties supprimées, et permet d'additionner la sortie de la dernière couche à l'entrée du bloc même si ces données ne sont pas de dimensions identiques. En effet, le bloc réalise une addition sélective avec les sorties conservées, et conserve les autres données inchangées. Cela permet de conserver la structure résiduelle du réseau tout en réduisant toutes les couches afin de minimiser sa taille. Le bloc d'addition sélective pénalise légèrement le temps d'exécution du réseau, mais ce désavantage est compensé par le gain occasionné par la réduction de taille des couches de sortie. La structure obtenue est représentée à la figure 7.

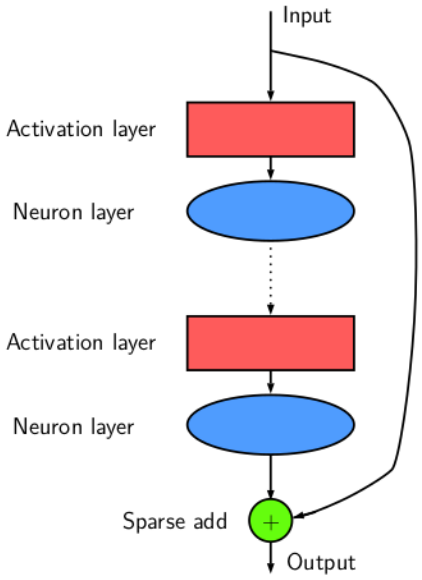

FIGURE 7 – Bloc résiduel

La réduction appliquée au réseau résiduel de [ZK16] a permis de réduire la taille de celui-ci de 91.2%, tout en conservant une erreur raisonnable de 4.9%. Cela représente donc une perte de moins de 1% de précision, tandis que la taille totale du réseau est divisée par dix. Ces résultats montrent que l'implémentation de la réduction peut être généralisée avec succès à des réseaux autres que ceux étudiés dans [ZAP16].

Il est intéressant d'observer l'évolution de la forme du réseau liée à sa réduction de taille. Les figures 8 et 9 représentent la largeur des couches du réseau selon leur profondeur dans le réseau, pour la structure initiale et pour le réseau réduit respectivement.

Le réseau initial est divisé en trois sections de largeur constante, de plus en plus larges avec la profondeur du réseau. La section la plus large est de dimension 640. Cette architecture est directement issue de [ZK16]. La structure réduite est intéressante à analyser. En effet, on y distingue aussi trois sections différentes :

1. une section de faible largeur à l'entrée du réseau
2. une section constituée d'auto-encodeurs (succession de couches larges et étroites) au centre du réseau
3. une section contenant les couches les plus larges à la sortie du réseau

Cette architecture correspond à des résultats expérimentaux visant à optimiser l'efficacité des réseaux de neurones profonds. Il est intéressant de noter que l'algorithme de réduction a trouvé par lui-même cette structure optimisée. Il est donc envisageable d'utiliser cette méthode pour rechercher des architectures spécifiques permettant d'optimiser les performances de différents types de réseau.

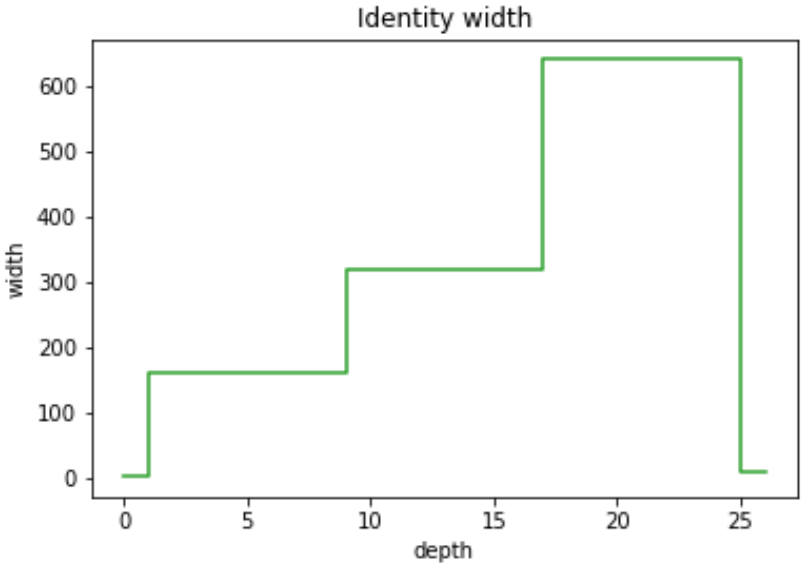

FIGURE 8 – Structure du réseau initial

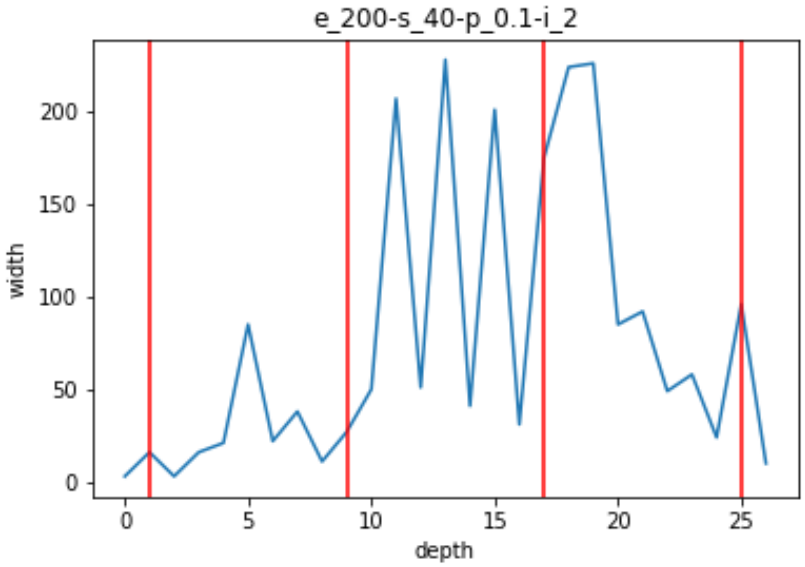

FIGURE 9 – Structure du réseau réduit

4.6 Influence des paramètres

L'implémentation de l'algorithme demande de fixer plusieurs paramètres avant de lancer la séquence d'entraînement. Ceux-ci sont :

— le nombre d'époques à entraîner
— le seuil de réduction utilisé
— l'intervalle en époques entre deux applications de la réduction

Afin de comprendre dans quelle mesure ceux-ci influencent les résultats en termes de réduction et d'efficacité, des mesures ont été réalisées sur le réseau LeNet-MNIST. Un ensemble de 50 mesures ont été réalisées en faisant varier les paramètres dans les gammes suivantes :

époques	[10 – 40]
seuil	[0.15 – 0.25]
intervalle	[1 – 2]

Les influences ont été calculées par méthode des moindres carrés en utilisant le modèle d'interactions suivant :

$$f = a_0 + a_1 E + a_2 P + a_3 I + a_4 EP + a_5 EI + a_6 PI + a_7 EPI$$

Où E, P et I représentent respectivement le nombre d'époques, le seuil de réduction et l'intervalle entre deux réductions. Le même modèle a été

utilisé pour calculer l'influence vis-à-vis de la précision et de la réduction des réseaux.

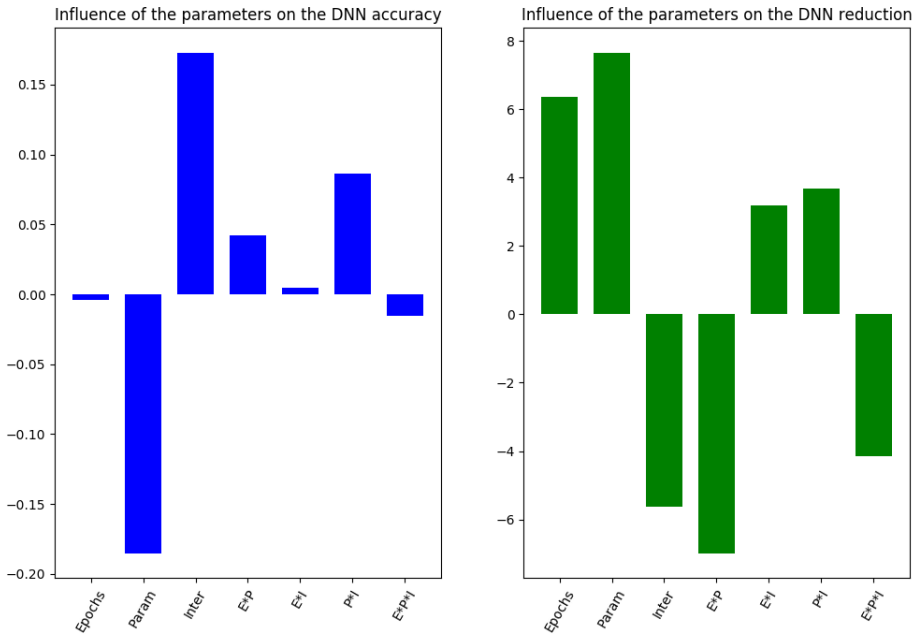

FIGURE 10 – Influence des paramètres sur la précision et la réduction

La figure 10 montre les coefficients d'influence mesurés. On y constate que le seuil de réduction a une influence positive sur le taux de réduction, mais qu'il réduit la précision du réseau. En effet, un réseau trop fortement réduit aura trop peu de neurones que pour remplir sa tâche correctement. On voit aussi qu'augmenter l'intervalle entre deux réductions aide à avoir une meilleure précision, mais diminue le taux de réduction atteint. On constate enfin qu'augmenter le nombre d'époques permet une meilleure réduction sans trop affecter la précision.

Les influences mutuelles sont difficiles à interpréter, et il faut garder à l'esprit que le modèle utilisé est simpliste. Les résultats fournis sont plutôt qualitatifs et permettent de se faire une idée du comportement de l'algorithme en fonction de ses paramètres. Toutefois, une analyse plus poussée serait nécéssaire pour pouvoir quantifier exactement cette influence, et analyser l'effet des influences mutuelles. Il est aussi à noter que les figures sont issues d'expériences réalisées sur un seul type de réseau. Bien que les conclusions tirées ci-avant se répètent empiriquement avec les autres structures, les influences exactes n'y ont pas été mesurées.

Expérimentalement, deux stratégies se sont avérées efficaces pour réduire les réseaux. La première consiste à choisir un seuil de réduction petit (de l'ordre de 10%) et un faible intervalle (de l'ordre de 2 époques). Cela mène à une réduction de faible ampleur mais effectuée fréquemment. La seconde stratégie consiste à choisir un seuil de réduction fort (de l'ordre de 80%) et un plus grand intervalle (8 à 10 époques). De la sorte, une forte réduction est appliquée au réseau mais à intervalles espacés. Les deux méthodes citées mènent à des résultats similaires et ne peuvent être discriminées a priori.

4.7 Dynamique d'entraînement

Pour mieux comprendre le fonctionnement de l'algorithme utilisé, il est intéressant de se pencher sur la dynamique du système lors de l'entraînement du réseau. Pour cela, deux variables ont été mesurées au cours de l'entraînement : la fonction de coût et le taux de réduction du réseau. Afin de disposer d'un intervalle de temps suffisant pour bien visualiser l'évolution du système, les mesures ont été effectuées sur le réseau résiduel présenté à la section 4.5.

La figure 11 représente l'évolution de la fonction coût de l'optimiseur en fonction de l'époque d'entraînement. Les trois premiers paliers qu'on y distingue ne sont pas liés à la réduction, mais à la politique d'entraînement utilisée (ils correspondent à des modifications du taux d'apprentissage du réseau).

Le premier effet visible de la réduction est l'oscillation en dents de scie qu'on peut voir sur la courbe. Cet effet est lié à l'intervalle entre deux réductions, qui a été fixé à 2 pour cette mesure. Dès lors, une époque sur deux n'est pas réduite, et l'optimiseur tente de minimiser le coût, ce qui diminue celui-ci. L'époque suivante est réduite, et l'algorithme de réduction ne cherche pas à optimiser le coût mais à diminuer la taille du réseau. Cela fait donc remonter le coût, et donne cette forme en dents de scie à la courbe.

Le second effet notable est la dynamique en fin d'entraînement. Comme la réduction ne tient pas compte de l'optimiseur, il est important de désactiver l'algorithme de réduction à la fin de l'entraînement afin de permettre une optimisation plus précise du coût par l'optimiseur. Sur la figure 11, la droite verticale située à l'époque 160 marque l'arrêt de l'algorithme de réduction. On constate qu'après cet instant, la fonction coût est encore réduite de moitié par rapport à son dernier palier. Afin d'obtenir les meilleures performances du réseau, il est donc très important de prévoir ces époques entraînées sans réduction à la fin du processus.

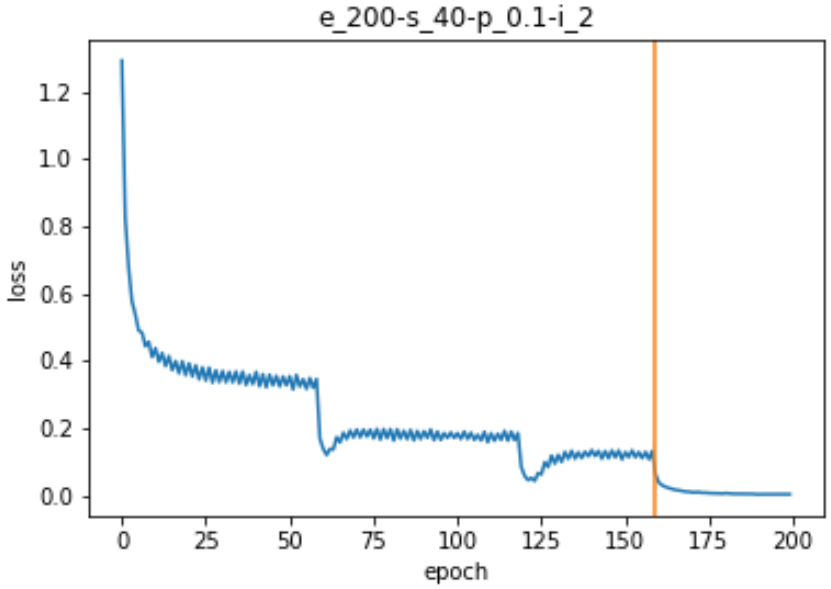

FIGURE 11 – Evolution de la fonction de coût au cours de l'entraînement

La figure 12 représente l'évolution du taux de réduction du réseau en fonction de l'entraînement du réseau. On y constate que la réduction ne commence qu'après un certain nombre d'époques, et que l'évolution du taux de réduction suit une allure exponentielle négative. L'une des particularités de cet algorithme est que la réduction s'effectue tout au long de l'entraînement, et pas seulement à la fin de celui-ci. Cela réduit de manière effective le temps d'entraînement, mais nécessite de réentraîner de zéro dans le cas de structures existantes.

Les premières époques où le taux de réduction n'évolue pas découlent du seuillage relatif effectué dans le réseau. Ce-dernier étant initialisé aléatoirement avec des poids équivalents, plusieurs époques d'apprentissage sont nécessaires avant de pouvoir distinguer les neurones les moins pertinents et les supprimer.

4.8 Publication de la librairie

Afin de rendre l'implémentation de l'algorithme de réduction utilisable dans d'autres projets, celle-ci a été publiée sous forme de librairie sur le serveur GitLab de l'Idiap. Ce choix permet de favoriser sa distribution et son utilisation au sein des autres projets de l'institut. A cette fin, le code source a été normalisé selon le standard PEP-8 et entièrement commenté.

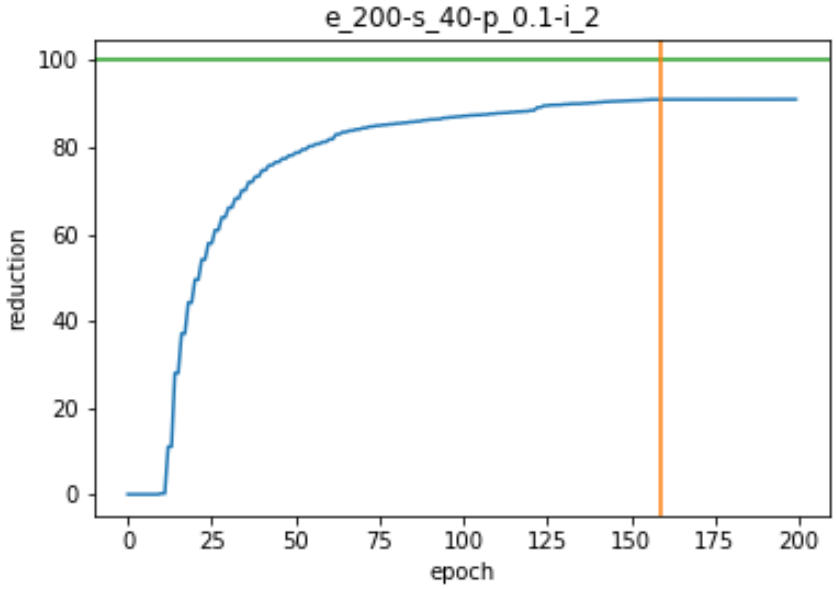

FIGURE 12 – Evolution du taux de réduction au cours de l'entraînement

La documentation a été générée avec l'outil de génération sphinx, afin de permettre une maintenabilité simple de la librairie. Cet outil permet en effet de régénérer automatiquement la documentation lors de modification de la source de la librairie. A titre informatif, la documentation de l'implémentation délivrée en fin de stage est reprise en annexe A.

Pour faciliter la prise en main de l'outil et l'apprentissage de la librairie, celle-ci est étoffée d'exemples et de tutoriaux. Ceux-ci permettent de découvrir progressivement l'utilisation de l'algorithme et son utilité dans différentes situations. Les exemples reprennent les codes sources des réseaux étudiés au cours du projet, afin d'illustrer l'utilisation de la librairie dans des cas d'application concrets.

5 Conclusion

Les réseaux de neurones artificiels sont une technologie utilisée dans des domaines de plus en plus variés actuellement. Cependant, l'évolution croissante de leur volume et de leur complexité limite leur déploiement sur des plateformes limitées en mémoire et en ressources de calcul. Les algorithmes de réduction de taille des réseaux de neurones répondent à ce problème en diminuant l'espace mémoire requis pour stocker les réseaux.

L'implémentation d'un algorithme de réduction réalisée au cours de ce stage de Master permet de réduire fortement la taille d'un système sans trop affecter sa précision. Ce comportement a été mesuré et validé sur des applications et des systèmes de complexité croissante, depuis un simple réseau convolutionnel LeNet jusqu'à un réseau résiduel de 25 couches. Certains réseaux ont pu être réduits jusqu'à 90% en taille tout en perdant moins de 1% d'efficacité.

L'outil développé a été publié sous forme de librairie python, afin de pouvoir facilement et rapidement l'intégrer à différents projets. Il contient des exemples et des explications en vue de faciliter sa prise en main et son utilisation. L'implémentation a été entièrement commentée et documentée afin de permettre une bonne maintenabilité à l'avenir.

Malgré l'étude de la dynamique de l'algorithme réalisée au cours de ce projet, le comportement exact de la réduction reste un domaine à investiguer. La connaissance précise de l'influence des différents paramètres sur les résultats de l'algorithme nécessite des tests plus poussés. Ceux-ci pourraient mettre en évidence les stratégies optimales à utiliser pour choisir ces paramètres, ainsi que les types de réseaux sur lesquels la réduction est la plus efficace. Il serait aussi intéressant de pousser l'optimisation des paramètres plus avant afin de connaître les niveaux de réduction et de précision qui peuvent être atteints avec cette implémentation.

Références

[Gas17] Xavier Gastaldi. Shake-shake regularization. *arXiv preprint arXiv :1705.07485*, 2017.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[KH09] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[LCB10] Yann LeCun, Corinna Cortes, and Christopher JC Burges. Mnist handwritten digit database. *AT&T Labs [Online]. Available : http ://yann. lecun. com/exdb/mnist*, 2, 2010.

[LJB+95] Yann LeCun, LD Jackel, Léon Bottou, Corinna Cortes, John S Denker, Harris Drucker, Isabelle Guyon, UA Muller, Eduard Sackinger, Patrice Simard, et al. Learning algorithms for classification : A comparison on handwritten digit recognition. *Neural networks : the statistical mechanics perspective*, 261 :276, 1995.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[ZAP16] Hao Zhou, Jose M Alvarez, and Fatih Porikli. Less is more : Towards compact cnns. In *European Conference on Computer Vision*, pages 662–677. Springer, 2016.

[ZK16] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv :1605.07146*, 2016.

A Documentation

Welcome to LIMutils's documentation !

LIMutils

LIMutils is an implementation of size reduction algorithms for DNN implemented with pyTorch.

It is based on the following article :

> **Zhou, Hao, Jose M. Alvarez, and Fatih Porikli. "Less is more : Towards compact cnns." European Conference on Computer Vision. Springer International Publishing, 2016.**

PDF available at :

`http://porikli.com/mysite/pdfs/porikli%202016%20-%20Less%20is%20more%20Towards%20compact%20CNNs.pdf`

LIMutils.absolute_reduce(weights, threshold, norm)

Applies the reducing algorithm based on fixed threshold, returns the indexes of outputs to be kept

Parameters : — **weights** (torch.autograd.Variable) - the input weights
— **threshold** (float) - the reduction threshold
— **norm** (torch.autograd.Variable) - the norm of the input filters

Return : — **keep_output** (torch.LongTensor) - the indexes of outputs that must be kept

LIMutils.relative_reduce(weights, threshold_ratio, norm)

Applies the reducing algorithm based on percentage of maximum norm

Parameters : — **weights** (torch.autograd.Variable) - the input weights

— **threshold_ratio** (float) - the percentage of the maximum norm for threshold
— **norm** (torch.autograd.Variable) - the norm of the input filters

Return : — **keep_output** (torch.LongTensor) - the indexes of outputs that must be kept

LIMutils.relative_reduce_1D(weights, threshold_ratio)

Applies the relative reduction algorithm to the weights of a layer with 1D input (ex : nn.Linear)

Parameters : — **weights** (torch.autograd.Variable) - the input weights of the layer
— **threshold_ratio** (float) - the percentage of the maximum norm for threshold

Return : — **keep_output** (torch.LongTensor) - the indexes of outputs that must be kept

LIMutils.relative_reduce_2D(weights, threshold_ratio)

Applies the relative reduction algorithm to the weights of a layer with 2D input (ex : nn.Conv2D)

Parameters : — **weights** (torch.autograd.Variable) - the input weights of the layer
— **threshold_ratio** (float) - the percentage of the maximum norm for threshold

Return : — **keep_output** (torch.LongTensor) - the indexes of outputs that must be kept

LIMutils.relative_reduce_layer(layer, threshold_ratio)

Applies the relative reduction algorithm to a nn.Module

Parameters : — **layer** (torch.nn.Module) - the module to be reduced
— **threshold_ratio** (float) - the percentage of the maximum norm for threshold

Return : — **keep_output** (torch.LongTensor) - the indexes of outputs that must be kept

LIMutils.shrink_layer(layer, keep_indexes_input=None, keep_indexes_output=None, unfold_factor=1)

Shrinks a nn.Module and keeps only the selected inputs and outputs, taking unfolding factor into account for inputs

Parameters : — **layer** (torch.nn.Module) - the module to be shrunk
— **keep_indexes_input** (torch.LongTensor) - the indexes of the inputs that must be kept
— **keep_indexes_output** (torch.LongTensor) - the indexes of the outputs that must be kept
— **unfold_factor** (int=1) - the unfolding factor between the outputs of the

preceding layer and the inputs of this layer (see below)

Return : — **shrunk_layer** (torch.nn.Module) - the shrunk module

Use of *unfold_factor* : The *unfold_factor* represents the ratio between the amount of input features of a layer and the amount of outputs of the preceding channel. For example, if a torch.nn.Linear (fully connected layer) directly follows a torch.nn.Conv2d, the sample must be resized from a 2D representation to a 1D representation. If the number of **output channels** of the convolution layer is **N**, and the image is of size HxW, then the amount of **input features** for the linear layer will be : **NxHxW**. The **unfolding factor** will then be **HxW** *for the linear layer.*

LIMutils.shrink_tensor(tensor, keep_indexes_input=None, keep_indexes_output=None, unfold_factor=1)

Shrinks a tensor and keeps only the selected inputs and outputs, taking unfolding factor into account for inputs

Parameters : — **tensor** (torch.Tensor, torch.autograd.Variable) - the tensor to be shrunk
— **keep_indexes_input** (torch.LongTensor) - the indexes of the inputs that must be kept
— **keep_indexes_output** (torch.LongTensor) - the indexes of the outputs that must be kept
— **unfold_factor** (int=1) - the unfolding factor between the outputs of the

preceding layer and the inputs of this layer (see below)

Return : — **shrunk_tensor** (type(*tensor*)) - the shrunk tensor

Use of *unfold_factor* : The *unfold_factor* represents the ratio between the amount of input features of a layer and the amount of outputs of the preceding channel. For example, if a torch.nn.Linear (fully connected layer) directly follows a torch.nn.Conv2d, the sample must be resized from a 2D representation to a 1D representation. If the number of **output channels** of the convolution layer is **N**, and the image is of size HxW, then the amount of **input features** for the linear layer will be : **NxHxW**. The **unfolding factor** will then be **HxW** *for the linear layer.*

LIMutils.shrink_vector(vector, keep_indexes, unfold_factor=1)

Shrinks a vector and keeps only the selected indexes, taking unfolding factor into account

Parameters : — **vector** (torch.Tensor, torch.autograd.Variable) - the vector to be shrunk
— **keep_indexes** (torch.LongTensor) - the indexes of the outputs that must be kept
— **unfold_factor** (int=1) - the unfolding factor between the outputs of the

preceding layer and the inputs of this layer (see below)

Return : — **shrunk_vector** (type(*vector*)) - the shrunk vector

Use of *unfold_factor* : The *unfold_factor* represents the ratio between the amount of input features of a layer and the amount of outputs of the preceding channel. For example, if a torch.nn.Linear (fully connected layer) directly follows a torch.nn.Conv2d, the sample must be resized from a 2D representation to a 1D representation. If the number of **output channels** of the convolution layer is **N**, and the image is of size HxW, then the amount of **input features** for the linear layer will be : **NxHxW**. The **unfolding factor** will then be **HxW** *for the linear layer.*