UNIVERSITÉ DE MONTRÉAL

RÉDUIRE LA PRÉCISION ET LE NOMBRE DES MULTIPLICATIONS NÉCESSAIRES À L'ENTRAÎNEMENT D'UN RÉSEAU DE NEURONES

MATTHIEU COURBARIAUX
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
AOÛT 2015

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

RÉDUIRE LA PRÉCISION ET LE NOMBRE DES MULTIPLICATIONS NÉCESSAIRES À L'ENTRAÎNEMENT D'UN RÉSEAU DE NEURONES

présenté par : COURBARIAUX Matthieu
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. BRAULT Jean-Jules, Ph. D., président
M. DAVID Jean-Pierre, Ph. D., membre et directeur de recherche
M. BENGIO Yoshua, Ph. D., membre et codirecteur de recherche
M. MEMISEVIC Roland, Ph. D., membre

DÉDICACE

"No matter how much knowledge and wisdom you acquire during your life, not one jot will be passed on to your children by genetic means."

Dawkins (2006)

REMERCIEMENTS

Je tiens tout d'abord à remercier mon directeur de recherche Jean-Pierre David de m'avoir si souvent fait part de ses inquiétudes à propos de l'avenir de ma maîtrise et aussi tant encouragé à publier le travail accompli. Je tiens aussi à remercier mon codirecteur de recherche Yoshua Bengio pour m'avoir introduit à l'apprentissage profond, avoir activement participé à l'écriture de nos articles, son expérience, sa bienveillance, et ses encouragements. Je les remercie tous deux pour les conversations que nous avons eu ensemble, leurs conseils et la grande autonomie dont j'ai bénéficié.

Je remercie le professeur Roland Memisevic pour son excellent cours d'apprentissage automatique pour la vision, ses conseils, son enthousiasme et pour avoir accepté d'être membre du jury d'examen de ma maîtrise. Je remercie le professeur Jean-Jules Brault d'avoir accepté de diriger le jury d'examen de ma maîtrise.

Je remercie l'École Polytechnique de Montréal et son département de génie électrique de m'avoir admis en tant qu'étudiant en double-diplôme. Je remercie le CRSNG, les chaires de recherche du Canada, Calcul Canada et ICRA de leurs fonds.

Je remercie les étudiants et stagiaires passés et présents du GRM avec qui j'ai pu sympathiser, en particulier : Michel Gémieux, Clément Michaud, Marc-André Daigneault, Gontran Sion et Adrien Labarnet. Je remercie aussi les étudiants du MILA que j'ai pu rencontrer. Enfin, de manière plus générale, je remercie mes amis et ma famille pour leur soutien et leur affection.

RÉSUMÉ

Les Réseaux de Neurones (RdNs) sont à l'état de l'art pour un grand nombre de tâches, les meilleurs résultats étant obtenus avec de grands ensembles de données et de grands modèles. La vitesse de calcul des cartes graphiques est en grande partie à l'origine de ces progrès. À l'avenir, l'accélération des RdNs pendant les phases d'entrainement et de test permettra probablement une performance accrue ainsi que des applications grand public plus efficaces énergétiquement. En conséquence, la recherche en systèmes numériques dédiés aux RdNs est d'actualité. Les systèmes numériques sont principalement faits de mémoires et d'opérateurs arithmétiques. Les multiplieurs sont de loin les opérateurs arithmétiques les plus coûteux en termes de transistors d'un système numérique dédié aux RdNs.

Dans notre premier article, nous entraînons un ensemble de RdNs à l'état de l'art (les réseaux Maxout) sur trois ensembles de données de référence : MNIST, CIFAR-10 et SVHN. Ils sont entraînés avec trois formats distincts : virgule flottante, virgule fixe et virgule fixe dynamique. Pour chacun de ces ensembles de données et pour chacun de ces formats, nous évaluons l'impact de la précision des multiplications sur l'erreur finale après l'entrainement. Nous trouvons qu'une précision très faible est suffisante non seulement pour tester des RdNs, mais aussi pour les entraîner. Par exemple, il est possible d'entraîner des réseaux Maxout avec des multiplications 10 bits.

Des poids binaires, c'est à dire des poids qui sont contraints à seulement deux valeurs possibles (e.g. -1 ou 1), permettraient de beaucoup réduire le nombre de multiplications nécessaires lors de l'entraînement d'un RdN. Dans notre deuxième article, nous introduisons BinaryConnect, une méthode qui consiste à entraîner un RdN avec des poids binaires durant les propagations en avant et en arrière, tout en conservant la précision des poids stockés dans lesquels les gradients sont accumulés. Comme les autres variantes de Dropout, nous montrons que BinaryConnect agit comme régulariseur et nous obtenons des résultats proches de l'état de l'art avec BinaryConnect sur le MNIST invariant aux permutations.

ABSTRACT

Deep Neural Networks (DNNs) have achieved state-of-the-art results in a wide range of tasks, with the best results obtained with large training sets and large models. In the past, GPUs enabled these breakthroughs because of their greater computational speed. In the future, faster computation at both training and test time is likely to be crucial for further progress and for consumer applications on low-power devices. As a result, there is much interest in research and development of dedicated hardware for Deep Learning (DL). Computer hardware is mainly made out of memories and arithmetic operators. Multipliers are by far the most space and power-hungry arithmetic operators of the digital implementation of neural networks.

In our first article, we train a set of state-of-the-art neural networks (Maxout networks) on three benchmark datasets: MNIST, CIFAR-10 and SVHN. They are trained with three distinct formats: floating point, fixed point and dynamic fixed point. For each of those datasets and for each of those formats, we assess the impact of the precision of the multiplications on the final error after training. We find that very low precision is sufficient not just for running trained networks but also for training them. For example, it is possible to train Maxout networks with 10 bits multiplications.

Binary weights, i.e., weights which are constrained to only two possible values (e.g. -1 or 1), would greatly reduce the number of multiplications required to train a DL. In our second article, we introduce BinaryConnect, a method which consists in training a DNN with binary weights during the forward and backward propagations, while retaining precision of the stored weights in which gradients are accumulated. Like other dropout schemes, we show that BinaryConnect acts as regularizer and we obtain near state-of-the-art results with BinaryConnect on the permutation-invariant MNIST.

TABLE DES MATIÈRES

DÉDICACE iii

REMERCIEMENTS iv

RÉSUMÉ v

ABSTRACT vi

TABLE DES MATIÈRES vii

LISTE DES TABLEAUX ix

LISTE DES FIGURES x

LISTE DES SIGLES ET ABRÉVIATIONS xi

LISTE DES ANNEXES xiii

CHAPITRE 1 INTRODUCTION 1
1.1 Réseaux de neurones 1
1.2 Processeurs dédiés à l'apprentissage 1
1.3 Réduire la précision et le nombre des multiplications 2
1.4 Plan du mémoire 2

CHAPITRE 2 ÉTAT DES CONNAISSANCES 3
2.1 Apprentissage profond 3
2.1.1 Réseau de neurones 3
2.1.2 Ensemble de données 5
2.1.3 Descente du gradient 6
2.1.4 Généralisation 7
2.1.5 Réseaux à convolution 9
2.1.6 Points clés 9
2.2 Systèmes numériques 10
2.2.1 Des transistors aux fonctions logiques 10
2.2.2 Niveau d'abstraction des transferts de registres 12
2.2.3 Processeurs 12

2.2.4 Unités arithmétiques et logiques 13
2.2.5 Mémoires 15
2.2.6 Unités de Contrôle 16
2.2.7 Entraînement distribué 16
2.2.8 Points clés 16

CHAPITRE 3 SYNTHÈSE DE L'ENSEMBLE DU TRAVAIL 18
3.1 Entraînement de réseaux de neurones avec des multiplications moins précises 18
3.2 BinaryConnect : entraînement de réseaux de neurones avec des poids binaires pendant les propagations 19

CHAPITRE 4 DISCUSSION GÉNÉRALE 20
4.1 Entraînement de réseaux de neurones avec des multiplications moins précises 20
4.2 BinaryConnect : entraînement de réseaux de neurones avec des poids binaires pendant les propagations 20

CHAPITRE 5 CONCLUSION 22
5.1 Synthèse 22
5.2 Accélération résultante 22
5.3 Travaux futurs 22

RÉFÉRENCES 24

ANNEXES A : ARTICLE 1 : LOW PRECISION STORAGE FOR DEEP LEARNING 29

ANNEXES B : ARTICLE 2 : BINARYCONNECT : TRAINING DEEP NEURAL NETWORKS WITH BINARY WEIGHTS DURING PROPAGATIONS 52

LISTE DES TABLEAUX

Tableau 2.1 Coût en ALMs d'un multiplieur-accumulateur sur un FPGA Stratix V d'Altera en fonction des précisions du multiplieur et de l'accumulateur. Sur les FPGAs modernes, les multiplications peuvent aussi être implémentées avec des blocs de multiplications (DSPs). Un DSP correspond à 1 multiplieur 27 bits, 2 multiplieurs 18 bits ou 3 multiplieurs 9 bits. 12

Tableau 3.1 Estimations des puissances de calcul de processeurs utilisant des multiplieurs 10 bits ou/et BinaryConnect. Diminuer le coût des multiplieurs permet d'augmenter leur nombre et donc d'obtenir une accélération. Il est possible d'avoir $(32/10)^2 = 10.24$ multiplieurs 10 bits pour le prix d'un multiplieur 32 bits. BinaryConnect divise par 3 le nombre des multiplications nécessaires lors de l'entraînement d'un RdN. La puissance de calcul est le nombre de connexions synaptiques qu'il est possible de simuler par seconde (TCo/s). 18

LISTE DES FIGURES

Figure 2.1 Neurone biologique. Copie autorisée d'une figure réalisée par Nicolas Rougier en 2007 et disponible à l'adresse `https://commons.wikimedia.org/wiki/File:Neuron-figure-fr.svg` 4

Figure 2.2 Neurone artificiel. x1, x2 et x3 sont les entrées, w11, w12 et w13 les poids, b1 le biais, f la non-linéarité, et y1 la sortie du neurone. Le neurone effectue une somme de ses entrées pondérée par ses poids. . . 5

Figure 2.3 Perceptron multi-couches. 6

Figure 2.4 Quelques images du MNIST. 7

Figure 2.5 Descente du gradient. E est l'erreur, w un des paramètres et dE/dw le gradient de E par rapport à w. 8

Figure 2.6 Pattern de connectivité local et partage de poids d'un réseau à convolution. Les connexions de même couleur partagent le même poids. . . 8

Figure 2.7 Couche de neurones à convolution avec sous-échantillonnage. 9

Figure 2.8 Porte logique NAND faîte à partir de 4 transistors CMOS. Les deux entrées sont A et B, et la sortie est Out. Copie autorisée d'une figure réalisée par Justin Force en 2007 et disponible à l'adresse `https://commons.wikimedia.org/wiki/File:CMOS_NAND.svg` 10

Figure 2.9 Architecture d'un processeur moderne vectorisé. 11

Figure 2.10 Multiplication-accumulation (MAC). A et B sont les entrées, et C la sortie. 12

Figure 2.11 Hiérarchie de la mémoire du GPU Titan X. 13

Figure 2.12 Unité de contrôle simplifiée. 14

LISTE DES SIGLES ET ABRÉVIATIONS

DL Deep Learning

DNN Deep Neural Network

BN Batch Normalization

CNN Convolutional Neural Network

SVM Support Vector Machine

BP BackPropagation

EP Expectation Propagation

EBP Expectation BackPropagation

PI Permutation Invariant

ZCA Zero-phase Component Analysis

MLP Perceptron multi-couches

ReLU Unité linéaire rectifiée

SGD Descente du gradient stochastique

RdN Réseau de Neurones

IA Intelligence Artificielle

MNIST Mixed National Institute of Standards and Technology

SVHN Street View House Numbers

CIFAR Canadian Institute For Advanced Research

ICRA Institut Canadien de Recherches Avancées

NSERC Natural Sciences and Engineering Research Council of Canada

CRSNG Conseil de Recherches en Sciences Naturelles et en Génie du Canada

ICLR International Conference on Learning Representations

NIPS Neural Information Processing Systems

GRM Groupe de Recherche en Microélectronique

MILA Institut des algorithmes d'apprentissage de Montréal

GPU Processeur graphique

CPU Unité centrale de traitement

FPGA Réseau pré-diffusé programmable

RAM Mémoire à accès aléatoire

MAC multiplieur-accumulateur

ALM Module logique adaptable

DSP bloc de multiplication

CU Unité de contrôle

NAND Porte logique non-et

RTL Niveau d'abstraction des transferts de registres

HDL Langage de description de matériel

LISTE DES ANNEXES

Annexe A ARTICLE 1 : LOW PRECISION STORAGE FOR DEEP LEARNING 29

Annexe B ARTICLE 2 : BINARYCONNECT : TRAINING DEEP NEURAL NETWORKS WITH BINARY WEIGHTS DURING PROPAGATIONS 52

CHAPITRE 1 INTRODUCTION

1.1 Réseaux de neurones

Un réseau de neurones (RdN) artificiel est un modèle mathématique inspiré des systèmes nerveux des animaux, en particulier ceux des grenouilles (Lettvin et al., 1959) et des chats (Eckhorn et al., 1988; Gray and Singer, 1989; Gray et al., 1989). Lorsqu'un un signal, par exemple une image, est présentée à un RdN, sa couche d'entrée 'analyse' le signal et transmet les résultats à la couche suivante. Ce processus est répété pour chacune des couches du RdN, c'est-à-dire entre 10 et 30 fois pour un RdN moderne (Szegedy et al., 2014; Simonyan and Zisserman, 2014).

Ces cinq dernières années, les RdNs ont significativement repoussé les limites de l'Intelligence Artificielle (IA), tout particulièrement en reconnaissance de la parole (Hinton et al., 2012; Sainath et al., 2013), en traduction par ordinateur statistique (Devlin et al., 2014; Sutskever et al., 2014; Bahdanau et al., 2015), en reconnaissance d'images (Krizhevsky et al., 2012; Szegedy et al., 2014), en contrôle automatique de jeux Atari (Mnih et al., 2015) et même en art abstrait (expressionniste ?) automatique (Mordvintsev et al., 2015). Les RdNs sont ce que nous avons de plus proche d'une IA forte (i.e. égalant l'humain) à ce jour. Une IA forte serait un tournant dans l'histoire des hommes.

1.2 Processeurs dédiés à l'apprentissage

La capacité d'un RdN dépend en grande partie de sa taille, c'est-à-dire son nombre de connexions. Le nombre de connexions est limité par le temps qui est nécessaire pour les ajuster : le temps d'apprentissage. Le temps d'apprentissage dépend essentiellement de la puissance de calcul disponible. Un des facteurs à l'origine de progrès majeurs en apprentissage de RdN est l'utilisation de cartes graphiques (GPUs), avec des accélérations de l'ordre de 10× à 30× par rapport aux unités centrales de traitement (CPUs), en commençant avec Raina et al. (2009), et des améliorations similaires avec l'apprentissage distribué sur plusieurs CPUs (Bengio et al., 2003; Dean et al., 2012).

La puissance de calcul reste néanmoins un des facteurs limitant la capacité des RdNs. L'apprentissage d'un RdN moderne peut ainsi prendre plusieurs jours de calculs sur plusieurs GPUs (Szegedy et al., 2014; Simonyan and Zisserman, 2014). Ceci, plus la volonté de mettre des RdNs sur des systèmes de faible-consommation (ce qui n'est pas le cas des GPUs), augmente grandement l'intérêt dans la recherche et le développement de processeurs dédiés aux

RdNs (Kim et al., 2009; Farabet et al., 2011; Pham et al., 2012; Chen et al., 2014a,b). L'objectif de ce mémoire est de trouver un moyen d'accélérer les processeurs dédiés aux RdNs.

1.3 Réduire la précision et le nombre des multiplications

Une connexion synaptique d'un RdN artificiel est modélisée comme une multiplication par un poids (synaptique). Les multiplieurs sont sans surprise les opérateurs arithmétiques les plus coûteux en transistors d'un processeur numérique dédié aux RdNs. Réduire la précision des multiplieurs réduirait leur coût, ce qui permettrait d'augmenter leur nombre et donc d'obtenir une accélération. C'est le sujet du premier article de ce mémoire (Annexe A). Dans cet article, nous démontrons qu'il est possible d'entraîner un RdN avec des multiplications 10 bits, ce qui est une nette amélioration comparé aux 32 bits des GPUs et aux travaux précédents.

Alternativement, il est possible de proposer un nouveau modèle de connexion synaptique nécessitant moins (ou pas) de multiplications. Dans notre deuxième article (Annexe B), nous proposons une nouvelle méthode, BinaryConnect, qui consiste à contraindre les poids des connexions à une valeur binaire (e.g. -1 ou 1) pendant les propagations, ce qui divise par trois le nombre de multiplications nécessaires à l'entraînement d'un RdN.

1.4 Plan du mémoire

Ce mémoire se divise en cinq chapitres (et deux articles en annexe) :

1. cette introduction (chapitre 1),
2. un état des connaissances en apprentissage profond et en processeurs numériques, incluant les processeurs dédiés à l'apprentissage profond (chapitre 2),
3. une synthèse de l'ensemble du travail réalisé (chapitre 3),
4. une discussion des résultats obtenus dans nos deux articles (chapitre 4),
5. et une ouverture sur les prochains travaux à réaliser (chapitre 5).

CHAPITRE 2 ÉTAT DES CONNAISSANCES

L'objectif de ce mémoire est d'accélérer les systèmes numériques dédiés à l'apprentissage profond. Ce mémoire est donc à l'intersection entre deux domaines de recherches : l'apprentissage profond et les systèmes numériques. Ce chapitre a pour but de faire l'état des connaissances dans ces deux domaines et leur intersection. La section 2.1 présente l'apprentissage profond et la section 2.2 présente les systèmes numériques, incluant ceux dédiés à l'apprentissage profond.

2.1 Apprentissage profond

Cette section a pour objectif de faire l'état des connaissances en apprentissage profond.

2.1.1 Réseau de neurones

Entrées : Une fonction d'erreur E, des paramètres initiaux θ_0, un nombre de pas d'optimisation T et un taux d'apprentissage η.
Sorties : Des paramètres optimisés θ_n.

$t \leftarrow 0$
tant que $t < T$ **faire**
  Calculer $\frac{\partial E}{\partial \theta_t}$ avec la règle de la dérivation en chaîne
  $\theta_{t+1} \leftarrow \theta_t - \eta \times \frac{\partial E}{\partial \theta_t}$
  $t \leftarrow t + 1$
fin tant que

Algorithme 1 Descente du gradient.

Les réseaux de neurones (RdNs) artificiels sont une famille de fonctions paramétrées qui est grandement inspirée des RdNs biologiques. Un neurone biologique fait la somme des signaux électriques qui arrivent à ses dendrites, et si cette somme dépasse un seuil, envoie des signaux par son axone. La figure 2.1 est un schéma de neurone biologique. De manière similaire, un neurone artificiel effectue une somme de ses entrées pondérée par ses poids puis applique une non-linéarité à la somme résultante, ce qu'illustre la figure 2.2.

Un perceptron multi-couches (MLP) est un RdN constitué de plusieurs couches de neurones, chacune des couches étant entièrement connectée à la suivante, ce qu'illustre la figure 2.3. Il est possible de modéliser une couche de MLP comme le produit entre un vecteur d'entrées

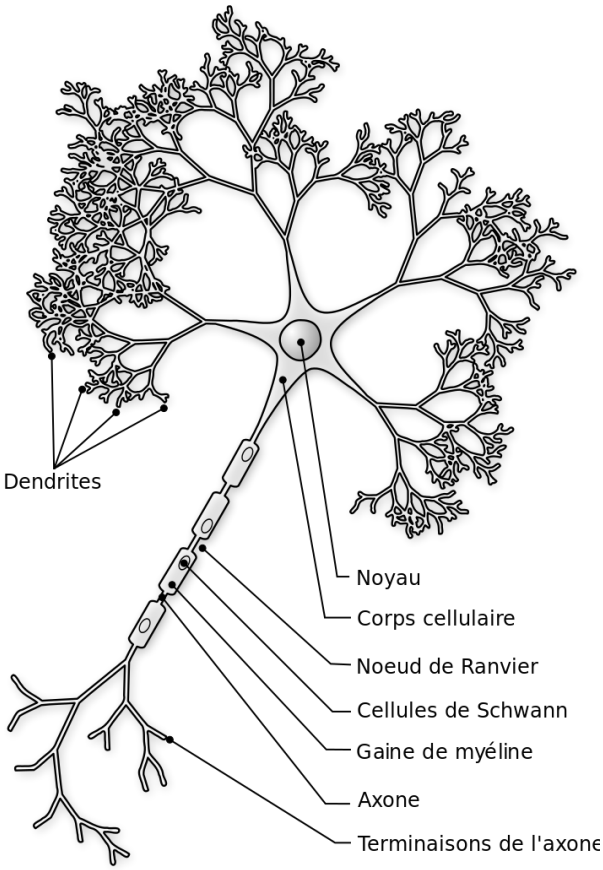

Figure 2.1 Neurone biologique. Copie autorisée d'une figure réalisée par Nicolas Rougier en 2007 et disponible à l'adresse `https://commons.wikimedia.org/wiki/File:Neuron-figure-fr.svg` .

et une matrice de poids, suivi d'une non-linéarité :

$$y_j = f(\sum_{i=1}^{n} W_{ji} x_i + b_j) \tag{2.1}$$

$$\leftrightarrow y = f(W.x + b) \tag{2.2}$$

où y est le vecteur de sortie de la couche de neurones, f la non-linéarité, n la taille du vecteur d'entrées, W la matrice des poids, b le vecteur des biais et x le vecteur d'entrées. Une non-linéarité actuellement très populaire est l'unité linéaire rectifiée (ReLU) (Nair and Hinton, 2010; Glorot et al., 2011; Krizhevsky et al., 2012) :

$$f(z) = \max(0, z) \tag{2.3}$$

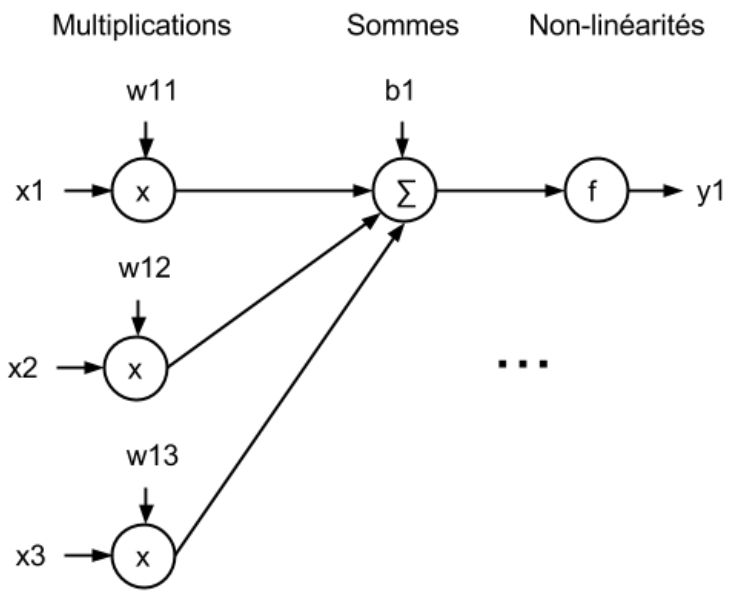

Figure 2.2 Neurone artificiel. x1, x2 et x3 sont les entrées, w11, w12 et w13 les poids, b1 le biais, f la non-linéarité, et y1 la sortie du neurone. Le neurone effectue une somme de ses entrées pondérée par ses poids.

Les MLPs ont la propriété intéressante d'être des approximateurs de fonction universels (Hornik, 1991), ce qui veut dire qu'un MLP dont les paramètres sont suffisamment nombreux et bien ajustés peut approximer n'importe quelle fonction dépendant uniquement de ses entrées. La question est : comment ajuster les paramètres d'un MLP ?

2.1.2 Ensemble de données

Dans la plupart des cas, nous ne partons pas de rien. Nous disposons d'un ensemble de données, c'est-à-dire un ensemble d'échantillons de la fonction que nous cherchons à approximer. Par exemple, le MNIST est un ensemble de données de classification d'images (LeCun et al., 1998). Classifier une image, c'est y associer la catégorie qui lui correspond. Le MNIST regroupe pas moins de 70000 images 28×28 en nuance de gris représentant des chiffres écrits à la main entre 0 et 9. À chacune de ces images est associé le chiffre correspondant. La figure 2.4 contient quelques images du MNIST. La performance d'un RdN dépend beaucoup de la taille et de la qualité de l'ensemble de données.

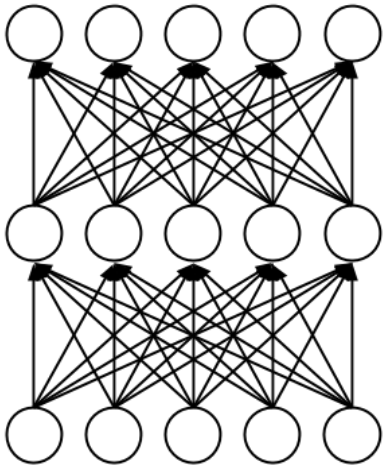

Figure 2.3 Perceptron multi-couches.

2.1.3 Descente du gradient

Comment utiliser un ensemble de données pour ajuster les paramètres d'un RdN ? La réponse est dans le titre de la section : en faisant une descente du gradient. La descente du gradient est une méthode d'optimisation numérique très populaire en apprentissage profond. La première étape de la descente du gradient consiste à définir une fonction d'erreur dérivable qui dépend des exemples de l'ensemble de données et des paramètres du RdN :

$$E((x, y), \theta) = \sum_{i=1}^{n} \|RdN(x_i, \theta) - y_i\|_2^2 \tag{2.4}$$

où E est l'erreur quadratique, (x, y) l'ensemble de données, n le nombre d'exemples et θ les paramètres du RdN. L'objectif de l'optimisation numérique est de trouver les paramètres θ^* pour lesquels l'erreur est minimale :

$$\theta^* = \inf_{\theta} E((x, y), \theta) \tag{2.5}$$

La seconde étape de la descente du gradient consiste à mettre à jour les paramètres dans la direction de leur gradient, ce qu'illustrent la figure 2.5 et l'algorithme 1.

En pratique, on préfère à la descente du gradient une de ses variantes appelée la descente du gradient stochastique (SGD). Contrairement à la fonction d'erreur de la descente du gradient, celle de la SGD ne dépend que d'une sous-partie de l'ensemble de données (appelée mini-batch) et non de l'ensemble de données au complet. La SGD a pour avantages de

Figure 2.4 Quelques images du MNIST.

converger beaucoup plus rapidement (Wilson and Martinez, 2003) et d'être biologiquement plus plausible.

2.1.4 Généralisation

Nous savons maintenant comment entraîner un RdN sur un ensemble de données. La question est : notre RdN est il capable de généraliser ? Quelle est sa performance sur des exemples ne faisant pas partie de son ensemble de données ? Une sous-partie de l'ensemble de données, appelée ensemble de test, est réservée à l'évaluation de la performance du RdN. Il est absolument interdit d'utiliser l'ensemble de test pour ajuster des paramètres (Simonite, 2015).

L'entraînement d'un MLP par SGD possède de nombreux paramètres en dehors des poids du MLP, par exemple le nombre de neurones et de couches du MLP, le nombre de pas et le taux d'apprentissage de la SGD. Ces paramètres sont appelés des hyper-paramètres. Une seconde sous-partie de l'ensemble de données, appelée ensemble de validation, est réservée à l'ajustement ce ces hyper-paramètres, le but étant de sélectionner le RdN qui généralise le mieux. Un ensemble de données est donc divisé en trois sous-parties :

1. l'ensemble d'entraînement, qui sert à ajuster les poids du RdN par descente du gradient,

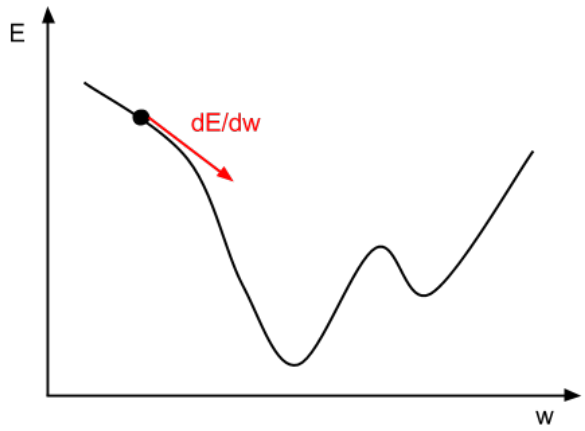

Figure 2.5 Descente du gradient. E est l'erreur, w un des paramètres et dE/dw le gradient de E par rapport à w.

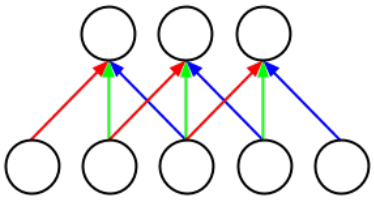

Figure 2.6 Pattern de connectivité local et partage de poids d'un réseau à convolution. Les connexions de même couleur partagent le même poids.

2. l'ensemble de validation, qui sert à ajuster les hyper-paramètres du RdN et de la SGD, le but étant de maximiser le pouvoir de généralisation du RdN,
3. et l'ensemble de test, qui sert à évaluer la performance du RdN (avec les hyper-paramètres sélectionnés sur l'ensemble de validation).

Par exemple, les ensembles de validation et de test du MNIST comptent 10000 images chacun et son ensemble d'entrainement les 50000 images restantes.

Des méthodes dites de régularisation ont pour but d'augmenter la pouvoir de généralisation d'un RdN. Par exemple, Dropout (Srivastava, 2013; Srivastava et al., 2014) est une méthode de régularisation populaire et efficace. Dropout consiste à ignorer aléatoirement une moitié des neurones du RdN durant la phase de calcul des gradients des paramètres (cf. algorithme 1). Dropout fait en sorte que le RdN apprenne des paramètres plus robustes au bruit.

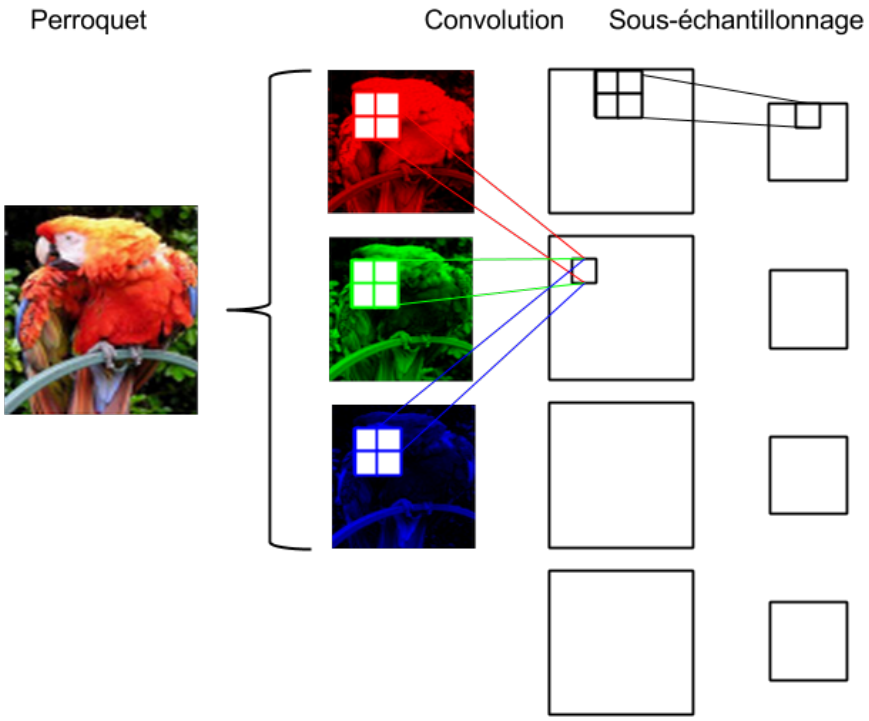

Figure 2.7 Couche de neurones à convolution avec sous-échantillonnage.

2.1.5 Réseaux à convolution

Les réseaux à convolution (LeCun et al., 1998) sont des RdNs très populaires en vision par ordinateur. Ils sont actuellement à l'état de l'art en reconnaissance d'objet (Krizhevsky et al., 2012; Szegedy et al., 2014). Un réseau à convolution est une variante du MLP qui se distingue par son pattern de connectivité local et son partage de poids, ce qu'illustre la figure 2.6. Il est possible de modéliser une couche à convolution 2D comme les convolutions d'un ensemble de matrices d'entrées par un ensemble de filtres, ce qu'illustre la figure 2.7.

2.1.6 Points clés

- Un MLP est un RdN constitué de plusieurs couches de neurones, chacune de ces couches étant entièrement connectée à la suivante.
- Il est possible de modéliser une couche de MLP comme le produit entre un vecteur d'entrées et une matrice de poids, suivi d'une non-linéarité.
- Un MLP est un approximateur de fonction universel.

– Un ensemble de données est un ensemble d'échantillons de la fonction que nous cherchons à approximer.
– La SGD est une méthode d'optimisation numérique qui permet d'ajuster les poids d'un MLP en utilisant un ensemble de données.
– L'ensemble de données est typiquement divisé en trois : les ensembles d'entraînement, de validation et de test, le but étant de maximiser et d'évaluer le pouvoir de généralisation du MLP.
– Des méthodes de régularisation permettent d'augmenter le pouvoir de généralisation d'un MLP.
– Les réseaux à convolution sont une variante populaire des MLPs où les produits vecteurs-matrices sont remplacés par des convolutions.

2.2 Systèmes numériques

Cette section a pour objectif de faire l'état des connaissances en conception de systèmes numériques, incluant ceux dédiés à l'apprentissage profond.

.

2.2.1 Des transistors aux fonctions logiques

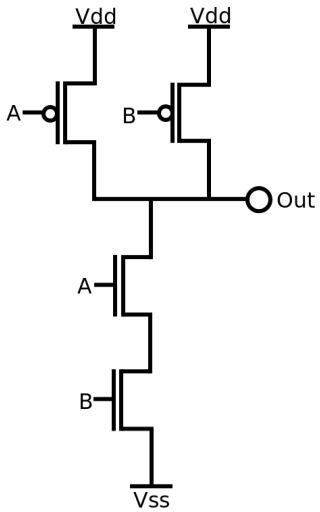

Figure 2.8 Porte logique NAND faîte à partir de 4 transistors CMOS. Les deux entrées sont A et B, et la sortie est Out. Copie autorisée d'une figure réalisée par Justin Force en 2007 et disponible à l'adresse `https://commons.wikimedia.org/wiki/File:CMOS_NAND.svg` .

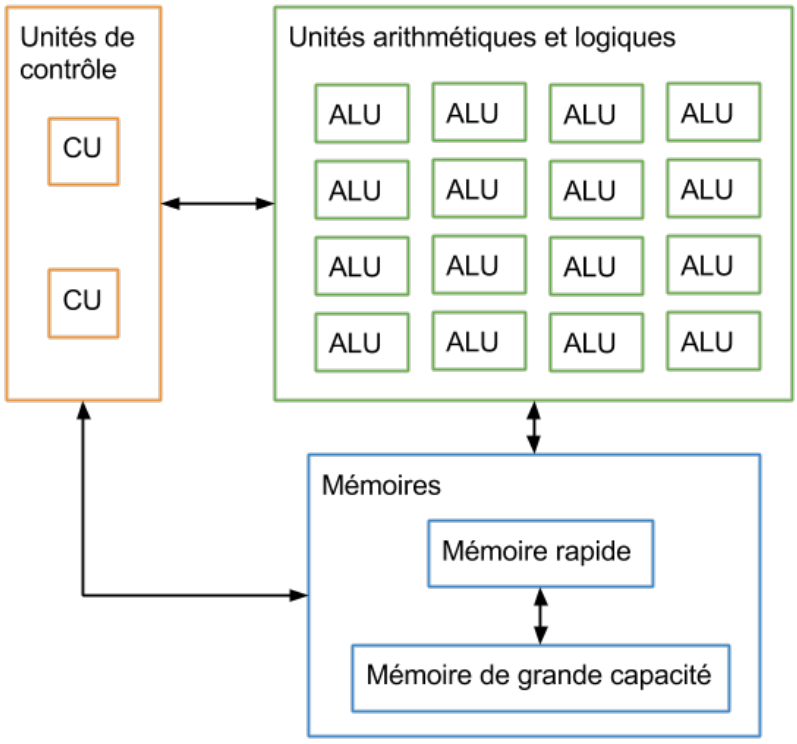

Figure 2.9 Architecture d'un processeur moderne vectorisé.

Les systèmes numériques modernes sont faits à partir de transistors. Les transistors sont des semi-conducteurs qui se comportent comme des interrupteurs. Il est possible de construire une porte logique non-et (NAND) à partir de 4 transistors, ce qu'illustre la figure 2.8. De manière plus générale, il est possible de construire n'importe quelle fonction logique (comme un RdN) à partir de portes NAND et donc de transistors. Les seules véritables limites sont le coût des transistors et la consommation énergétique :

– Le coût des transistors dépend essentiellement de la finesse de gravure. Celle-ci diminue chaque année (Moore, 1965). Aujourd'hui, la finesse de gravure est aux alentours de 28nm et un système numérique moderne peut compter plus de 8 milliards de transistors (Smith, 2015).
– La consommation énergétique dépend du type, du nombre et de l'activité des transistors, et augmente linéairement avec la fréquence d'horloge du système (Weste and Harris, 2010, chapitre 5).

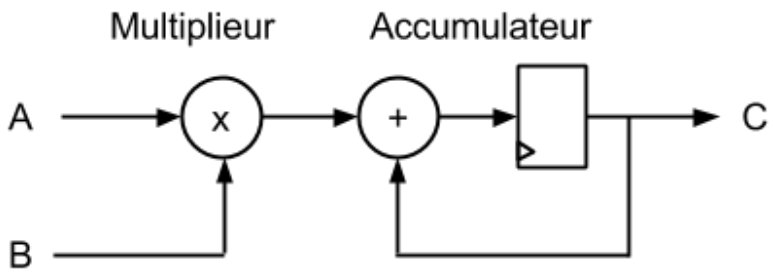

Figure 2.10 Multiplication-accumulation (MAC). A et B sont les entrées, et C la sortie.

Tableau 2.1 Coût en ALMs d'un multiplieur-accumulateur sur un FPGA Stratix V d'Altera en fonction des précisions du multiplieur et de l'accumulateur. Sur les FPGAs modernes, les multiplications peuvent aussi être implémentées avec des blocs de multiplications (DSPs). Un DSP correspond à 1 multiplieur 27 bits, 2 multiplieurs 18 bits ou 3 multiplieurs 9 bits.

Multiplieur (bits)	Accumulateur (bits)	Modules logiques adaptables (ALMs)
32	32	504
16	32	138
16	16	128

2.2.2 Niveau d'abstraction des transferts de registres

À ce jour, la conception de système numériques ne se fait plus au niveau d'abstraction des transistors et des portes NAND, mais à celui des transferts de registres (RTL). Les circuits logiques de base sont alors des éléments à mémoire comme par exemple des mémoires à accès aléatoire (RAM) ou bien des opérateurs logiques et arithmétiques comme par exemple des multiplieurs. Le rôle du concepteur consiste à décrire les connexions entre ces circuits de base en utilisant un langage de description matérielle (HDL), le but étant de réaliser (dans notre cas) un processeur dédié à l'entraînement rapide de RdNs.

2.2.3 Processeurs

Un processeur de données est une véritable usine de production et de traitement de données. Il est divisé en quatre sous-parties :

1. des unités arithmétiques et logiques (ALUs) qui traitent les données (section 2.2.4),
2. un ensemble de mémoires qui stockent les données (section 2.2.5),

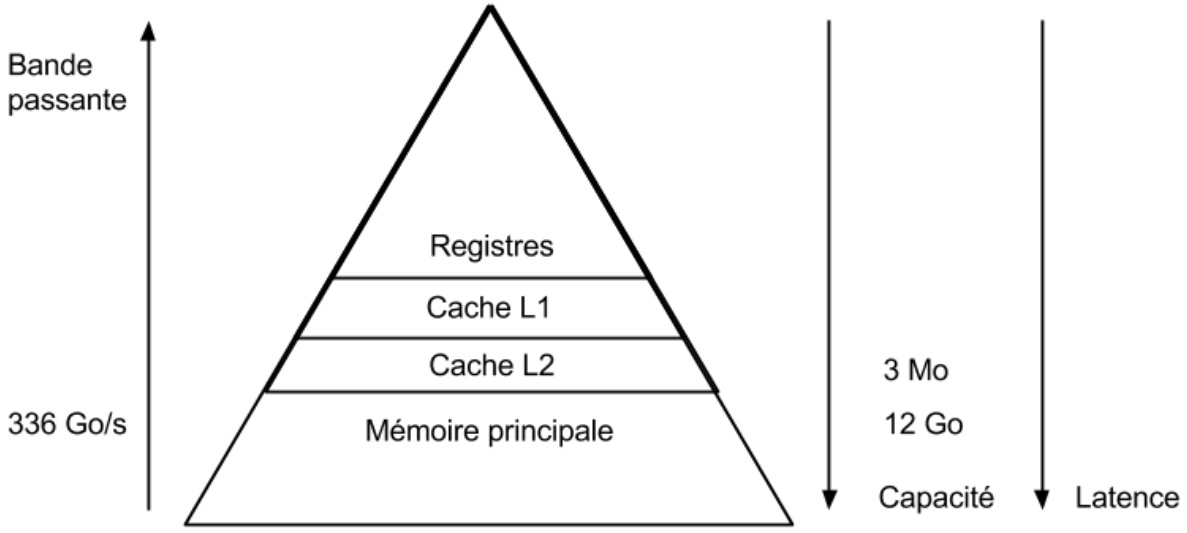

Figure 2.11 Hiérarchie de la mémoire du GPU Titan X.

3. des unités de contrôle (CU) qui donnent des ordres aux ALUs et aux mémoires (section 2.2.6),
4. et un réseau d'interconnexions.

La figure 2.9 illustre l'architecture d'un processeur moderne vectorisé.

2.2.4 Unités arithmétiques et logiques

Typiquement, une ALU peut réaliser des opérations arithmétiques (e.g. additions, multiplications), des opérations logiques (e.g. et, ou) et des comparaisons. Entraîner un RdN c'est principalement effectuer des produits matriciels. Par exemple, lors de l'entraînement d'un réseau à convolution sur un processeur graphique (GPU), plus de 95% du temps d'exécution est consacré aux produits matriciels (Jia, 2014, chapitre 6). Un produit matriciel ne nécessite qu'un seul opérateur arithmétique : le multiplieur-accumulateur (MAC). La figure 2.10 illustre l'architecture d'un MAC (qui, sans surprise, est assez proche de celle d'un neurone artificiel).

Les processeurs modernes, tels que les unités centrales de traitement (CPU) et les processeurs graphiques (GPU) ont une architecture parallèle. Cela veut dire qu'ils disposent de plusieurs ALUs qui travaillent en parallèle. Par exemple, le GPU Titan X possède pas moins de 3072 ALUs dont la fréquence d'horloge est 1GHz (Smith, 2015). Chacune de ces ALUs peut ef-

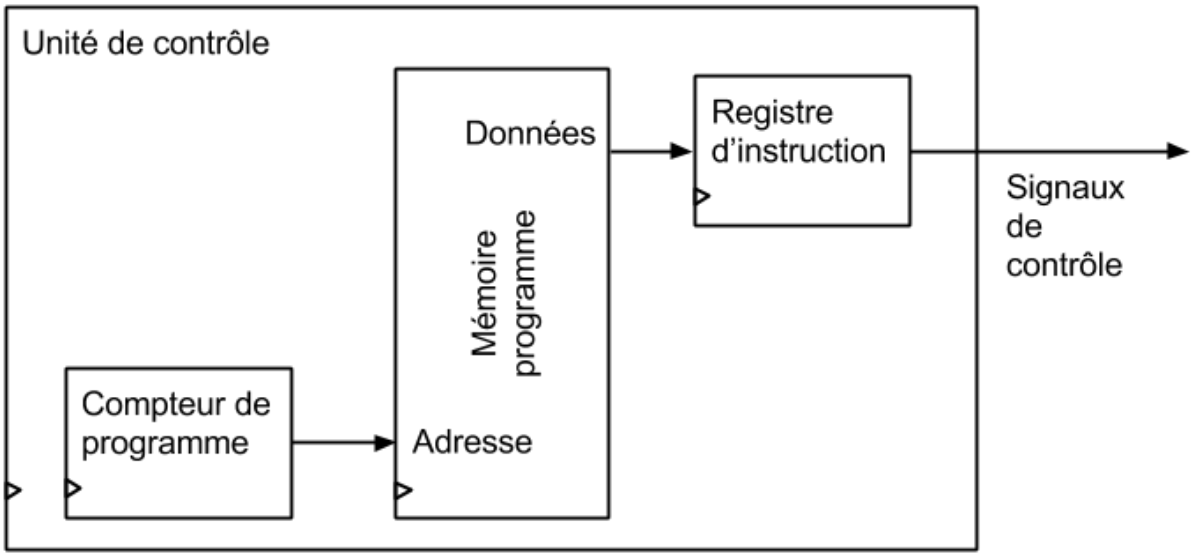

Figure 2.12 Unité de contrôle simplifiée.

fectuer une MAC par coup d'horloge, soit 3072 × 1GHz = 3 TMAC/s (i.e. 6 TFLOP/s). Le nombre de MACs qu'il est possible de mettre dans un processeur dépend avant tout de la précision des multiplieurs. Le coût en transistors d'un multiplieur est de $O(n^2)$ quand n est petit (David et al., 2007), n étant le nombre de bits de précision (32 bits dans le cas des GPUs). Le coût des accumulateurs ($O(n)$) est négligeable devant celui des multiplieurs ($O(n^2)$), ce qu'illustre le tableau 2.1.

Entraîner un RdN avec des multiplications faible précision permettrait d'augmenter le nombre des multiplieurs d'un processeur dédié et ainsi d'obtenir une accélération. Plusieurs travaux évaluent l'impact de la précision des multiplications sur l'entraînement (Holt and Baker, 1991; Presley and Haggard, 1994; Savich et al., 2007; Gupta et al., 2015). Gupta et al. (2015) proposent une méthode, l'arrondi stochastique, qui a pour propriété intéressante de ne pas être biaisée (i.e. l'espérance de l'erreur d'arrondissage est nulle), ce qui permet d'entraîner un RdN avec des multiplications de 16 bits. Dans notre premier article (Annexe A), nous proposons une méthode, la virgule fixe dynamique (Williamson, 1991//), un hybride entre les virgules flottante et fixe, qui permet d'entraîner un RdN avec des multiplications de seulement 10 bits. Combiner notre virgule fixe dynamique avec l'arrondi stochastique de Gupta et al. (2015) permettrait probablement d'entraîner un RdN avec des multiplications de moins de 8 bits.

Alternativement, il serait intéressant de réduire le nombre de multiplications nécessaires à l'entraînement d'un RdN. Simard and Graf (1994) proposent de remplacer les multiplications par des décalages (autrement dit, contraindre une des deux opérandes des multiplications à une puissance de 2, ce qui simplifie grandement la multiplication en base 2, de la même façon qu'il est facile de multiplier par une puissance de 10 en base 10). Perrone and Cooper (1995) proposent une approximation du produit matriciel ne nécessitant que très peu de multiplications. Dans notre deuxième article (Annexe B), nous proposons une nouvelle méthode, BinaryConnect, qui consiste à contraindre les poids des connexions à une valeur binaire (e.g. 1 ou 0) pendant les propagations, ce qui divise par trois le nombre de multiplications nécessaires lors de l'entraînement d'un RdN.

2.2.5 Mémoires

La mémoire a pour rôle de stocker les données que traitent les ALUs. Le coût en transistors d'une mémoire dépends à la fois de sa capacité : la quantité maximale de données qu'elle peut stocker, de sa latence : le temps minimum qu'il faut pour accéder à une donnée quelconque, et de sa bande passante : le débit maximale des accès. Plutôt que de faire un compromis entre capacité, latence et bande passante, les systèmes numériques modernes disposent de plusieurs mémoires : des mémoires rapides de faible capacité et des mémoires lentes de grande capacité. On parle alors d'une hiérarchie de la mémoire, ce qu'illustrent les figures 2.9 et 2.11.

Revenons à nos RdNs. Effectuer un produit vecteur-matrice entre un vecteur d'entrées et une matrice de poids est un véritable défi pour la mémoire. Un RdN moderne peut avoir plusieurs centaines de millions de poids (Simonyan and Zisserman, 2014), ce qui demande une mémoire de grande capacité (plusieurs GOctets). De plus, il faut une grande bande passante pour alimenter toutes les ALUs du processeur. Une mémoire combinant une grande bande passante et une grande capacité est très coûteuse en termes de transistors ...

Une solution existe : regrouper plusieurs vecteurs d'entrées dans une matrice (appelée mini-batch) et ainsi remplacer le produit vecteur-matrice par un produit matrice-matrice. De cette manière, la matrice des poids est réutilisée pour chacun des vecteurs d'entrées du mini-batch, ce qui ouvre la porte à des optimisations. Il est par exemple possible de réaliser un produit matrice-matrice avec deux mémoires distinctes : une mémoire de grande capacité et de faible bande passante, et une mémoire de faible capacité et de grande bande passante (Jia-Wei and Kung, 1981). Le besoin en bande passante de la mémoire de grande capacité est alors divisé par $\sqrt{(M)}$, M étant la taille de la mémoire de grande bande passante.

2.2.6 Unités de Contrôle

Une unité de contrôle (CU) a pour rôle de superviser une ou plusieurs ALUs. Une CU est typiquement constituée d'un compteur et d'une mémoire à accès aléatoire (RAM), ce qu'illustre la figure 2.12. La RAM contient une liste d'instructions (le programme) qui est défini par l'utilisateur (le programmeur). Un compteur est connecté au bus d'adresse de la RAM et ce compteur est incrémenté à chaque coup d'horloge. L'ALU exécute ainsi une nouvelle instruction à chaque coup d'horloge.

Comme nous l'avons vu dans la section 2.2.4, un processeur parallèle dispose de plusieurs ALUs qui travaillent en parallèle. Cela dit, il ne dispose pas nécessairement d'une CU par ALU. Plusieurs ALUs peuvent partager la même CU, on parle alors d'un processeur vectorisé (ou SIMD), ce qu'illustre la figure 2.9. Par exemple, le GPU Titan X est un processeur hautement vectorisé (Smith, 2015). Il possède 3072 ALUs mais seulement 24 CUs. Autrement dit, chaque groupe de 128 ALUs partage la même CU et donc le même programme. Le bénéfice de ce partage est bien entendu l'économie de logique de contrôle et le coût est une relative perte de flexibilité du processeur : il ne peut que paralléliser les tâches très répétitives.

Dans le cas d'un RdN artificiel, tous les neurones ont un comportement identique. Autrement dit, un produit matriciel est un algorithme très répétitif. Il n'y a donc pas besoin d'un grand nombre de CUs pour entrainer un RdN. Le contrôle des RdNs est relativement simple. Un processeur hautement vectorisé (comme par exemple le GPU Titan X) fait très bien l'affaire.

2.2.7 Entraînement distribué

Un moyen courant d'accélérer l'entraînement de RdNs est de le distribuer sur plusieurs GPUs. La section 2.4. de (Liu, 2015) est un excellent état des connaissances en entraînement de RdN distribué. En bref, il est difficile de distribuer l'entraînement de RdNs sur plusieurs processeurs.

2.2.8 Points clés

- Les systèmes numériques sont faits de transistors.
- Leur coût dépend du nombre et de la finesse de gravure des transistors.
- La consommation énergétique augmente linéairement avec la fréquence d'horloge.
- Le niveau d'abstraction le plus populaire en conception de systèmes numériques est le RTL.
- Un processeur est fait d'ALUs, de mémoires et de CUs.
- Les ALUs d'un RdN sont de simples MACs.
- Le coût d'un multiplieur est de $O(n^2)$, n étant le nombre de bits de précision.

- Le coût d'une mémoire dépend de sa latence, sa bande passante et sa capacité.
- Les processeurs modernes disposent de mémoires rapides et de mémoires de grande capacité.
- Les besoins en mémoire des RdNs peuvent être réduits à condition de regrouper les vecteurs d'entrées dans une matrice (mini-batch).
- Un processeur vectorisé possède moins de CUs que d'ALUs.
- Le contrôle des RdNs est relativement simple.
- Il est difficile de distribuer l'entraînement de RdNs sur plusieurs processeurs.

CHAPITRE 3 SYNTHÈSE DE L'ENSEMBLE DU TRAVAIL

Tableau 3.1 Estimations des puissances de calcul de processeurs utilisant des multiplieurs 10 bits ou/et BinaryConnect. Diminuer le coût des multiplieurs permet d'augmenter leur nombre et donc d'obtenir une accélération. Il est possible d'avoir $(32/10)^2 = 10.24$ multiplieurs 10 bits pour le prix d'un multiplieur 32 bits. BinaryConnect divise par 3 le nombre des multiplications nécessaires lors de l'entraînement d'un RdN. La puissance de calcul est le nombre de connexions synaptiques qu'il est possible de simuler par seconde (TCo/s).

Méthode	Accélération	Puissance de calcul (TCo/s)
Titan X (32 bits)	1	3
Multiplieurs 10 bits	10.24	30.72
BinaryConnect	3	9
Combinaison des deux	30.72	92.16

L'objectif de ce mémoire est d'accélérer d'éventuels processeurs dédiés à l'apprentissage profond. Dans le chapitre précédant, nous avons vu que les multiplieurs sont les opérateurs arithmétiques les plus couteux d'un processeur dédié aux RdNs et que leur coût en transistors est de $O(n^2)$, n étant le nombre de bits de précision. L'objectif de ce chapitre est de faire une synthèse de l'ensemble du travail réalisé.

3.1 Entraînement de réseaux de neurones avec des multiplications moins précises

Dans notre premier article (Annexe A), nous entraînons un ensemble de RdNs à l'état de l'art (les réseaux Maxout) sur trois ensembles de données de référence : MNIST, CIFAR-10 et SVHN. Ils sont entraînés avec trois formats distincts : virgule flottante, virgule fixe et virgule fixe dynamique. Pour chacun de ces ensembles de données et pour chacun de ces formats, nous évaluons l'impact de la précision des multiplications sur l'erreur finale après l'entrainement. Nous trouvons qu'une précision très faible est suffisante non seulement pour tester des RdNs, mais aussi pour les entraîner. Par exemple, il est possible d'entraîner des réseaux Maxout avec des multiplications 10 bits.

3.2 BinaryConnect : entraînement de réseaux de neurones avec des poids binaires pendant les propagations

Des poids binaires, c'est à dire des poids qui sont contraints à seulement deux valeurs possibles (e.g. -1 ou 1), permettraient de diviser par trois le nombre de multiplications nécessaires lors de l'entraînement d'un RdN. Dans notre deuxième article (Annexe B), nous introduisons BinaryConnect, une méthode qui consiste à entraîner un RdN avec des poids binaires durant les propagations en avant et en arrière, tout en conservant la précision des poids stockés dans lesquels les gradients sont accumulés. Comme les autres variantes de Dropout (Srivastava, 2013; Srivastava et al., 2014), nous montrons que BinaryConnect agit comme régulariseur et nous obtenons des résultats proches de l'état de l'art avec BinaryConnect sur le MNIST invariant aux permutations. Le tableau 3.1 estime l'accélération résultante de nos deux articles.

CHAPITRE 4 DISCUSSION GÉNÉRALE

4.1 Entraînement de réseaux de neurones avec des multiplications moins précises

Dans notre premier article (Annexe A), nous proposons une méthode, la virgule fixe dynamique (Williamson, 1991//), un hybride entre les virgules flottante et fixe, qui permet d'entraîner un RdN avec des multiplications de seulement 10 bits, ce qui est une nette amélioration comparé aux travaux connexes qui ne descendent pas en dessous de 16 bits (Holt and Baker, 1991; Presley and Haggard, 1994; Savich et al., 2007; Gupta et al., 2015).

Gupta et al. (2015) proposent une méthode, l'arrondi stochastique, qui a pour propriété intéressante de ne pas être biaisée (i.e. l'espérance de l'erreur d'arrondi est nulle), ce qui permet d'entraîner un RdN avec des multiplications de 16 bits. Combiner notre virgule fixe dynamique avec l'arrondi stochastique de Gupta et al. (2015) permettrait probablement d'entraîner un RdN avec des multiplications de moins de 8 bits.

Le coût en transistors d'un multiplieur augmente de manière quadratique avec sa précision (cf. chapitre 2). Diminuer le coût des multiplieurs permet d'augmenter leur nombre et donc d'obtenir une accélération. Il est possible d'avoir $(32/10)^2 = 10.24$ multiplieurs 10 bits pour le prix d'un multiplieur 32 bits (précision des GPUs) et $(16/10)^2 = 2.56$ multiplieurs 10 bits pour le prix d'un multiplieur 16 bits (précision des travaux connexes), ce qu'illustre le tableau 3.1.

4.2 BinaryConnect : entraînement de réseaux de neurones avec des poids binaires pendant les propagations

Dans notre deuxième article (Annexe B), nous proposons une nouvelle méthode, BinaryConnect, qui consiste à contraindre les poids des connexions à une valeur binaire (e.g. -1 ou 1) pendant les propagations, ce qui divise par trois le nombre de multiplications nécessaires lors de l'entraînement d'un RdN. De plus, comme les autres variantes de Dropout (Srivastava, 2013; Srivastava et al., 2014), nous avons montré que BinaryConnect augmente la capacité de généralisation des RdNs.

Simard and Graf (1994) proposent de remplacer les multiplications par des décalages lors de l'entraînement de RdNs (autrement dit, contraindre une des deux opérandes des multiplications à une puissance de 2, ce qui simplifie grandement la multiplication en base 2, de

la même façon qu'il est facile de multiplier par une puissance de 10 en base 10). Il serait intéressant de remplacer les multiplications restantes de BinaryConnect par des décalages. Perrone and Cooper (1995) proposent une approximation du produit matriciel ne nécessitant que très peu de multiplications.

CHAPITRE 5 CONCLUSION

5.1 Synthèse

Dans notre premier article (Annexe A), nous avons montré que la virgule fixe dynamique, un hybride entre les virgules flottante et fixe, permet d'entraîner des RdNs avec des multiplieurs de seulement 10 bits. Dans notre second article (Annexe B), nous avons introduit BinaryConnect, une méthode qui consiste à contraindre les poids des connexions à une valeur binaire (e.g. 1 ou 0) pendant les propagations, ce qui divise par trois le nombre de multiplications nécessaires lors de l'entraînement d'un RdN tout en augmentant sa performance.

5.2 Accélération résultante

Le coût en transistors d'un multiplieur augmente de manière quadratique avec sa précision (cf. chapitre 2). Diminuer le coût des multiplieurs permet d'augmenter leur nombre et donc d'obtenir une accélération. Il est possible d'avoir $(32/10)^2 = 10.24$ multiplieurs 10 bits pour le prix d'un multiplieur 32 bits (précision des GPUs). L'accélération théorique résultante de nos deux articles est donc de $10.24 \times 3 = 30.72$, ce qu'illustre le tableau 3.1.

La loi de Moore (Moore, 1965), une loi empirique que certains pensent être en fin de vie, nous dit que le nombre de transistors d'un processeur double tous les deux ans. Une accélération de 30.72 correspond donc à environs $2 \times log_2(30.72) = 10$ années de loi de Moore. Cela veut dire qu'un processeur utilisant la virgule fixe dynamique 10 bits et BinaryConnect ne serait égalé (pour entraîner des RdNs) par les processeurs généraux (GPUs) que dans 10 ans. Cette solide accélération va permettre d'entraîner des RdNs avec plus de connexions et donc plus de capacité, ce qui est un pas de plus vers une IA forte.

5.3 Travaux futurs

Nous avons montré qu'il est possible d'accélérer l'entraînement des RdNs en réduisant la précision et le nombre des multiplications. Des travaux futurs devront réaliser cette accélération hypothétique sur un véritable processeur. L'idéal serait de modifier un processeur vectorisé (figure 2.9) existant, comme par exemple un GPU, afin de pouvoir réutiliser les librairies d'apprentissage profond les plus populaires (Bergstra et al., 2010; Bastien et al., 2012). Les modifications à apporter serait :

– réduire à 10 bits la précision des ALUs,

– enlever les multiplieurs des ALUs, car il est possible de faire les multiplications restantes (après BinaryConnect) avec des additionneurs et des décaleurs (Wikipédia, 2012),
– augmenter le nombre de ces ALUs (dont le coût est très réduit),
– et redimensionner les mémoires du processeur (si nécessaire).

D'autres travaux devront réduire davantage le nombre des multiplications nécessaires à l'entraînement d'un RdN. Un moyen simple serait de binariser les sorties des neurones ou leurs gradients, ou tout du moins les contraindre à des puissances de deux et ainsi remplacer les multiplications par des décalages (Simard and Graf, 1994).

L'idéal serait bien entendu de pouvoir binariser toutes les variables de l'entraînement : les poids (BinaryConnect), les sorties des neurones et leurs gradients. Les multiplieurs seraient alors remplacés par de simples portes logiques ET et les accumulateurs par des compteurs. De manière assez encourageante, des travaux très récents montrent qu'il est possible de binariser les poids et les sorties des neurones après l'entraînement sans trop sacrifier en performance (Kim and Smaragdis, 2015).

RÉFÉRENCES

D. Bahdanau, K. Cho, et Y. Bengio, “Neural machine translation by jointly learning to align and translate”, dans *ICLR’2015, arXiv :1409.0473*, 2015.

F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard, et Y. Bengio, “Theano : new features and speed improvements”, Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

Y. Bengio, R. Ducharme, P. Vincent, et C. Jauvin, “A neural probabilistic language model”, *Journal of Machine Learning Research*, vol. 3, pp. 1137–1155, 2003.

Y. Bengio, I. J. Goodfellow, et A. Courville, “Deep learning”, 2015, book in preparation for MIT Press. En ligne : `http://www.iro.umontreal.ca/~bengioy/dlbook`

J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, et Y. Bengio, “Theano : a CPU and GPU math expression compiler”, dans *Proceedings of the Python for Scientific Computing Conference (SciPy)*, Juin 2010, oral Presentation.

T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, et O. Temam, “Diannao : A small-footprint high-throughput accelerator for ubiquitous machine-learning”, dans *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ACM, 2014, pp. 269–284.

Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, “Dadiannao : A machine-learning supercomputer”, dans *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 2014, pp. 609–622.

J. David, K. Kalach, et N. Tittley, “Hardware complexity of modular multiplication and exponentiation”, *Computers, IEEE Transactions on*, vol. 56, no. 10, pp. 1308–1319, Oct 2007. DOI : `10.1109/TC.2007.1084`

R. Dawkins, *The selfish gene*. Oxford university press, 2006, no. 199.

J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, et A. Y. Ng, “Large scale distributed deep networks”, dans *NIPS’2012*, 2012.

J. Devlin, R. Zbib, Z. Huang, T. Lamar, R. Schwartz, et J. Makhoul, “Fast and robust neural network joint models for statistical machine translation”, dans *Proc. ACL’2014*, 2014.

R. Eckhorn, R. Bauer, W. Jordan, M. Brosch, W. Kruse, M. Munk, et H. Reitboeck, “Coherent oscillations : A mechanism of feature linking in the visual cortex ?” *Biological cybernetics*, vol. 60, no. 2, pp. 121–130, 1988.

C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, et Y. LeCun, “Neuflow : A runtime reconfigurable dataflow processor for vision”, dans *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*. IEEE, 2011, pp. 109–116.

X. Glorot, A. Bordes, et Y. Bengio, “Deep sparse rectifier neural networks”, dans *AISTATS’2011*, 2011.

C. M. Gray et W. Singer, “Stimulus-specific neuronal oscillations in orientation columns of cat visual cortex”, *Proceedings of the National Academy of Sciences*, vol. 86, no. 5, pp. 1698–1702, 1989.

C. M. Gray, P. König, A. K. Engel, W. Singer *et al.*, “Oscillatory responses in cat visual cortex exhibit inter-columnar synchronization which reflects global stimulus properties”, *Nature*, vol. 338, no. 6213, pp. 334–337, 1989.

S. Gupta, A. Agrawal, K. Gopalakrishnan, et P. Narayanan, “Deep learning with limited numerical precision”, *CoRR*, vol. abs/1502.02551, 2015. En ligne : `http://arxiv.org/abs/1502.02551`

G. Hinton, L. Deng, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, et B. Kingsbury, “Deep neural networks for acoustic modeling in speech recognition”, *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, Nov. 2012.

J. L. Holt et T. E. Baker, “Back propagation simulations using limited precision calculations”, dans *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, vol. 2. IEEE, 1991, pp. 121–126.

K. Hornik, “Approximation capabilities of multilayer feedforward networks”, *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.

Y. Jia, “Learning semantic image representations at a large scale”, 2014.

H. Jia-Wei et H.-T. Kung, “I/o complexity : The red-blue pebble game”, dans *Proceedings of the thirteenth annual ACM symposium on Theory of computing*. ACM, 1981, pp. 326–333.

M. Kim et P. Smaragdis, “Bitwise neural networks”, dans *ICML Workshop*, 2015.

S. K. Kim, L. C. McAfee, P. L. McMahon, et K. Olukotun, “A highly scalable restricted Boltzmann machine FPGA implementation”, dans *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE, 2009, pp. 367–372.

A. Krizhevsky, I. Sutskever, et G. Hinton, “ImageNet classification with deep convolutional neural networks”, dans *NIPS’2012*, 2012.

P. Langlois, “Inf3500 - conception et réalisation de systèmes numériques - notes de cours”, Juillet 2013.

Y. LeCun, L. Bottou, Y. Bengio, et P. Haffner, “Gradient-based learning applied to document recognition”, *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

J. Y. Lettvin, H. R. Maturana, W. S. McCulloch, et W. H. Pitts, “What the frog’s eye tells the frog’s brain”, *Proceedings of the IRE*, vol. 47, no. 11, pp. 1940–1951, 1959.

L. Liu, “Acoustic models for speech recognition using deep neural networks based on approximate math”, Thèse de doctorat, Massachusetts Institute of Technology, 2015.

V. Mnih, K. Kavukcuoglo, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidgeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, et D. Hassabis, “Human-level control through deep reinforcement learning”, *Nature*, vol. 518, pp. 529–533, 2015.

G. Moore, “Moore’s law”, *Electronics Magazine*, 1965.

A. Mordvintsev, C. Olah, et M. Tyka, “Inceptionism : Going deeper into neural networks”, 2015, accessed : 2015-06-30. En ligne : `http://googleresearch.blogspot.co.uk/2015/06/inceptionism-going-deeper-into-neural.html`

V. Nair et G. Hinton, “Rectified linear units improve restricted Boltzmann machines”, dans *ICML’2010*, 2010.

M. P. Perrone et L. N. Cooper, “The ni1000 : High speed parallel vlsi for implementing multilayer perceptrons”, dans *Advances in Neural Information Processing Systems*, 1995, pp. 747–754.

P.-H. Pham, D. Jelaca, C. Farabet, B. Martini, Y. LeCun, et E. Culurciello, “Neuflow : dataflow vision processing system-on-a-chip”, dans *Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on*. IEEE, 2012, pp. 1044–1047.

R. K. Presley et R. L. Haggard, “A fixed point implementation of the backpropagation learning algorithm”, dans *Southeastcon’94. Creative Technology Transfer-A Global Affair., Proceedings of the 1994 IEEE*. IEEE, 1994, pp. 136–138.

R. Raina, A. Madhavan, et A. Y. Ng, “Large-scale deep unsupervised learning using graphics processors”, dans *ICML’2009*, 2009.

T. Sainath, A. rahman Mohamed, B. Kingsbury, et B. Ramabhadran, “Deep convolutional neural networks for LVCSR”, dans *ICASSP 2013*, 2013.

A. W. Savich, M. Moussa, et S. Areibi, “The impact of arithmetic representation on implementing mlp-bp on fpgas : A study”, *Neural Networks, IEEE Transactions on*, vol. 18, no. 1, pp. 240–252, 2007.

P. Simard et H. P. Graf, “Backpropagation without multiplication”, dans *Advances in Neural Information Processing Systems*, 1994, pp. 232–239.

T. Simonite, “Why and how baidu cheated an artificial intelligence test”, 2015, accessed : 2015-06-29. En ligne : `http://www.technologyreview.com/view/538111/why-and-how-baidu-cheated-an-artificial-intelligence-test/`

K. Simonyan et A. Zisserman, “Very deep convolutional networks for large-scale image recognition”, *arXiv preprint arXiv :1409.1556*, 2014.

R. Smith, “The nvidia geforce gtx titan x review”, 2015, accessed : 2015-06-17. En ligne : `http://www.anandtech.com/show/9059/the-nvidia-geforce-gtx-titan-x-review`

N. Srivastava, “Improving neural networks with dropout”, Mémoire de maîtrise, U. Toronto, 2013.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, et R. Salakhutdinov, “Dropout : A simple way to prevent neural networks from overfitting”, *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014. En ligne : `http://jmlr.org/papers/v15/srivastava14a.html`

I. Sutskever, O. Vinyals, et Q. V. Le, “Sequence to sequence learning with neural networks”, dans *NIPS’2014*, 2014.

C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, et A. Rabinovich, “Going deeper with convolutions”, arXiv :1409.4842, Rapp. tech., 2014.

N. Weste et D. Harris, *CMOS VLSI Design : A Circuits and Systems Perspective*, 4e éd. USA : Addison-Wesley Publishing Company, 2010.

Wikipédia, “Technique de la multiplication dans l’Égypte antique — wikipédia, l’encyclopédie libre”, 2012, [En ligne ; Page disponible le 28-juin-2015]. En ligne : `http://fr.wikipedia.org/w/index.php?title=Technique_de_la_multiplication_dans_l%27%C3%89gypte_antique&oldid=83688343`

D. Williamson, “Dynamically scaled fixed point arithmetic”, New York, NY, USA, 1991//, pp. 315 – 18, dynamic scaling ;iteration stages ;digital filters ;overflow probability ;fixed point arithmetic ;fixed-point filter ;. En ligne : `http://dx.doi.org/10.1109/PACRIM.1991.160742`

D. R. Wilson et T. R. Martinez, “The general inefficiency of batch training for gradient descent learning”, *Neural Networks*, vol. 16, no. 10, pp. 1429–1451, 2003.

ANNEXE A ARTICLE 1 : LOW PRECISION STORAGE FOR DEEP LEARNING

Paru le 8 Mai 2015 aux ateliers ICLR 2015.

Deep learning with low precision multiplications

Matthieu Courbariaux[1], **Yoshua Bengio**[2] **and Jean-Pierre David**[1]

[1]École Polytechnique de Montréal

[2]Université de Montréal and CIFAR Senior Fellow

Keywords: precision, multipliers, fixed point, deep learning, neural networks

Abstract

Multipliers are the most space and power-hungry arithmetic operators of the digital implementation of deep neural networks. We train a set of state-of-the-art neural networks (Maxout networks) on three benchmark datasets: MNIST, CIFAR-10 and SVHN. They are trained with three distinct formats: floating point, fixed point and dynamic fixed point. For each of those datasets and for each of those formats, we assess the impact of the precision of the multiplications on the final error after training. We find that *very low precision is sufficient* not just for running trained networks but *also for training them*. For example, it is possible to train Maxout networks with **10** bits multiplications.

1 Introduction

The *training* of deep neural networks is very often limited by hardware. Lots of previous works address the best exploitation of general-purpose hardware, typically CPU clusters (Dean *et al.*, 2012) and GPUs (Coates *et al.*, 2009; Krizhevsky *et al.*, 2012a). Faster implementations usually lead to state of the art results (Dean *et al.*, 2012; Krizhevsky *et al.*, 2012a).

Actually, such approaches always consist in adapting the algorithm to best exploit state of the art general-purpose hardware. Nevertheless, some dedicated deep learning hardware is appearing as well. FPGA and ASIC implementations claim a better power efficiency than general-purpose hardware (Kim *et al.*, 2009; **?**; Pham *et al.*, 2012; Chen *et al.*, 2014a,b). In contrast with general-purpose hardware, dedicated hardware such as ASIC and FPGA enables to build the hardware from the algorithm.

Hardware is mainly made out of memories and arithmetic operators. Multipliers are the most space and power-hungry arithmetic operators of the digital implementation of deep neural networks. The objective of this article is to assess the possibility to reduce the precision of the multipliers for deep learning:

- We train deep neural networks with low precision multipliers and high precision accumulators (Section 2).

- We carry out experiments with three distinct formats:

    1. Floating point (Section 3)

    2. Fixed point (Section 4)

3. Dynamic fixed point, which we think is a good compromise between floating and fixed points (Section 5)

- We use a higher precision for the parameters during the updates than during the forward and backward propagations (Section 6).

- Maxout networks (Goodfellow *et al.*, 2013a) are a set of state-of-the-art neural networks (Section 7). We train Maxout networks with slightly less capacity than Goodfellow *et al.* (2013a) on three benchmark datasets: MNIST, CIFAR-10 and SVHN (Section 8).

- For each of the three datasets and for each of the three formats, we assess the impact of the precision of the multiplications on the final error of the training. We find that *very low precision multiplications are sufficient* not just for running trained networks but *also for training them* (Section 9). We made our code available [1].

2 Multiplier-accumulators

Multiplier (bits)	Accumulator (bits)	Adaptive Logic Modules (ALMs)
32	32	504
16	32	138
16	16	128

Table 1: Cost of a fixed point multiplier-accumulator on a Stratix V Altera FPGA

[1] `https://github.com/MatthieuCourbariaux/deep-learning-multipliers`

Algorithm 1 Forward propagation with low precision multipliers.

for all layers **do**

Reduce the precision of the parameters and the inputs

Apply convolution or dot product with high precision (multiply-accumulate)

Reduce the precision of the weighted sum

Apply activation functions with high precision

end for

Reduce the precision of the outputs

Applying a deep neural network (DNN) mainly consists in convolutions and matrix multiplications. The key arithmetic operation of DNNs is thus the multiply-accumulate operation. Artificial neurons are basically multiplier-accumulators computing weighted sums of their inputs. The cost of a multiplier varies as the square of the precision for small widths while the cost of adders and accumulators varies as a linear function of the precision (David *et al.*, 2007). As a result, the cost of a multiplier-accumulator mainly depends on the precision of the multiplier, as shown in table 1. In this article, we train deep neural networks with low precision multipliers and high precision accumulators, as illustrated in Algorithm 1.

In modern FPGAs, the multiplications can also be implemented with dedicated DSP blocks/slices. One DSP block/slice can implement a single 27×27 multiplier, a double 18×18 multiplier or a triple 9×9 multiplier. Reducing the precision can thus lead to a gain of 3 in the number of available multipliers inside a modern FPGA.

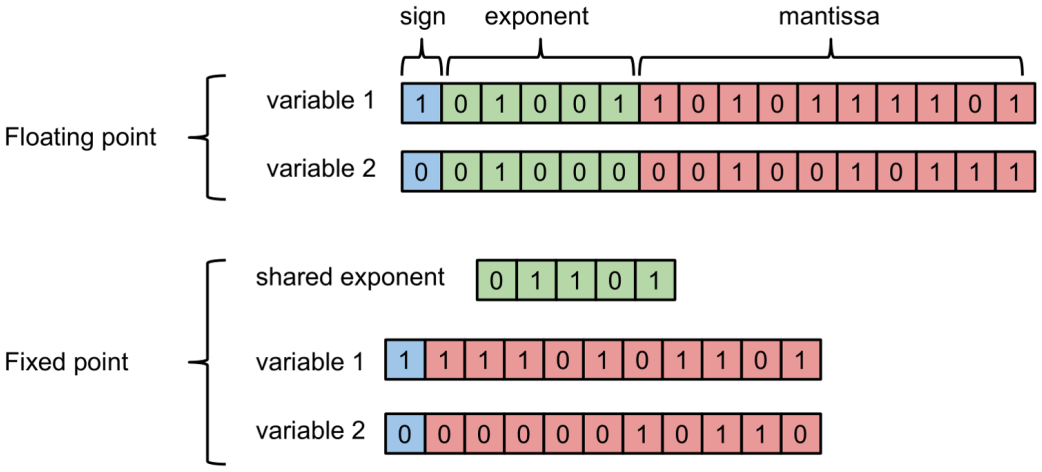

Figure 1: Comparison of the floating point and fixed point formats.

3 Floating point

Floating point formats are often used to represent real values. They consist in a sign, an exponent, and a mantissa, as illustrated in figure 1. The exponent gives the floating point formats a wide range, and the mantissa gives them a good precision. One can compute the value of a single floating point number using the following formula:

$$value = (-1)^{sign} \times \left(1 + \frac{mantissa}{2^{23}}\right) \times 2^{(exponent-127)}$$

Table 2 shows the exponent and mantissa widths associated with each floating point format. In our experiments, we use single precision floating point format as our reference because it is the most widely used format in deep learning, especially for GPU computation. We show that the use of half precision floating point format has little to no impact on the training of neural networks. At the time of writing this article, no standard exists below the half precision floating point format.

Format	Total bit-width	Exponent bit-width	Mantissa bit-width
Double precision floating point	64	11	52
Single precision floating point	32	8	23
Half precision floating point	16	5	10

Table 2: Definitions of double, single and half precision floating point formats.

4 Fixed point

Fixed point formats consist in a signed mantissa and a global scaling factor shared between all fixed point variables. The scaling factor can be seen as the position of the radix point. It is usually fixed, hence the name ”fixed point”. Reducing the scaling factor reduces the range and augments the precision of the format. The scaling factor is typically a power of two for computational efficiency (the scaling multiplications are replaced with shifts). As a result, fixed point format can also be seen as a floating point format with a *unique shared fixed exponent* , as illustrated in figure 1. Fixed point format is commonly found on embedded systems with no FPU (Floating Point Unit). It relies on integer operations. It is hardware-wise cheaper than its floating point counterpart, as the exponent is shared and fixed.

5 Dynamic fixed point

When training deep neural networks,

1. activations, gradients and parameters have *very different ranges*.

Algorithm 2 Policy to update a scaling factor.

Require: a matrix M, a scaling factor s_t, and a maximum overflow rate r_{max}.

Ensure: an updated scaling factor s_{t+1}.

if the overflow rate of $M > r_{max}$ **then**

$s_{t+1} \leftarrow 2 \times s_t$

else if the overflow rate of $2 \times M \leq r_{max}$ **then**

$s_{t+1} \leftarrow s_t/2$

else

$s_{t+1} \leftarrow s_t$

end if

2. gradients ranges *slowly diminish* during the training.

As a result, the fixed point format, with its unique shared fixed exponent, is ill-suited to deep learning.

The dynamic fixed point format (Williamson, 1991) is a variant of the fixed point format in which there are *several scaling factors* instead of a single global one. Those scaling factors are *not fixed*. As such, it can be seen as a compromise between floating point format - where each scalar variable owns its scaling factor which is updated during each operations - and fixed point format - where there is only one global scaling factor which is never updated. With dynamic fixed point, a few grouped variables share a scaling factor which is updated from time to time to reflect the statistics of values in the group.

In practice, we associate each layer's weights, bias, weighted sum, outputs (post-nonlinearity) and the respective gradients vectors and matrices with a different scaling

factor. Those scaling factors are initialized with a global value. The initial values can also be found during the training with a higher precision format. During the training, we update those scaling factors at a given frequency, following the policy described in Algorithm 2.

6 Updates vs. propagations

We use a higher precision for the parameters during the updates than during the forward and backward propagations, respectively called fprop and bprop. The idea behind this is to be able to accumulate small changes in the parameters (which requires more precision) and while on the other hand sparing a few bits of memory bandwidth during fprop. This can be done because of the implicit averaging performed via stochastic gradient descent during training:

$$\theta_{t+1} = \theta_t - \epsilon \frac{\partial C_t(\theta_t)}{\partial \theta_t}$$

where $C_t(\theta_t)$ is the cost to minimize over the minibatch visited at iteration t using θ_t as parameters and ϵ is the learning rate. We see that the resulting parameter is the sum

$$\theta_T = \theta_0 - \epsilon \sum_{t=1}^{T-1} \frac{\partial C_t(\theta_t)}{\partial \theta_t}.$$

The terms of this sum are not statistically independent (because the value of θ_t depends on the value of θ_{t-1}) but the dominant variations come from the random sample of examples in the minibatch (θ moves slowly) so that a strong averaging effect takes place, and each contribution in the sum is relatively small, hence the demand for sufficient precision (when adding a small number with a large number).

7 Maxout networks

A Maxout network is a multi-layer neural network that uses maxout units in its hidden layers. A maxout unit outputs the maximum of a set of k dot products between k weight vectors and the input vector of the unit (e.g., the output of the previous layer):

$$h_i^l = \max_{j=1}^{k}(b_{i,j}^l + w_{i,j}^l \cdot h^{l-1})$$

where h^l is the vector of activations at layer l and weight vectors $w_{i,j}^l$ and biases $b_{i,j}^l$ are the parameters of the j-th filter of unit i on layer l.

A maxout unit can be seen as a generalization of the rectifying units (Jarrett *et al.*, 2009; Nair and Hinton, 2010; Glorot *et al.*, 2011; Krizhevsky *et al.*, 2012b)

$$h_i^l = \max(0, b_i^l + w_i^l \cdot h^{l-1})$$

which corresponds to a maxout unit when $k = 2$ and one of the filters is forced at 0 (Goodfellow *et al.*, 2013a). Combined with dropout, a very effective regularization method (Hinton *et al.*, 2012), maxout networks achieved state-of-the-art results on a number of benchmarks (Goodfellow *et al.*, 2013a), both as part of fully connected feedforward deep nets and as part of deep convolutional nets. The dropout technique provides a good approximation of model averaging with shared parameters across an exponentially large number of networks that are formed by subsets of the units of the original noise-free deep network.

8 Baseline results

We train Maxout networks with slightly less capacity than Goodfellow *et al.* (2013a) on three benchmark datasets: MNIST, CIFAR-10 and SVHN. In Section 9, we use the same hyperparameters as in this section to train Maxout networks with low precision multiplications.

Dataset	Dimension	Labels	Training set	Test set
MNIST	784 (28×28 grayscale)	10	60K	10K
CIFAR-10	3072 (32×32 color)	10	50K	10K
SVHN	3072 (32×32 color)	10	604K	26K

Table 3: Overview of the datasets used in this paper.

8.1 MNIST

The MNIST (LeCun *et al.*, 1998) dataset is described in Table 3. We do not use any data-augmentation (e.g. distortions) nor any unsupervised pre-training. We simply use minibatch stochastic gradient descent (SGD) with momentum. We use a linearly decaying learning rate and a linearly saturating momentum. We regularize the model with dropout and a constraint on the norm of each weight vector, as in (Srebro and Shraibman, 2005).

We train two different models on MNIST. The first is a permutation invariant (PI) model which is unaware of the structure of the data. It consists in two fully connected maxout layers followed by a softmax layer. The second model consists in three convo-

Format	Prop.	Up.	PI MNIST	MNIST	CIFAR-10	SVHN
Goodfellow *et al.* (2013a)	32	32	0.94%	0.45%	11.68%	2.47%
Single precision floating point	32	32	1.05%	0.51%	14.05%	2.71%
Half precision floating point	16	16	1.10%	0.51%	14.14%	3.02%
Fixed point	20	20	1.39%	0.57%	15.98%	2.97%
Dynamic fixed point	10	12	1.28%	0.59%	14.82%	4.95%

Table 4: Test set error rates of single and half floating point formats, fixed and dynamic fixed point formats on the permutation invariant (PI) MNIST, MNIST (with convolutions, no distortions), CIFAR-10 and SVHN datasets. **Prop.** is the bit-width of the propagations and **Up.** is the bit-width of the parameters updates. The single precision floating point line refers to the results of our experiments. It serves as a baseline to evaluate the degradation brought by lower precision.

lutional maxout hidden layers (with spatial max pooling on top of the maxout layers) followed by a densely connected softmax layer.

This is the same procedure as in Goodfellow *et al.* (2013a), except that we do not train our model on the validation examples. As a consequence, our test error is slightly larger than the one reported in Goodfellow *et al.* (2013a). The final test error is in Table 4.

8.2 CIFAR-10

Some comparative characteristics of the CIFAR-10 (Krizhevsky and Hinton, 2009) dataset are given in Table 3. We preprocess the data using global contrast normal-

ization and ZCA whitening. The model consists in three convolutional maxout layers, a fully connected maxout layer, and a fully connected softmax layer. We follow a similar procedure as with the MNIST dataset. This is the same procedure as in Goodfellow *et al.* (2013a), except that we reduced the number of hidden units and that we do not train our model on the validation examples. As a consequence, our test error is slightly larger than the one reported in Goodfellow *et al.* (2013a). The final test error is in Table 4.

8.3 Street View House Numbers

The SVHN (Netzer *et al.*, 2011) dataset is described in Table 3. We applied local contrast normalization preprocessing the same way as Zeiler and Fergus (2013). The model consists in three convolutional maxout layers, a fully connected maxout layer, and a fully connected softmax layer. Otherwise, we followed the same approach as on the MNIST dataset. This is the same procedure as in Goodfellow *et al.* (2013a), except that we reduced the length of the training. As a consequence, our test error is bigger than the one reported in Goodfellow *et al.* (2013a). The final test error is in Table 4.

9 Low precision results

9.1 Floating point

Half precision floating point format has little to no impact on the test set error rate, as shown in Table 4. We conjecture that a high-precision fine-tuning could recover the small degradation of the error rate.

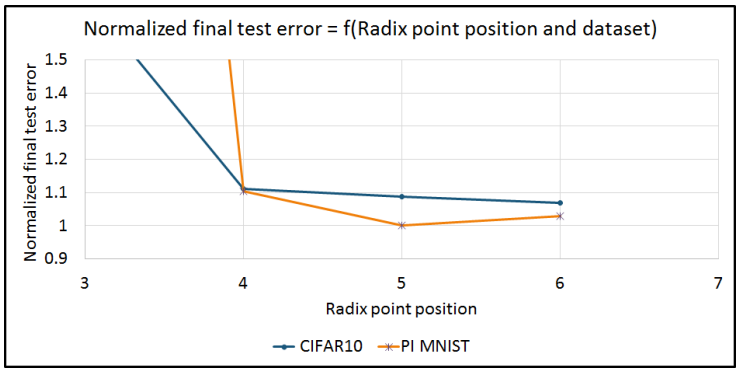

Figure 2: Final test error depending on the radix point position (5 means after the 5th most significant bit) and the dataset (permutation invariant MNIST and CIFAR-10). The final test errors are normalized, that is to say divided by the dataset single float test error. The propagations and parameter updates bit-widths are both set to 31 bits (32 with the sign).

9.2 Fixed point

The optimal radix point position in fixed point is after the fifth (or arguably the sixth) most important bit, as illustrated in Figure 2. The corresponding range is approximately [-32,32]. The corresponding scaling factor depends on the bit-width we are using. The minimum bit-width for propagations in fixed point is 19 (20 with the sign). Below this bit-width, the test set error rate rises very sharply, as illustrated in Figure 3. The minimum bit-width for parameter updates in fixed point is 19 (20 with the sign). Below this bit-width, the test set error rate rises very sharply, as illustrated in Figure 4. Doubling the number of hidden units does not allow any further reduction of the bit-widths on the permutation invariant MNIST. In the end, using 19 (20 with the sign) bits for both

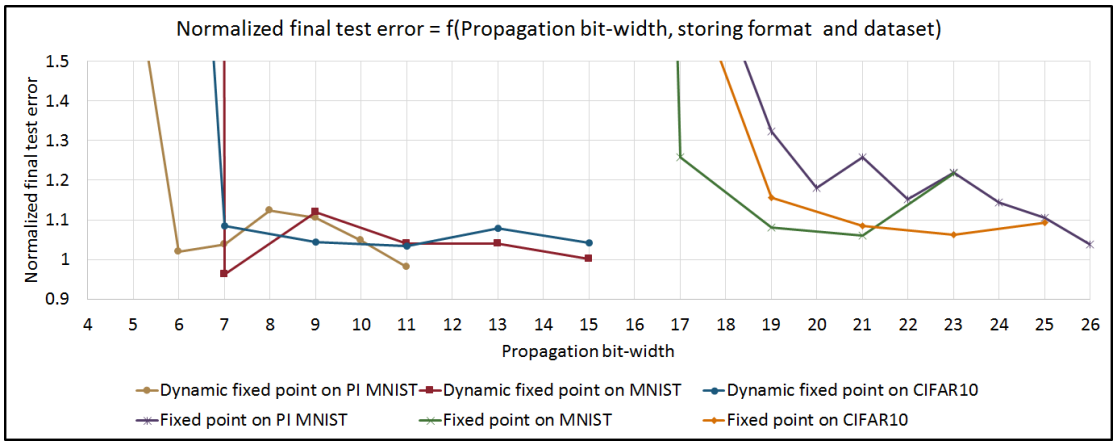

Figure 3: Final test error depending on the propagations bit-width, the format (dynamic fixed or fixed point) and the dataset (permutation invariant MNIST, MNIST and CIFAR-10). The final test errors are normalized, which means that they are divided by the dataset single float test error. For both formats, the parameter updates bit-width is set to 31 bits (32 with the sign). For fixed point format, the radix point is set after the fifth bit. For dynamic fixed point format, the maximum overflow rate is set to 0.01%.

the propagations and the parameter updates has little impact on the final test error, as shown in Table 4.

9.3 Dynamic fixed point

We find the initial scaling factors by training with a higher precision format. Once those scaling factors are found, we reinitialize the model parameters. We update the scaling factors once every 10000 examples. Augmenting the maximum overflow rate allows us to reduce the propagations bit-width but it also significantly augments the final test error rate, as illustrated in Figure 5. As a consequence, we use a low maximum overflow rate

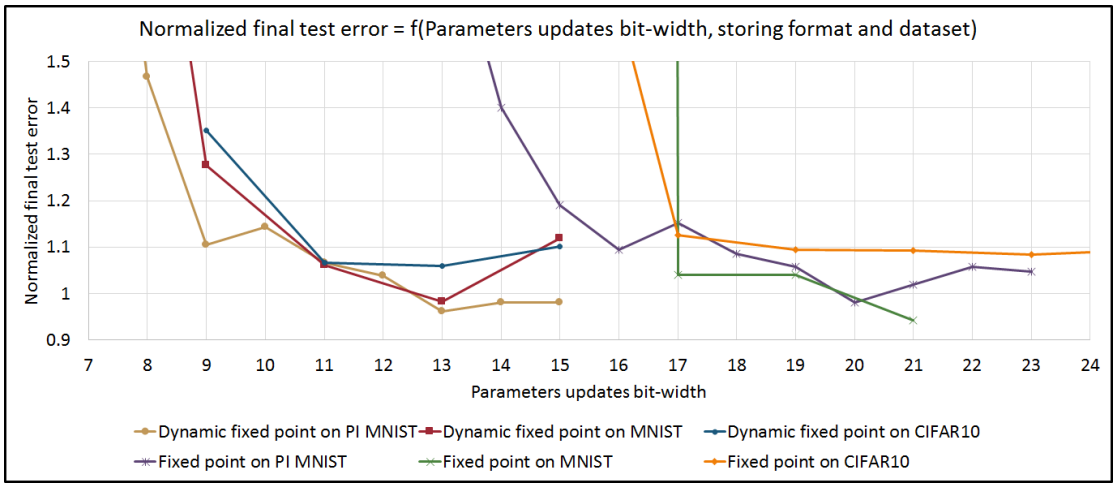

Figure 4: Final test error depending on the parameter updates bit-width, the format (dynamic fixed or fixed point) and the dataset (permutation invariant MNIST, MNIST and CIFAR-10). The final test errors are normalized, which means that they are divided by the dataset single float test error. For both formats, the propagations bit-width is set to 31 bits (32 with the sign). For fixed point format, the radix point is set after the fifth bit. For dynamic fixed point format, the maximum overflow rate is set to 0.01%.

of 0.01% for the rest of the experiments. The minimum bit-width for the propagations in dynamic fixed point is 9 (10 with the sign). Below this bit-width, the test set error rate rises very sharply, as illustrated in Figure 3. The minimum bit-width for the parameter updates in dynamic fixed point is 11 (12 with the sign). Below this bit-width, the test set error rate rises very sharply, as illustrated in Figure 4. Doubling the number of hidden units does not allow any further reduction of the bit-widths on the permutation invariant MNIST. In the end, using 9 (10 with the sign) bits for the propagations and 11 (12 with the sign) bits for the parameter updates has little impact on the final test error, with the

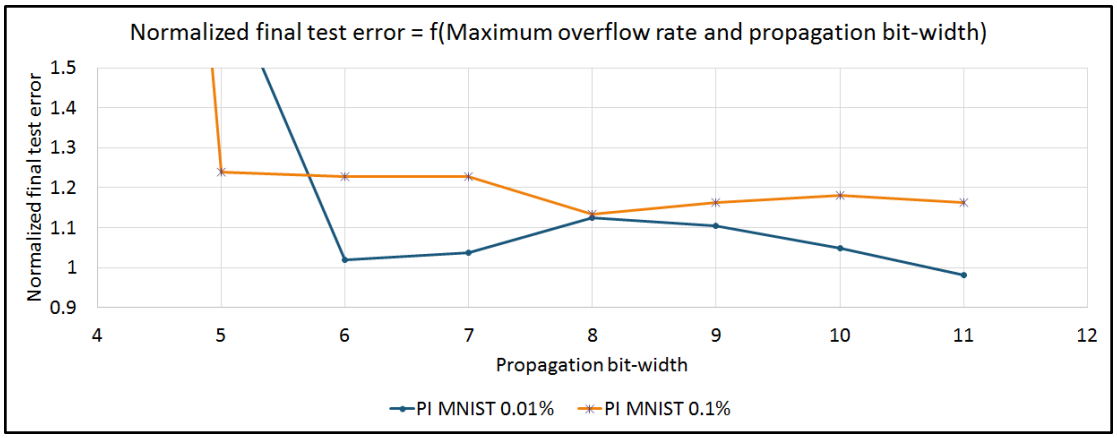

Figure 5: Final test error depending on the maximum overflow rate and the propagations bit-width. The final test errors are normalized, which means that they are divided by the dataset single float test error. The parameter updates bit-width is set to 31 bits (32 with the sign).

exception of the SVHN dataset, as shown in Table 4. This is significantly better than fixed point format, which is consistent with our predictions of Section 5.

10 Related works

Vanhoucke *et al.* (2011) use 8 bits linear quantization to store activations and weights. Weights are scaled by taking their maximum magnitude in each layer and normalizing them to fall in the [-128, 127] range. The total memory footprint of the network is reduced by between 3× and 4×. This is very similar to the dynamic fixed point format we use (Section 5). However, Vanhoucke *et al.* (2011) only *apply* already trained neural networks while we actually *train* them.

Training neural networks with low precision arithmetic has already been done in previous works (Holt and Baker, 1991; Presley and Haggard, 1994; Simard and Graf, 1994; Wawrzynek *et al.*, 1996; Savich *et al.*, 2007) [2]. Our work is nevertheless original in several regards:

- We are the first to train deep neural networks with the dynamic fixed point format.
- We use a higher precision for the weights during the updates.
- We train some of the latest models on some of the latest benchmarks.

11 Conclusion and future works

We have shown that:

- Very low precision multipliers are sufficient for training deep neural networks.
- Dynamic fixed point seems particularly well suited for deep learning.
- Using a higher precision for the parameters during the updates helps.

Our work can be exploited to:

- Optimize memory usage on general-purpose hardware (Gray *et al.*, 2015).
- Design very power-efficient hardware dedicated to deep learning.

[2] A very recent work (Gupta *et al.*, 2015) also trains neural networks with low precision. The authors propose to replace round-to-nearest with stochastic rounding, which allows to reduce the numerical precision to 16 bits while using the fixed point format. It would be very interesting to combine dynamic fixed point and stochastic rounding.

There is plenty of room for extending our work:

- Other tasks than image classification.
- Other models than Maxout networks.
- Other formats than floating point, fixed point and dynamic fixed point.

12 Acknowledgement

We thank the developers of Theano (Bergstra *et al.*, 2010; Bastien *et al.*, 2012), a Python library which allowed us to easily develop a fast and optimized code for GPU. We also thank the developers of Pylearn2 (Goodfellow *et al.*, 2013b), a Python library built on the top of Theano which allowed us to easily interface the datasets with our Theano code. We are also grateful for funding from NSERC, the Canada Research Chairs, Compute Canada, and CIFAR.

References

Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I. J., Bergeron, A., Bouchard, N., and Bengio, Y. (2012). Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation.

Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., and Temam, O. (2014a). Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 269–284. ACM.

Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., *et al.* (2014b). Dadiannao: A machine-learning supercomputer. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 609–622. IEEE.

Coates, A., Baumstarck, P., Le, Q., and Ng, A. Y. (2009). Scalable learning for object detection with gpu hardware. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 4287–4293. IEEE.

David, J., Kalach, K., and Tittley, N. (2007). Hardware complexity of modular multiplication and exponentiation. *Computers, IEEE Transactions on*, **56**(10), 1308–1319.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Le, Q., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., and Ng, A. Y. (2012). Large scale distributed deep networks. In *NIPS'2012*.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *AISTATS'2011*.

Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. (2013a). Maxout networks. Technical Report Arxiv report 1302.4389, Université de Montréal.

Goodfellow, I. J., Warde-Farley, D., Lamblin, P., Dumoulin, V., Mirza, M., Pascanu,

R., Bergstra, J., Bastien, F., and Bengio, Y. (2013b). Pylearn2: a machine learning research library. *arXiv preprint arXiv:1308.4214*.

Gray, S., Leishman, S., and Koster, U. (2015). Nervanagpu library. Accessed: 2015-06-30.

Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. (2015). Deep learning with limited numerical precision. *CoRR*, **abs/1502.02551**.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. Technical report, arXiv:1207.0580.

Holt, J. L. and Baker, T. E. (1991). Back propagation simulations using limited precision calculations. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume 2, pages 121–126. IEEE.

Jarrett, K., Kavukcuoglu, K., Ranzato, M., and LeCun, Y. (2009). What is the best multi-stage architecture for object recognition? In *Proc. International Conference on Computer Vision (ICCV'09)*, pages 2146–2153. IEEE.

Kim, S. K., McAfee, L. C., McMahon, P. L., and Olukotun, K. (2009). A highly scalable restricted Boltzmann machine FPGA implementation. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 367–372. IEEE.

Krizhevsky, A. and Hinton, G. (2009). Learning multiple layers of features from tiny images. Technical report, University of Toronto.

Krizhevsky, A., Sutskever, I., and Hinton, G. (2012a). ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25 (NIPS'2012)*.

Krizhevsky, A., Sutskever, I., and Hinton, G. (2012b). ImageNet classification with deep convolutional neural networks. In *NIPS'2012*.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, **86**(11), 2278–2324.

Nair, V. and Hinton, G. (2010). Rectified linear units improve restricted Boltzmann machines. In *ICML'2010*.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. (2011). Reading digits in natural images with unsupervised feature learning. Deep Learning and Unsupervised Feature Learning Workshop, NIPS.

Pham, P.-H., Jelaca, D., Farabet, C., Martini, B., LeCun, Y., and Culurciello, E. (2012). Neuflow: dataflow vision processing system-on-a-chip. In *Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on*, pages 1044–1047. IEEE.

Presley, R. K. and Haggard, R. L. (1994). A fixed point implementation of the backpropagation learning algorithm. In *Southeastcon'94. Creative Technology Transfer-A Global Affair., Proceedings of the 1994 IEEE*, pages 136–138. IEEE.

Savich, A. W., Moussa, M., and Areibi, S. (2007). The impact of arithmetic represen-

tation on implementing mlp-bp on fpgas: A study. *Neural Networks, IEEE Transactions on*, **18**(1), 240–252.

Simard, P. and Graf, H. P. (1994). Backpropagation without multiplication. In *Advances in Neural Information Processing Systems*, pages 232–239.

Srebro, N. and Shraibman, A. (2005). Rank, trace-norm and max-norm. In *Proceedings of the 18th Annual Conference on Learning Theory*, pages 545–560. Springer-Verlag.

Vanhoucke, V., Senior, A., and Mao, M. Z. (2011). Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*.

Wawrzynek, J., Asanovic, K., Kingsbury, B., Johnson, D., Beck, J., and Morgan, N. (1996). Spert-ii: A vector microprocessor system. *Computer*, **29**(3), 79–86.

Williamson, D. (1991//). Dynamically scaled fixed point arithmetic. pages 315 – 18, New York, NY, USA. dynamic scaling;iteration stages;digital filters;overflow probability;fixed point arithmetic;fixed-point filter;.

Zeiler, M. D. and Fergus, R. (2013). Stochastic pooling for regularization of deep convolutional neural networks. In *International Conference on Learning Representations*.

ANNEXE B ARTICLE 2 : BINARYCONNECT : TRAINING DEEP NEURAL NETWORKS WITH BINARY WEIGHTS DURING PROPAGATIONS

Soumis le 5 Juin 2015 et accepté le 4 Septembre 2015 à la conférence NIPS 2015.

BinaryConnect: Training Deep Neural Networks with binary weights during propagations

Matthieu Courbariaux
École Polytechnique de Montréal
matthieu.courbariaux@polymtl.ca

Yoshua Bengio
Université de Montréal, CIFAR Senior Fellow
yoshua.bengio@gmail.com

Jean-Pierre David
École Polytechnique de Montréal
jean-pierre.david@polymtl.ca

Abstract

Deep Neural Networks (DNN) have achieved state-of-the-art results in a wide range of tasks, with the best results obtained with large training sets and large models. In the past, GPUs enabled these breakthroughs because of their greater computational speed. In the future, faster computation at both training and test time is likely to be crucial for further progress and for consumer applications on low-power devices. As a result, there is much interest in research and development of dedicated hardware for Deep Learning (DL). Binary weights, i.e., weights which are constrained to only two possible values (e.g. 0 or 1), would bring great benefits to specialized DL hardware by replacing many multiply-accumulate operations by simple accumulations, as multipliers are the most space and power-hungry components of the digital implementation of neural networks. We introduce BinaryConnect, a method which consists in training a DNN with binary weights during the forward and backward propagations, while retaining precision of the stored weights in which gradients are accumulated. Like other dropout schemes, we show that BinaryConnect acts as regularizer and we obtain near state-of-the-art results with BinaryConnect on the permutation-invariant MNIST. Lastly, we discuss in detail how BinaryConnect would be beneficial for DL hardware.

1 Introduction

Deep Neural Networks (DNN) have substantially pushed the state-of-the-art in a wide range of tasks, especially in speech recognition [1, 2] and computer vision, notably object recognition from images [3, 4]. More recently, deep learning is making important strides in natural language processing, especially statistical machine translation [5, 6, 7]. Interestingly, one of the key factors that enabled this major progress has been the advent of Graphics Processing Units (GPUs), with speed-ups on the order to 10 to 30-fold, starting with [8], and similar improvements with distributed training [9, 10]. Indeed, the ability to train larger models on more data has enabled the kind of breakthroughs observed in the last few years. Today, researchers and developers designing new deep learning algorithms and applications often find themselves limited by computational capability. This along, with the drive to put deep learning systems on low-power devices (unlike GPUs) is greatly increasing the interest in research and development of specialized hardware for deep networks [11, 12, 13].

Most of the computation performed during training and application of deep networks regards the multiplication of a real-valued weight by a real-valued activation (in the recognition or forward propagation phase of the back-propagation algorithm) or gradient (in the backward propagation phase of the back-propagation algorithm). This paper proposes an approach called BinaryConnect

to eliminate the need for these multiplications by forcing the weights used in these forward and backward propagations to be binary, i.e. constrained to only two values (not necessarily 0 and 1). We show that state-of-the-art results can be achieved with BinaryConnect on a deep fully-connected network on the MNIST benchmark.

What makes this workable are two ingredients:

1. Sufficient precision is necessary to accumulate and average a large number of stochastic gradients, but noisy weights (and we can view discretization into a small number of values as a form of noise, especially if we make this discretization stochastic) are quite compatible with Stochastic Gradient Descent (SGD), the main type of optimization algorithm for deep learning. SGD explores the space of parameters by making small and noisy steps and that noise is *averaged out* by the stochastic gradient contributions accumulated in each weight. Therefore, it is important to keep sufficient resolution for these accumulators, which at first sight suggests that high precision is absolutely required. [14] and [15] show that randomized or stochastic rounding can be used to provide unbiased discretization. [14] have shown that SGD requires weights with a precision of at least 6 to 8 bits and [16] successfully train DNNs with 12 bits dynamic fixed-point computation. Besides, the estimated precision of the brain synapses varies between 6 and 12 bits [17]
2. Noisy weights actually provide a form of regularization which can help to generalize better, as previously shown with variational weight noise [18], Dropout [19, 20] and DropConnect [21], which add noise to the activations or to the weights. For instance, DropConnect [21], which is closest to BinaryConnect, is a very efficient regularizer that randomly substitutes half of the weights with zeros during propagations. What these previous works show is that *only the expected value of the weight needs to have high precision*, and that noise can actually be beneficial.

The main contributions of this article are the following.

- We introduce BinaryConnect, a method which consists in training a DNN with binary weights during the forward and backward propagations (Section 2).
- We show that BinaryConnect is a regularizer and we obtain near state-of-the-art results on the permutation-invariant MNIST (Section 3).
- We discuss how BinaryConnect would bring benefits to specialized hardware for deep learning (Section 4).
- We make the code for BinaryConnect available [1].

2 BinaryConnect

In this section we give a more detailed view of BinaryConnect, considering which values to choose, how to discretize, and how to perform inference.

2.1 Propagations vs updates

Let us consider the different steps of back-propagation with SGD udpates and whether it makes sense, or not, to discretize the weights, at each of these steps.

1. Given the DNN input, compute the unit activations layer by layer, leading to the top layer which is the output of the DNN, given its input. This step is referred as the forward propagation.
2. Given the DNN target, compute the training objective's gradient w.r.t. each layer's activations, starting from the top layer and going down layer by layer until the first hidden layer. Compute the gradient w.r.t. each layer's parameters along the way. This step is referred to as the backward propagation or backward phase of back-propagation.
3. Update the parameters, using their computed gradients and their previous values.

[1] `https://github.com/AnonymousWombat/BinaryConnect`

Algorithm 1 SGD training with BinaryConnect. $\mathcal{L}$ is the loss function for minibatch, f maps the input and the parameters the network's output. The functions binarize(W) and clip(W) specify how to binarize and clip weights.

Require: a minibatch of (input, target) examples (x, y), previous parameters w_{t-1} (weights) and b_{t-1} (biases) collectively for all the layers, and learning rate η.
Ensure: updated parameters w_t and b_t.
Forward propagation:
$w_b \leftarrow \text{binarize}(w_{t-1})$
$\hat{y} \leftarrow f(x, w_b, b_{t-1})$
$L \leftarrow \mathcal{L}(\hat{y}, y)$
Backward propagation:
Compute $\frac{\partial L}{\partial w_b}$ by back-propagation
Compute $\frac{\partial L}{db_{t-1}}$ along the way
Parameter update:
$w_t \leftarrow \text{clip}(w_{t-1} - \eta \frac{\partial L}{\partial w_b})$
$b_t \leftarrow b_{t-1} - \eta \frac{\partial L}{\partial b_{t-1}}$

A key point to understand with BinaryConnect is that we only binarize the weights during the forward and backward propagations but not during the updates, as illustrated in Algorithm 1. Keeping good precision weights during the updates is necessary for SGD to work at all. These parameter changes are tiny by virtue of being obtained by gradient descent, i.e., SGD performs a large number of almost infinitesimal changes in the direction that most improves the training objective (plus noise). One way to picture all this is to hypothesize that what matters most at the end of training is the sign of the weights, w^*, but that in order to figure it out, we perform a lot of small changes to a continuous-valued quantity w, and only at the end consider its sign:

$$w^* = \text{sign}(\sum_t g_t) \tag{1}$$

where g_t is a noisy estimator of $\frac{\partial \mathcal{L}(f(x_t, w_{t-1}, b_{t-1}), y_t)}{\partial w_{t-1}}$, where $\mathcal{L}(f(x_t, w_{t-1}, b_{t-1}), y_t)$ is the value of the objective function on (input,target) example (x_t, y_t), when w_{t-1} are the previous weights and w^* is its final discretized value of the weights.

Another way to conceive of this discretization is as a form of corruption, and hence as a regularizer, and our empirical results confirm this hypothesis. In addition, *we can make the discretization errors on different weights approximately cancel each other while keeping a lot of precision by randomizing the discretization appropriately.* We propose a form of randomized discretization that *preserves the expected value of the discretized weight.*

Hence, at training time, BinaryConnect randomly picks one of two values for each weight, for each minibatch, for both the forward and backward propagation phases of backprop. However, the SGD update is accumulated in a real-valued variable storing the parameter.

An interesting analogy to understand BinaryConnect is the DropConnect algorithm [21]. Just like BinaryConnect, DropConnect only injects noise to the weights during the propagations. Whereas DropConnect's noise is added Gaussian noise, BinaryConnect's noise is a binary sampling process. In both cases the corrupted value has as expected value the clean original value.

2.2 Two values

Binarizing the weights means restraining them to only two possible values. The question is: which two values should we choose? In this article, we choose to use the weights initialization scaling coefficients from [22]:

$$H = \frac{1}{2} \times \sqrt{\frac{6}{fan_{in} + fan_{out}}} \tag{2}$$

$$L = -H \tag{3}$$

Where H and L are the two values, fan_{in} is the number of inputs of the layer and fan_{out} the number of outputs. Those two values work surprisingly well. Besides that, we use Batch Normalization (BN) [23] in most of our experiments. An interesting property of BN is that it reduces the overall impact of the parameters scale. Note however that in our experiments, BinaryConnect worked quite well even without BN.

2.3 Deterministic vs stochastic binarization

The binarization operation transforms the real-valued weights into two possible values. A very straightforward binarization operation would be based on the sign function:

$$w_b = \begin{cases} +H & \text{if } w \geq 0, \\ -H & \text{otherwise.} \end{cases} \tag{4}$$

Where w_b is the binarized weight and w the real-valued weight. Although this is a deterministic operation, averaging this discretization over the many input weights of a hidden unit could compensate for the loss of information. An alternative that allows a finer and more correct averaging process to take place is to binarize stochastically:

$$w_b = \begin{cases} +H & \text{with probability } p = \sigma(\frac{w}{H}), \\ -H & \text{with probability } 1 - p. \end{cases} \tag{5}$$

where σ is the *"hard sigmoid"* function:

$$\sigma(x) = \text{clip}(\frac{x+1}{2}, 0, 1) = \max(0, \min(1, \frac{x+1}{2})) \tag{6}$$

We use such a hard sigmoid rather than the soft version because it is far less computationally expensive (both in software and specialized hardware implementations) and yielded excellent results in our experiments. It is similar to the "hard tanh" non-linearity introduced by [24]. It is also piece-wise linear and corresponds to a bounded form of the rectifier [25].

2.4 Clipping

Since the binarization operation is not influenced by variations of the real-valued weights w when its magnitude is beyond the binary values $\pm H$, and since it is a common practice to bound weights (usually the weight vector) in order to regularize them, we have chosen to clip the real-valued weights within the $(-H, H)$ interval right after the weight updates, as per Algorithm 1, i.e.,

$$w_t \leftarrow clip(w_{t-1} - \text{gradient}, -H, +H) \tag{7}$$

The real-valued weights would otherwise grow very large without any impact on the binary weights.

2.5 Test-Time Inference

Up to now we have introduced different ways of *training* a DNN with on-the-fly weight binarization. What are reasonable ways of using such a trained network, i.e., performing test-time inference on new examples? We have considered three reasonable alternatives:

1. Use the resulting binary weights w_b (this makes most sense with the deterministic form of BinaryConnect).
2. Use the real-valued weights w, i.e., the binarization only helps to achieve faster training but not faster test-time performance.
3. In the stochastic case, many different networks can be sampled by sampling a w_b for each weight according to Eq. 5. The ensemble output of these networks can then be obtained by averaging the outputs from individual networks.

We use the first method with the deterministic form of BinaryConnect. As for the stochastic form of BinaryConnect, we focused on the training advantage and used the second method in the experiments, i.e., test-time inference using the real-valued weights. This follows the practice of Dropout methods, where at test-time the "noise" is removed.

Method	Validation error rate (%)	Test error rate (%)
No regularizer	1.21 ± 0.04	1.30 ± 0.04
BinaryConnect (det.)	1.17 ± 0.02	1.29 ± 0.08
BinaryConnect (stoch.)	1.12 ± 0.03	1.18 ± 0.04
50% Dropout	0.94 ± 0.04	1.01 ± 0.04
Maxout networks [26]		0.94
Deep L2-SVM [27]		0.87

Table 1: Error rates of a MLP trained on MNIST depending on the method. Methods using unsupervised pretraining are not taken into account. We see that in spite of using only a single bit per weight during propagation, performance is not worse than an ordinary (no regularizer) MLP, it is actually better, especially with the stochastic version, suggesting that BinaryConnect acts as a regularizer.

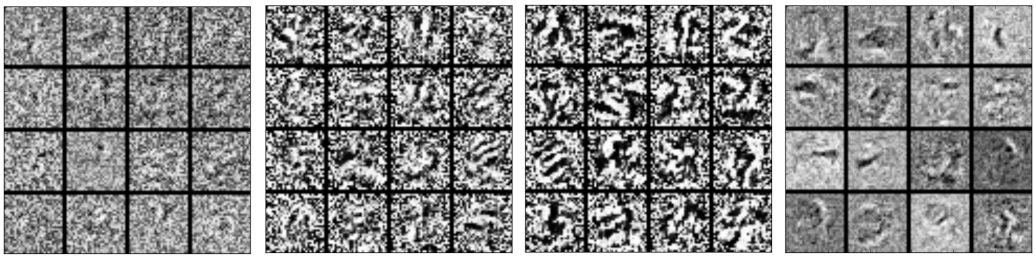

Figure 1: Features of the first layer of a MLP trained on MNIST depending on the regularizer. From left to right: no regularizer, deterministic BinaryConnect, stochastic BinaryConnect and Dropout.

3 Permutation-invariant MNIST

We show that BinaryConnect acts as regularizer and we obtain near state-of-the-art results with BinaryConnect on the permutation-invariant MNIST. MNIST is a benchmark image classification dataset [28]. It consists in a training set of 60000 and a test set of 10000 28×28 gray-scale images representing digits ranging from 0 to 9. Permutation-invariance means that the model must be unaware of the structure of the data (in other words, CNN are forbidden). Besides, we do not use any data-augmentation, preprocessing or unsupervised pretraining.

The MLP we train on the MNIST consists in 3 hidden layers of 1024 Rectifier Linear Units (ReLU) [29, 25, 3] and a L2-SVM output layer. L2-SVMs have been shown to perform better than Softmax on several classification benchmarks [27, 30]. The square hinge loss is minimized with SGD without momentum. We use an exponentially decaying learning rate. We use Batch Normalization with a minibatch of size 200 to speed up the training. As typically done, we use the last 10000 samples of the training set as a validation set for early stopping and model selection. We report the test error rate associated with the best validation error rate after 1000 epochs (we do not retrain on the validation set). We repeat each experiment 6 times with different initializations. The results are in Table 1. They suggest that the stochastic version of BinaryConnect can be considered a regularizer, although a less powerful one than Dropout, in this context.

4 Computer hardware

Computer hardware, be it general-purpose or dedicated, is made out of arithmetic operators and memories. In this section, we show that BinaryConnect could be of great benefit to DL dedicated hardware by replacing many multiply-accumulate operations by simple accumulations. We also show that BinaryConnect would likely make FPGAs faster (and much more power efficient) than GPUs for training DNNs.

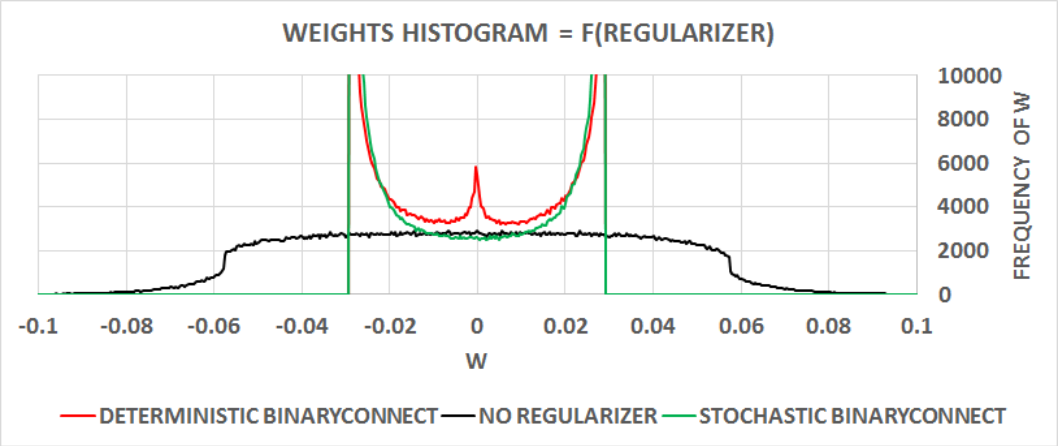

Figure 2: Histogram of the weights of the first layer of a MLP trained on MNIST depending on the regularizer. We can see that the weights of stochastic BinaryConnect are trying to become deterministic to reduce the training error. We can also see that some of the weights of deterministic BinaryConnect are stuck around 0, hesitating between $-H$ and H.

Implementation	Maximum processing rate
FPGA (VU440) without BinaryConnect	1.9 TMAC/s
FPGA (VU440) training with BinaryConnect	5.3 TMAC/s
FPGA (VU440) testing with deterministic BinaryConnect	45 TMAC/s
GPU (TitanX)	3.3 TMAC/s (=6.6 TFLOP/s)

Table 2: Maximum processing rate depending on the implementation. BinaryConnect would likely make FPGAs faster than GPUs for training and testing DNNs.

4.1 Multiply-accumulators

Applying a DNN mainly consists in convolutions and matrix multiplications. The key arithmetic operation of DNNs is thus the multiply-accumulate operation. Artificial neurons are basically multiply-accumulators computing weighted sums of their inputs.

With BinaryConnect, it is possible to replace many multiply-accumulate operations by simple accumulations (explanations follow). Instead of constraining the weights to either $-H$ or $+H$, it is possible to constrain them to either -1 or $+1$ and scale the inputs of the layer by H instead. The multiply-accumulate operations are thus replaced by simple additions. The scaling of the inputs requires far fewer multiplications ($O(n)$) than the multiply-accumulations would ($O(n^2)$). In other words, it is negligible.

This is a huge gain. Adders are much cheaper than multiply-accumulators as they require only a small fraction of the memory bandwidth and also as additions are much less complex than multiplications both in terms of hardware real estate and in energy expense. In other words, the cost of adders is negligible beside the cost of multiply-accumulators.

Sadly, some multiply-accumulate operations are still required for training a DNN with BinaryConnect during the backpropagation of the parameters. Those multiply-accumulate operations amounts to about a third of the total additions required to train a DNN. Assuming additions are negligible beside multiply-accumulate operations, we would still get about a $3\times$ speed-up during the training of a DNN on dedicated hardware.

4.2 Estimated gains on a modern FPGA

FPGA are composed of configurable Logic Cells (LC, typically small lookup tables), memory blocks, arithmetic blocks (typically small fixed point multiply-accumulate or MAC blocks) and dedicated blocks such as memory interfaces, communication IP and others. A configurable interconnect network enables the designer to generate any kind of circuit and tailor the hardware to a dedicated application. The size of FPGAs is increasing at a rapid rate. In January 2015, the manufacturer Xilinx announced the delivery of the Virtex UltraScale VU440 FPGA. Said FPGA contains more than 4.4 million LCs, 2520 Block RAM (BR) and 2880 DSP Slices (DSPS). Each BR is a 36 kbits memory that can be accessed through 72-bit words at a rate of up to 660MHz. Basically, each DSPS is a 27x18 fixed point signed multiplier followed by a 48-bit accumulator. At 660Mhz, the maximum computing power is 2880 MAC$\times$660Mhz = 1.9 TMAC/s. Nevertheless, this computing power is meaningful only if the system can feed all the operators with data at each clock cycle. In the context of neural networks, this typically means that all the weights are in the local memory (the BRs). For 16-bit weights, the VU440 can locally store up to $\frac{2520\times36\times1024}{16} = 5.8 \times 10^6$ weights, from which only $\frac{2520\times72}{16} = 11340$ will be available for processing at each clock cycle. This is not enough for modern deep learning.

Single bit weights have an impact both on the total number of weights that can be stored inside the FPGA and on the number of processing resources that are available. In this context, the VU440 FPGA can store $2520 \times 36 \times 1024 = 93 \times 10^6$ weights which is on par with state-of-the-art models on ImageNet, for example. Nevertheless, the number of weights that are actually available for processing at a given clock cycle is $2520 \times 72 = 181,440$. As mentioned in the previous section, for single bit weights, a MAC resource is reduced to an adder/subtracter, followed by a register (the accumulator). In the mentioned FPGA, such MACs only requires 32 LCs for a 32-bit accumulator if we do not use the dedicated DSPS. The maximum number of MAC resources available in the VU440 FPGA is thus theoretically $\frac{4.4\times10^6}{32} = 137500$. However, some logic resources must be kept to implement the storage of the inputs/outputs, the routing and the control of the system. If we use half the resources for the actual processing, we obtain a maximum processing rate of $\frac{137500}{2}\times660Mhz = 45$ TMAC/s.

Such impressive potential performances are theoretical peak performances because they do not take into account the time to get new inputs and to send the results out. Moreover, the actual clock frequency may not reach 660Mhz, depending on the complexity of the architecture. Nevertheless, it is reasonable to estimate that a pipelined architecture designed with great care may reach a peak processing rate in the order of 30-40 TMAC/s on such an FPGA for single bit weights in computing mode. In training mode, about a third of the accumulations require multiplications, we thus obtain a maximum processing rate of about 5.3 TMAC/s. Table 2 recaps the results of this subsection.

5 Related work

Training DNNs with binary weights has been the subject of very recent works [31, 32]. Even though we share the same objective, our approaches are very different. They do not train their DNN with Backpropagation (BP) but with a variant called Expectation Backpropagation (EBP). EBP is based on Expectation Propagation (EP) [33], which is a variational Bayes method used to do inference in probabilistic graphical models. Let us compared their method to the one presented here:

- It optimizes the weights posterior distribution (which is not binary). In this regard, our method is quite similar as we keep a real-valued version of the weights.
- It requires to average the outputs of several DNNs with weights sampled from the posterior distribution. This is not the case with versions 1 and 2 of BinaryConnect (use the binary or real-valued weights at test-time).
- It seems to yield worse classification accuracy than full-precision ordinary DNNs (which should be expected, in general). However, the comparisons were made on different datasets, so care must be taken in concluding hastily.

[34] binarize the weights of already trained DNNs at the cost of a loss in accuracy. The main difference with our work is that we actually train our DNNs with binary weights, which has two advantages. Firstly, BinaryConnect can be used to speed up the training on dedicated hardware.

Secondly, with deterministic BinaryConnect, each weight is optimized in the context where the other weights are binary. The weights we use at inference time are the binary weights and doing otherwise would actually result in a loss of accuracy.

6 Conclusion

We have introduced a novel binarization scheme for weights during propagations called BinaryConnect. We have shown that it is possible to train a fully-connected DNN on the MNIST dataset with BinaryConnect and achieve nearly state-of-the-art results when considering the same information (permutation invariance MNIST). The impact of such a method on specialized hardware implementations of deep networks could be major, by removing the need for 2/3 of the multiplications, and thus potentially allowing to speed-up by a factor of 3 at training time. With the deterministic version of BinaryConnect the impact at test time could be even more important, getting rid of the multiplications altogether and reducing by a factor of at least 16 (from 16 bits single-float precision to single bit precision) the memory requirement of deep networks, which has an impact of the memory to computation bandwidth and on the size of the models that can be run. Future works should extend those results to other models and datasets, and explore getting rid of the multiplications altogether during training, by removing their need from the weight update computation.

References

[1] Geoffrey Hinton, Li Deng, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 29(6):82–97, Nov. 2012.

[2] Tara Sainath, Abdel rahman Mohamed, Brian Kingsbury, and Bhuvana Ramabhadran. Deep convolutional neural networks for LVCSR. In *ICASSP 2013*, 2013.

[3] A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet classification with deep convolutional neural networks. In *NIPS'2012*. 2012.

[4] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. Technical report, arXiv:1409.4842, 2014.

[5] Jacob Devlin, Rabih Zbib, Zhongqiang Huang, Thomas Lamar, Richard Schwartz, and John Makhoul. Fast and robust neural network joint models for statistical machine translation. In *Proc. ACL'2014*, 2014.

[6] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *NIPS'2014*, 2014.

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR'2015, arXiv:1409.0473*, 2015.

[8] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. Large-scale deep unsupervised learning using graphics processors. In *ICML'2009*, 2009.

[9] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003.

[10] J. Dean, G.S Corrado, R. Monga, K. Chen, M. Devin, Q.V. Le, M.Z. Mao, M.A. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS'2012*, 2012.

[11] Sang Kyun Kim, Lawrence C McAfee, Peter Leonard McMahon, and Kunle Olukotun. A highly scalable restricted Boltzmann machine FPGA implementation. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 367–372. IEEE, 2009.

[12] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 269–284. ACM, 2014.

[13] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 609–622. IEEE, 2014.

[14] Lorenz K Muller and Giacomo Indiveri. Rounding methods for neural networks with low resolution synaptic weights. *arXiv preprint arXiv:1504.05767*, 2015.

[15] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *ICML'2015*, 2015.

[16] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Low precision arithmetic for deep learning. In *Arxiv:1412.7024, ICLR'2015 Workshop*, 2015.

[17] Thomas M Bartol, Cailey Bromer, Justin P Kinney, Michael A Chirillo, Jennifer N Bourne, Kristen M Harris, and Terrence J Sejnowski. Hippocampal spine head sizes are highly precise. *bioRxiv*, 2015.

[18] Alex Graves. Practical variational inference for neural networks. In J. Shawe-Taylor, R.S. Zemel, P.L. Bartlett, F. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2348–2356. Curran Associates, Inc., 2011.

[19] Nitish Srivastava. Improving neural networks with dropout. Master's thesis, U. Toronto, 2013.

[20] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[21] Li Wan, Matthew Zeiler, Sixin Zhang, Yann LeCun, and Rob Fergus. Regularization of neural networks using dropconnect. In *ICML'2013*, 2013.

[22] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS'2010*, 2010.

[23] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 2015.

[24] R. Collobert. *Large Scale Machine Learning*. PhD thesis, Université de Paris VI, LIP6, 2004.

[25] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *AISTATS'2011*, 2011.

[26] Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. Technical Report Arxiv report 1302.4389, Université de Montréal, February 2013.

[27] Yichuan Tang. Deep learning using linear support vector machines. Workshop on Challenges in Representation Learning, ICML, 2013.

[28] Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[29] V. Nair and G.E. Hinton. Rectified linear units improve restricted Boltzmann machines. In *ICML'2010*, 2010.

[30] Chen-Yu Lee, Saining Xie, Patrick Gallagher, Zhengyou Zhang, and Zhuowen Tu. Deeply-supervised nets. *arXiv preprint arXiv:1409.5185*, 2014.

[31] Daniel Soudry, Itay Hubara, and Ron Meir. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In *NIPS'2014*, 2014.

[32] Zhiyong Cheng, Daniel Soudry, Zexi Mao, and Zhenzhong Lan. Training binary multilayer neural networks for image classification using expectation backpropgation. *arXiv preprint arXiv:1503.03562*, 2015.

[33] Thomas P Minka. Expectation propagation for approximate bayesian inference. In *UAI'2001*, 2001.

[34] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.