# Neural network quantization methods for FPGA on-board processing of satellite images

Cédric Gernigon

▶ **To cite this version:**

Cédric Gernigon. Neural network quantization methods for FPGA on-board processing of satellite images. Neural and Evolutionary Computing [cs.NE]. Université de Rennes, 2024. English. ⟨NNT : 2024URENS122⟩. ⟨tel-05216300⟩

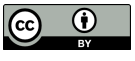
**HAL Id: tel-05216300**

**https://theses.hal.science/tel-05216300v1**

Submitted on 20 Aug 2025

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

COLLEGE MATHS, TELECOMS
DOCTORAL INFORMATIQUE, SIGNAL
BRETAGNE SYSTEMES, ELECTRONIQUE

Université de Rennes

# THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

ÉCOLE DOCTORALE N° 601
*Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique*
Spécialité : *Informatique*

Par
**Cédric GERNIGON**

# Neural Network Quantization Methods for FPGA On-Board Processing of Satellite Images

**Thèse présentée et soutenue à Rennes, le mercredi 18 décembre 2024**
**Unité de recherche : INRIA - CNES**

## Rapporteurs avant soutenance :

Florent de DINECHIN — Professor, INSA Lyon
David NOVO — Researcher, HDR, CNRS, LIRMM, Montpellier

## Composition du Jury :

| | | |
|---|---|---|
| Président : | Daniel MENARD | Professor, INSA Rennes |
| Examinateurs : | Florent de DINECHIN | Professor, INSA Lyon |
| | David NOVO | Researcher, HDR, CNRS, LIRMM, Montpellier |
| | Emanuel POPOVICI | Senior Lecturer, UCC, Cork, Ireland |
| | Tania POULI | Deputy Director of Strategy, b<>com, Rennes |
| | Alexandre TERMIER | Professor, University of Rennes |
| Dir. de thèse : | Olivier SENTIEYS | Professor, University of Rennes, INRIA |
| Co-enc. de thèse : | Silviu-Ioan FILIP | Researcher, INRIA, Rennes |

## Invité :

Clément COGGIOLA — Research Engineer, CNES, Toulouse (co-encadrant)

# Résumé Substantiel

## Contexte

Au cours des dernières années, l'*Intelligence Artificielle* (IA) a suscité un grand intérêt dans la recherche scientifique et l'industrie, notamment dans le domaine du traitement d'images. L'IA englobe tous les algorithmes et méthodes utilisés pour développer des systèmes capables de simuler l'intelligence. Traditionnellement, les systèmes d'IA sont basés sur des règles conçues pour imiter l'« intelligence » en utilisant des modèles fondés uniquement sur des règles prédéterminées. Ces systèmes reposent sur un ensemble de règles programmées manuellement qui conduisent à des résultats prédéfinis. À l'inverse, les algorithmes d'*Apprentissage Automatique* apprennent et s'adaptent sans suivre d'instructions explicites, en analysant et en déduisant des résultats à partir de motifs dans les données. Aujourd'hui, l'IA repose en grande partie sur l'apprentissage automatique et, plus précisément, sur l'un de ses sous-domaines, l'*Apprentissage Profond*. Ce sous-domaine vise à reproduire le fonctionnement du cerveau biologique en utilisant des modélisations mathématiques de neurones organisés en réseaux. Les premiers travaux sur les réseaux de neurones ont introduit des modèles composés de quelques couches qui ont rapidement montré leurs limites. Au cours des dernières décennies, plusieurs avancées ont permis d'entraîner efficacement des réseaux de neurones composés d'un grand nombre de couches, levant ainsi les limitations des premiers modèles et posant les bases de ce que l'on appelle aujourd'hui les *Réseaux de Neurones Profonds*. Jusqu'à la dernière décennie, l'augmentation du nombre de couches était principalement limitée par la nécessité de disposer de quantités exponentielles de données d'entraînement pour garantir la capacité de généralisation des réseaux de neurones. Parallèlement, l'utilisation de couches de convolution a conduit au succès des *Réseaux de Neurones Convolutifs* pour le traitement d'images [LeC+89]. Cependant, leur véritable percée est venue plus tard avec les résultats remarquables, pour l'époque, d'AlexNet [KSH12] sur le célèbre jeu de données ImageNet [Den+09]. Ces différents travaux ont posé les fondements sur lesquels les réseaux de neurones profonds se basent encore aujourd'hui. Leur gain en popularité, lui, s'explique surtout grâce à trois facteurs : la disponibilité de grandes quantités de données, l'augmentation de la puissance de

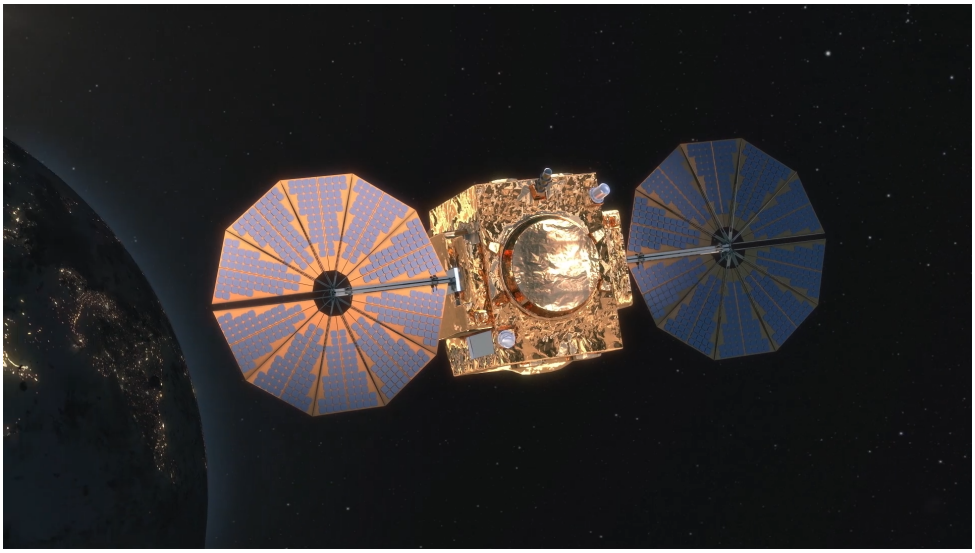


Figure 1: Satellite Pléiades Neo. Crédit: ©Airbus 2022

calcul, notamment avec les processeurs graphiques (GPU), et les avancées algorithmiques.

Dans le secteur spatial, l'apprentissage profond commence à être utilisé avec succès dans diverses applications, telles que la conduite autonome de rovers [Swa+21], la détection de débris spatiaux pour prévenir les collisions [AL20a], ou encore la détection de corps célestes [LS23]. Notre partenaire pour cette thèse, le Centre National d'Etudes Spatiales (CNES), travaille également sur des applications d'apprentissage profond liées aux données spatiales, notamment à l'aide de réseaux de neurones convolutifs pour le traitement d'images satellites d'observation de la Terre (OT). L'OT permet de relever efficacement les informations physiques, chimiques et biologiques relatives à la Terre, y compris des variables climatiques essentielles, définie par le Système Mondial d'Observation du Climat (SMOC)[1], qui contribuent à la caractérisation du climat terrestre et à l'étude du changement climatique. Les données collectées par les satellites d'OT sont également largement utilisées dans différents domaines de recherche et applications, avec des exemples notables comme la surveillance maritime [SZZ20], la détection et la surveillance de catastrophes naturelles (tel que les feux de forêt [Kan+22]) ou la détection de nuages pour les événements météorologiques [Jep+19]), et la mesure du niveau des eaux des rivières [Nin+21]. Dans les différents cas cités, les mesures effectuées par les satellites d'OT sont souvent

[1] https://gcos.wmo.int/en/essential-climate-variables

indispensables. L'OT nécessite l'extraction d'informations géographiques (par exemple, l'utilisation des sols, la répartition de la biomasse, etc.) à partir des données acquises par les capteurs embarqués. Cela repose souvent sur une étape de cartographie visant à produire automatiquement une carte sémantique contenant diverses régions d'intérêt basées sur des données optiques, un domaine largement étudié en apprentissage profond : la *segmentation sémantique*.

Depuis le lancement du premier satellite en 1957, qui a marqué le début de la course à l'espace et conduit à la création des premiers satellites d'OT dans les années 60, le nombre de satellites d'OT en orbite a considérablement augmenté, dépassant le millier en quelques décennies. Au fil des ans, les capacités des satellites d'imagerie n'ont cessé de progresser tout autant que leur nombre. Les capteurs optiques ont vu leur résolution s'améliorer, et les constellations de satellites ont réduit le temps de revisite. En conséquence, le volume de données générées par les satellites d'OT est devenu difficile à appréhender. En 2008, on enregistrait déjà plus de 10 téraoctets de données par jour [TGH08]. Aujourd'hui, la constellation Pléiades Neo (Figure 1) produit 40 téraoctets de données quotidiennement [Jér19]. À mesure que les coûts de lancement diminuent et que le nombre de satellites augmente, ces quantités énormes de données devraient continuer à croître. La transmission de toutes ces données est rendu possible grâce à la communication entre les satellites et les stations au sol. Cependant, les systèmes d'OT sont limités par ces communications de liaison descendante en raison de contraintes matérielles, de limitations de puissance à bord ou de coûts d'exploitation des stations au sol. Il est donc nécessaire de réduire la quantité de données transmises par la liaison descendante. Le traitement d'images à bord pour améliorer l'efficacité des données n'est pas nouveau. La compression a été étudiée dès les débuts de l'imagerie satellite [RW73]. Aujourd'hui, la plupart des systèmes satellites intègrent du matériel dédié à la compression d'images pour réduire le volume de données, comme l'algorithme CCSDS-123.0-B-2 [Spa19], qui exploite la forte corrélation inter-canaux dans les images satellites. Toutefois, la transmission uniquement des données pertinentes grâce au traitement à bord n'a commencé à susciter de l'intérêt que très récemment. L'industrie spatiale et notre partenaire, le CNES, se sont récemment intéressés au déploiement de l'apprentissage profond à bord des satellites pour la réduction des données ou les systèmes autonomes. Bien que les réseaux de neurones profonds aient la capacité d'extraire des informations sémantiques, l'inférence de nombreux modèles est encore principalement réalisée sur des plateformes au sol en raison de leur

empreinte mémoire et de leur intensité de calcul [Fur+20].

Les ressources nécessaires pour entraîner les modèles de réseaux de neurones profonds exigent d'énormes quantités de puissance de calcul, disponibles uniquement sur des serveurs. Bien que de nombreux calculs d'inférence soient également réalisés sur des serveurs, il devient de plus en plus souhaitable de déployer des réseaux de neurones profonds sur des appareils périphériques, tels que les smartphones et les appareils connectés, en raison de préoccupations liées à la confidentialité, à la sécurité, à la latence ou aux limitations de bande passante. Les réseaux de neurones profonds modernes contiennent des millions de paramètres et nécessitent des milliards d'opérations arithmétiques, rendant leur déploiement sur des appareils embarqués difficile, voire impossible, dans de nombreux cas. Pour atténuer ces problèmes, diverses techniques de compression ont été proposées, telles que l'élagage [HMD15], le partage de poids [Dup+20], la distillation de connaissances [TM19], et la quantification [Hub+17]. La quantification, qui consiste à approximer les paramètres des réseaux de neurones avec une largeur de bits réduite, est une technique populaire pour réduire l'empreinte mémoire d'un modèle et la complexité de ses calculs tout en maintenant une qualité de prédictions proche du modèle de référence en virgule flottante. De plus, l'émergence de plateformes matérielles offrant un meilleur support pour les calculs à faible précision (par exemple, les dernières générations de GPU de Nvidia et TPU de Google) et à précision personnalisée (par exemple, les FPGA et ASIC) place la quantification au premier plan des méthodes utilisées pour améliorer l'efficacité de l'inférence des modèles de réseaux de neurones profonds.

Cette thèse se concentre sur la quantification des réseaux de neurones convolutifs pour faciliter le déploiement de modèles de segmentation sémantique embarqués à bord de satellites. En particulier, nous avons étudié l'impact de la quantification sur les tâches de détection de navires à partir d'images satellites. Ce document est structuré en deux parties : Contexte et Contributions, chacune comprenant trois chapitres. La première partie présente les informations générales nécessaires à la compréhention de nos travaux et la seconde partie introduit nos différentes contributions.

## Chapitre 1 : Réseaux neuronaux profonds pour la vision par ordinateur

Les fondements mathématiques de l'apprentissage automatique reposent sur des modèles statistiques et des algorithmes capables d'apprendre à partir de données et de généraliser à de nouvelles données. Dans ce chapitre, nous nous concentrons sur l'évolution des méthodes d'apprentissage automatique basées sur les réseaux de neurones. L'étude des réseaux de neurones artificiels trouve son origine dans l'ambition des scientifiques de créer un système informatique simulant le cerveau humain. Pour construire un tel système, il est nécessaire de comprendre comment fonctionne notre système cognitif. Les premiers travaux sur les réseaux de neurones artificiels ont introduit des modèles simples, souvent constitués d'une seule couche. La profondeur de ces premiers modèles était principalement limitée par les méthodes d'entraînement de l'époque. Au cours des dernières décennies, une série de développements a permis d'entraîner efficacement des réseaux de neurones avec de nombreuses couches, levant ainsi les limitations des premiers modèles et posant les bases des réseaux de neurones profonds.

Le chapitre 1 introduit les concepts de base de l'apprentissage profond afin de faciliter la compréhension des travaux présentés dans cette thèse. Nous retracerons brièvement l'histoire de l'apprentissage profond en abordant certaines des avancées majeures. Ensuite, nous décrirons en détail l'architecture des modèles de réseaux de neurones profonds, en commençant par une explication des perceptrons et de la manière dont ils s'assemblent pour former des réseaux de neurones. Puis, nous expliquerons les mécanismes d'entraînement des réseaux de neurones. Enfin, nous aborderons deux tâches courantes en vision par ordinateur, la *classification* et la *segmentation*, ainsi que les modèles modernes performants pour ces tâches.

## Chapitre 2 : Quantification des réseaux de neurones profonds pour l'accélération de l'inférence

L'entraînement d'un modèle de réseau de neurones profonds nécessite une grande quantité de données et des ressources de calcul importantes. Cette phase peut durer plusieurs jours, voir semaines, ce qui explique pourquoi elle est généralement réalisée sur des serveurs. L'inférence, bien que principalement effectuée sur des serveurs, peut également être exécutée sur des appareils périphériques en raison de préoccupations liées à la confidentialité,

à la sécurité, à la latence et aux limitations de bande passante. Cependant, les réseaux de neurones profonds modernes peuvent contenir plusieurs millions de paramètres et nécessitent des milliards d'opérations arithmétiques. Dans de nombreux cas, les coûts en mémoire et en calcul rendent le déploiement sur des appareils périphériques difficile, voire impossible. Pour pallier ces problèmes, les recherches récentes se sont concentrées sur la compression des réseaux de neurones. Diverses techniques, telles que l'élagage, le partage de poids, la distillation et la quantification, peuvent être utilisées pour réduire l'intensité des calculs et les rendre compatibles avec une exécution embarquée. Avec l'émergence de plateformes matérielles offrant un meilleur support pour les calculs à faible précision, en particulier, les dernières générations de GPU Nvidia, et à précision personnalisée, la quantification est au premier plan des méthodes utilisées pour améliorer l'efficacité de l'inférence [Dup+22].

Le chapitre 2 introduit les concepts de base de la quantification des réseaux de neurones afin de faciliter la compréhension des travaux réalisés dans cette thèse. Tout d'abord, nous passerons brièvement en revue diverses méthodes de compression des réseaux de neurones, en mettant l'accent sur la quantification. Ensuite, nous décrierons plusieurs formats de quantification largement utilisés en apprentissage automatique, y compris les formats en virgule flottante et la quantification affine pour la conversion en entier. Enfin, nous détaillerons les procédures utilisées pour quantifier les modèles de réseaux de neurones, qui seront utilisées dans les chapitres suivants.

### Chapitre 3 : Circuits intégrés pour l'exécution de l'inférence des réseaux de neurones profonds embarqués

Le calcul avec des réseaux de neurones profonds repose sur deux tâches principales : l'entraînement et l'inférence. Aujourd'hui, la complexité de l'entraînement et la grande quantité de données nécessaires requièrent une capacité de calcul qui ne peut être assurée que par des serveurs. Les GPU restent l'unité de calcul la plus utilisée pour l'entraînement des réseaux de neurones profonds, bien qu'il existe quelques alternatives, comme les TPU. L'inférence, en revanche, est moins complexe, car elle ne nécessite plus l'exécution d'algorithmes d'entraînement ni la gestion d'une grande quantité de données. Bien que l'inférence soit tout de même principalement réalisée sur des serveurs, il est possible de déployer des réseaux de neurones directement sur des appareils périphériques. Cependant,

la taille des réseaux de neurones modernes implique des coûts en mémoire et en calcul important rendant leur déploiement sur ces appareils périphériques difficile, voire impossible. Par conséquent, des architectures spécialisées deviennent nécessaires pour répondre aux exigences de calcul des réseaux de neurones. Ces nouvelles opportunités ont conduit au développement de matériel mieux adapté aux applications embarquées, comme les edge GPU et les edge TPU, ou avec l'émergence de matériel spécialisé tel que les FPGA.

Le chapitre 3 introduit les concepts de base des architectures matérielles et des accélérateurs pouvant être utilisés pour implémenter des réseaux de neurones. Dans un premier temps, nous présenterons brièvement les différents choix populaires de matériel utilisés pour l'exécution de l'inférence sur des appareils périphériques et justifions notre décision d'utiliser un FPGA. Ensuite, nous décrierons les deux types d'accélérateurs de réseaux de neurones fréquemment rencontrés sur les FPGA. Enfin, nous détaillerons les architectures permettant de paralléliser le calcul des réseaux de neurones.

## Chapitre 4 : Une méthode basée sur la descente de gradient pour l'apprentissage de la largeur de bits pendant l'entraînement

De nombreux modèles peuvent être quantifiés de manière uniforme avec une largeur de bit inférieur à 8 bits [Zho+16], et dans certains cas, même avec des représentations ternaires (2 bits) ou binaires (1 bit). Cependant, l'utilisation une largeur de bit identique pour l'ensemble du réseau est sous-optimal, puisque les différentes couches présentent généralement des sensibilités variables à la quantification [ZBS22]. Ainsi, des largeurs de bit plus fines peuvent être utilisées à différents niveaux du modèle, comme par exemple au niveau des couches ou des noyaux de convolution. Dans la littérature, ces méthodes de quantification sont appelées *quantification en précision mixte*. Une granularité de largeur de bit affinée offre un ajustement plus flexible du compromis entre précision et efficacité que les modèles quantifiés de manière uniforme. La précision mixte est particulièrement intéressante pour les accélérateurs de flux de données, permettant une utilisation plus efficace des ressources disponibles. Cependant, l'espace de recherche pour les largeurs de bit par couche est une fonction exponentielle du nombre de couches, ce qui rend les assignations manuelles des largeurs de bit irréalisable. Cela nécessite des méthodes automatiques et spécialisées capables de déterminer efficacement les allocations en précision mixte.

Le chapitre 4 introduit une méthode d'optimisation pour l'exploration largeur de bits pour la quantification en précision mixte des poids et des activations. Tout d'abord, nous passons en revue les méthodes existantes de l'état de l'art utilisées pour aborder cette problémtique. Ensuite, nous présentons notre approche basée sur le gradient, qui utilise des largeurs de bit fractionnaires relaxées mises à jour à l'aide d'une règle de descente de gradient. Enfin, nous comparons cette approche avec d'autres méthodes basées sur le gradient et discutons de certains choix conceptuels clés.

Nos travaux démontrent que, contrairement aux approches précédentes généralement conçues pour des contextes de pré-entraînement, notre méthode est capable de fonctionner aussi bien dans des scénarios de pré-entraînement que dans des situations d'entraînement à partir de zéro. Les résultats obtenus sont comparables à ceux des approches de quantification en précision mixte de l'état de l'art sur CIFAR-10 avec un réseau ResNet-20. De plus, elle donne également d'excellents résultats avec les réseaux pré-entraînés ResNet-18 et MobileNet-V2 sur ImageNet.

## Chapitre 5 : Le format Minifloat pour améliorer l'inférence des réseaux neuronaux profonds

Si la quantification uniforme est largement utilisée pour compresser les réseaux de neurones et accélérer l'inférence, les formats d'arithmétique en virgule flottante de faible précision, aussi appelés *minifloats*, suscitent un intérêt croissant ces dernières années. L'une des principales raisons est leur utilisation pour accélérer le processus d'entraînement de grands modèles. Un autre intérêt majeur réside dans le processus de standardisation IEEE en cours concernant l'utilisation de formats en virgule flottante sur 8 bits (binary8). Nous nous attendons à ce qu'un tel standard soit adopté à l'avenir par les principaux concepteurs fournissant des accélérateurs d'IA. Bien que l'entraînement des réseaux de neurones soit la principale motivation pour l'adoption du format minifloat, l'inférence peut également tirer parti des formats en virgule flottante de faible précision.

Le chapitre 5 explore l'utilisation des formats minifloat pour l'inférence des réseaux de neurones. Tout d'abord, nous analyserons les différentes caractéristiques et propriétés des minifloats ainsi que leur utilité dans un contexte d'inférence. Sur la base de cette analyse, nous introduirons un schéma de codage en virgule flottante optimisé pour l'inférence,

basé sur un biais d'exposant asymétrique, et discutons de la manière de réaliser la quantification avec ces formats. Enfin, nous comparons cette approche avec les formats de quantification uniforme dans divers scénarios de test, en mettant en lumière ses forces et ses faiblesses.

Nos expériences sur le jeu de données Airbus Ship montrent des résultats prometteurs, les modèles minifloat étant compétitifs avec les modèles de référence en précision simple et les alternatives basées sur la quantification affine. Pour montrer l'impact matériel potentiel du format minifloat, nous avons également suggéré une implémentation d'un multiplicateur minifloat, brique de base pour un accélérateur complet.

## Chapitre 6 : Un modèle léger de segmentation sémantique pour la détection des navires

Ces dernières années, l'apprentissage profond est devenu incontournable pour les tâches de vision par ordinateur, et la segmentation sémantique ne fait pas exception. La popularisation des réseaux entièrement convolutifs avec une architecture encodeur-décodeur a permis de nombreux progrès dans ce domaine. En particulier, le modèle U-Net [RFB15] est largement utilisé dans de nombreuses tâches de segmentation sémantique comme modèle de référence. Dans la quête de précision, les réseaux de neurones négligent souvent la complexité de leurs architectures, qui comportent généralement des millions de paramètres. Par exemple, le modèle U-Net comprend 31 millions de paramètres, soit environ 288 Mo. Par conséquent, ces architectures nécessitent une puissance de calcul et une énergie importantes, rendant le déploiement de tels modèles sur des systèmes embarqués irréalisable. Des recherches récentes se sont concentrées sur la réduction de cette complexité tout en maintenant de bonnes performances de prédiction, en développant des modèles plus efficaces. Des modèles basés sur l'architecture U-Net ont été développés dans cette direction, en ajustant la profondeur [VXN20], les modules de convolution [Meh+18; Ans+22], ou le backbone de l'encodeur [EAA19]. Cependant, une caractéristique clé des architectures dit en «U» a reçu peu d'attention en ce qui concerne son empreinte mémoire : les connexions de saut. Elles sont encore plus critiques car les mouvements de données sont plus énergivores que les calculs [Hor14], et les connexions de saut dans les architectures dit en «U» impliquent des mouvements de données nettement plus importants que dans d'autres modèles plus conventionnels.

Dans le chapitre 6, nous cherchons à évaluer l'impact des connexions de saut sur la précision d'un modèle de segmentation dit en «U» pour proposer un modèle plus léger. Dans un premier temps, nous passerons brièvement en revue des modèles de segmentation utilisés pour les applications spatiales, avec un accent particulier sur une caractéristique : les longues connexions de saut. Ensuite, nous analyserons l'impact des connexions de saut sur la précision du modèle Thin U-Net pour la détection de navires afin de proposer une version plus compacte. Enfin, nous présenterons une implémentation du Thin U-Net allégé sur FPGA et analyserons le gain en termes de consommation de ressources.

Nos analyses expérimentales sur le jeu de données Airbus Ship ont révélé que la suppression des connexions de saut de début du modèle a peu d'impact sur les prédictions (<1%) tout en économisant 94% de l'utilisation mémoire des connexions de saut. La quantification du modèle sur 5 bits réduit encore davantage l'empreinte mémoire tout en limitant la perte de précision à moins de 2% par rapport au modèle en précision simple. En utilisant le logiciel open-source FINN, nous avons également déployé le modèle sur un FPGA. Nos résultats montrent un gain en consommation de mémoire, avec respectivement 37,3% et 27,7% de BRAM et de FF en moins.

## Conclusion

L'observation de la Terre par satellite fournit des données essentielles pour l'étude de notre planète. Cependant, les limitations des communications entre les satellites et les stations au sol nécessitent de réduire la quantité de données transmises. Bien que la compression des données soit couramment utilisée, l'intérêt récent se porte sur la transmission des seules informations pertinentes grâce à un traitement embarqué. L'apprentissage profond est de plus en plus utilisé dans les applications spatiales. Cependant, l'inférence des modèles est souvent réalisée sur des plateformes au sol en raison de leur empreinte mémoire et de leur intensité de calcul. Ce document présente des travaux de recherche sur la compression des réseaux de neurones convolutifs à l'aide de méthodes de quantification, en se concentrant sur la détection de navires à partir d'images satellites. Nous avons proposé une méthode d'exploration des largeurs de bits en précision mixte. Nous avons également étudié les minifloats comme format arithmétique pour l'inférence. Enfin, nous avons examiné l'architecture des modèles de segmentation sémantique dit en «U» et proposé une version plus compacte.

# Acknowledgement

I would like to express my deepest gratitude to all who contributed to my doctorate and to its success.

First of all, I am deeply grateful to my thesis director, Olivier SENTIEYS, and my supervisors, Silviu-Ioan FILIP and Clément COGGIOLA, for their support and invaluable guidance throughout my research training. Their expertise, patience, and constructive criticism have helped me improve technically and gain maturity in my research work.

I would also like to express gratitude to the reviewers and members of my doctorate Committee, Daniel MENARD, Florent de DINECHIN, David NOVO, Emanuel POPOVICI, Tania POULI, and Alexandre TERMIER for their interest in my work and for their constructive comments. Their expertise and suggestions have greatly enriched this thesis.

A big thank to my colleagues, in particular my office mates Sonia, Léo, and Louis, for their unwavering support, encouragement, and camaraderie. Their presence and assistance made this doctorate more enjoyable and fulfilling.

Finally, I dedicate these acknowledgments to my family, who have been an unending source of support and love. Their understanding, patience, and encouragement enabled me to persevere through difficult times.

# Table of Contents

# List of Tables

# List of Figures

# List of Acronyms

| | |
|---|---|
| **AI** | Artificial Intelligence |
| **ALU** | Arithmetic Logic Unit |
| **ANN** | Artificial Neural Network |
| **ASIC** | Application-Specific Integrated Circuit |
| **BN** | Batch Normalization |
| **BNN** | Binary Neural Network |
| **BRAM** | Block RAM |
| **CMOS** | Complementary Metal–Oxide–Semiconductor |
| **CNES** | Centre National d'Études Spatiales |
| **CNN** | Convolution Neural Network |
| **CPU** | Central Processing Unit |
| **DL** | Deep Learning |
| **DNN** | Deep Neural Network |
| **DRAM** | Dynamic Random-Access Memory |
| **DSP** | Digital Signal Processor |
| **ECV** | Essential Climate Variables |
| **EO** | Earth Observation |
| **FC** | Fully Connected |
| **FCN** | Fully Convolutional Network |

**FF** Flip-Flop

**FFT** Fast Fourier Transform

**FIFO** First In, First Out

**FN** False Negative

**FP** False Positive

**FPGA** Field-Programmable Gate Array

**GCOS** Global Climate Observing System

**GPU** Graphics Processing Unit

**HLC** High-Level Controller

**HLS** High-Level Synthesis

**IEEE** Institute of Electrical and Electronics Engineers

**ILSVRC** ImageNet Large Scale Visual Recognition Challenge

**Inf** Infinity

**IoU** Intersection over Union

**KL** Kullback-Leibler

**LASSO** Least Absolute Shrinkage and Selection Operator

**LLC** Low-Level Controller

**LUT** Look-Up Table

**MAC** Multiply-ACcumulate

**MCP** McCulloch-Pitts

**MCU** Micro-Controller Unit

**mIoU** mean Intersection over Union

**ML** Machine Learning

**MLP** Multi-Layer Perceptron

**MMAC** Matrix Multiply-and-Accumulate

**MVU** Matrix-Vector-Threshold Unit

**NaN** Not-a-Number

**NAS** Neural Architecture Search

**NN** Neural Network

**ONNX** Open Neural Network Exchange

**PE** Processing Element

**PTQ** Post-Training Quantization

**QAT** Quantization-Aware Training

**QONNX** Quantized Open Neural Network Exchange

**ReLU** Rectified Linear Unit

**RL** Reinforcement Learning

**RN** Round-to-Nearest

**RTL** Register Transfer Language

**SGD** Stochastic Gradient Descent

**SIMD** Single Instruction, Multiple Data

**SIMT** Single Instruction, Multiple Threads

**SoC** System-on-Chip

**SR** Stochastic Rounding

**SRAM** Static Random-Access Memory

**STE** Straight-Through Estimator

**SWU** Sliding Window Unit

**TNN** Ternary Neural Network

**TP** True Positive

**TPU** Tensor Processing Unit

**VGG** Visual Geometry Group

# NOTATIONS

## Numbers and Arrays

| | |
|---|---|
| $x$ | A real |
| $\hat{x}$ | An integer |
| $\mathbf{x}$ | A vector |
| $\mathbf{X}$ | A matrix |
| $\tilde{x}$ | A quantized real |
| $\widetilde{\mathbf{x}}$ | A vector of quantized elements |
| $\widetilde{\mathbf{X}}$ | A matrix of quantized elements |

## Sets

| | |
|---|---|
| $\mathbb{X}$ | A set |
| $\mathbb{X}^n$ | A set of $n$-dimentsional elements |
| $\mathbb{X}^*$ | A set with zero excluded |
| $\mathbb{X}_+$ | A set of positive elements |
| $\mathbb{X}_-$ | A set of negative elements |
| $\mathbb{Z}$ | The set of integers |
| $\mathbb{N}$ | The set of natural integers |
| $\mathbb{R}$ | The set of real numbers |
| $\{0, 1\}$ | The set containing 0 and 1 |
| $[\![a; b]\!]$ | The set of integers between $a$ and $b$ |
| $[a; b]$ | The real interval including $a$ and $b$ |
| $(a; b]$ | The real interval excluding $a$ and including $b$ |
| $\mathbb{A} \cap \mathbb{B}$ | Intersection of sets $\mathbb{A}$ and $\mathbb{B}$ |
| $\mathbb{A} \cup \mathbb{B}$ | Union of sets $\mathbb{A}$ and $\mathbb{B}$ |

# INTRODUCTION

In recent years, Artificial Intelligence (AI) has received much attention in scientific research and industry in a wide range of domains, including Image Processing. AI covers all the algorithms and methods used to implement a system capable of simulating intelligence. Traditionally, rule-based AI systems are designed to mimic "intelligence" using a model based solely on predetermined rules. These systems comprise a set of hand-coded rules that lead to predefined outcomes. In contrast to rule-based programming, Machine Learning (ML) algorithms learn and adapt without following explicit instructions by analyzing and inferring results from patterns in data. Today, a large part of AI is covered by ML and, more specifically, one of its sub-fields, Deep Learning (DL). This sub-field aims to mimic the functioning of the biological brain using mathematical modelizations of neurons organized in network structures. Early work on Neural Networks (NNs) introduced shallow models, which consisted of a few layers that quickly reached their limit. In the last decades, a series of developments have enabled NNs with many layers to be trained effectively, thereby removing previous limitations on the capabilities of these early models, setting the premise for what are now known as Deep Neural Networks (DNNs). Until the last decade, the increase in layers was primarily limited by the requirement of exponentially more training data to ensure their generalization ability, which has significantly slowed the progress of DNNs. At the same time, the use of convolutional layers has led to the success of Convolution Neural Networks (CNNs) for image processing [LeC+89]. Still, their big breakthrough came much later with the outstanding results of AlexNet [KSH12] on the popular ImageNet dataset. These achievements were the foundations on which DNNs exploded in popularity, mainly due to three main factors: data availability, the increase in computational processing power, especially with Graphics Processing Units (GPUs), and algorithmic advancements.

In the space industry, DL is starting to be successfully applied in various space applications such as self-driving rover reconnaissance [Swa+21], detecting space debris/waste to prevent collisions [AL20a] and celestial body detection [LS23]. Our partner in this Thesis,

Centre National d'Études Spatiales (CNES)[2], also works on space data-related DL applications, in particular, CNNs for Earth Observation (EO) satellite image processing. EO provides an effective way of exploring the physical, chemical, and biological information related to the Earth, including Essential Climate Variables (ECV), a list of variables required to study climate change defined by the Global Climate Observing System (GCOS)[3]. This information collected by EO satellites is widely used in various research fields and applications, with notable examples including maritime monitoring [SZZ20], river water level measurement [Nin+21], natural disaster detection and monitoring (*e.g.,* forest fires [Kan+22] and cloud detection [Jep+19] for weather events). In such cases, the measurements made by EO satellites are often indispensable. EO requires extracting thematic information (*e.g.,* land cover usage, biomass repartition, etc.) using data acquired from satellite sensors. It often relies on a mapping step that aims to automatically produce a semantic map containing various regions of interest based on some optical data, a domain of processing studied extensively in DL: semantic segmentation.

Since the launch of the first satellite in 1957 kicked off the Space Race, which led to the creation of the first EO satellites in the 1960s, the number of EO satellites in orbit has grown to more than a thousand in just a few decades. Over the years, the capabilities of imaging satellites have grown as much as their number. Sensors have increased resolution, and the constellations of satellites have lowered the revisit time. In consequence, the sheer volume of data created by EO satellites is hard to even comprehend. In 2008, we were already witnessing more than 10 Terabytes of daily data acquisition [TGH08]. Today, the Pléiades Neo constellation produces 40 Terabytes of image data daily [Jér19]. As launch costs keep diminishing and the number of satellites keeps growing, these enormous quantities of data should also continue to increase. Transmitting all this data is possible through communication between satellites and ground stations. However, EO systems are limited by these downlink communications due to hardware limitations, on-board power constraints, or ground-station operation costs, for example. Thus, there is a need to reduce the amount of data transmitted through the downlink. On-board image processing to improve data efficiency is not new. Compression has been studied since the early days of satellite imagery [RW73]. Nowadays, most satellite systems embed hardware dedicated to image compression to reduce data volume, such as the CCSDS-

[2]also known as the French Space Agency
[3]https://gcos.wmo.int/en/essential-climate-variables

123.0-B-2 algorithm [Spa19], which takes advantage of the high inter-channel correlation in satellite images. However, transmitting only relevant data through on-board processing has only recently started gaining interest. Recently, the space industry and our partner, CNES, have been interested in deploying DL on-board satellites for data reduction or autonomous systems. While DNNs have the ability to extract semantic information, the inference computation of many models is still mainly performed on ground platforms due to their memory footprint and computational intensity [Fur+20].

The resources needed to train these models require huge amounts of computing power that are only available in the cloud. While many inference computations are also done in the cloud, it is increasingly desirable to deploy trained DNNs to edge devices, such as mobile phones and wearable devices, due to privacy, security, and latency concerns or limitations in communication bandwidth. Modern DNNs contain at least millions of parameters and require billions of arithmetic operations, making deployment on embedded devices difficult, if not infeasible, in many cases. To mitigate these issues, various model compression techniques have been proposed, such as pruning [HMD15], weight sharing [Dup+20], knowledge distillation [TM19], and quantization [Hub+17]. Quantization, the process of approximating NN parameters with lower bit-width, is a popular technique for reducing the model memory footprint and the computation complexity while keeping the performance and quality of results as close as possible to the floating-point reference. Furthermore, the emergence of hardware platforms offering better support for low (*e.g.,* recent Nvidia GPUs and Google Tensor Processing Units (TPUs)) and custom precision (*e.g.,* Field-Programmable Gate Array (FPGA) and Application-Specific Integrated Circuit (ASIC) solutions) compute, quantization is at the forefront of methods used to increase the efficiency of DNN model inference.

## Contributions

This thesis focuses on **CNN quantization to facilitate the deployment of semantic segmentation models on satellites**. In particular, we will **study the impact of quantization on ship detection tasks from satellite images**. Our contributions can be summarized as follows:

- We **introduce AdaQAT, an optimization-based method for mixed-precision quantization of both weights and activations**. Its defining characteristic is that

it uses relaxed fractional bit-widths that are updated using a gradient descent rule but are otherwise discretized for all operations (in forward and backward passes).

- We investigate the efficient use of low-precision floating-point quantization (so-called *minifloats*). More precisely, we **introduce a minifloat encoding scheme optimized for inference based on an asymmetric exponent bias**.
- We **evaluate the impact of skip connections on the accuracy of a U-shaped segmentation model for ship detection and propose a more lightweight model**. We present a **dataflow implementation of the lightweight model on Field-Programmable Gate Array technology** and analyze the gain in available resources.

## Thesis Outline

This document is organized into two parts, *Background* and *Contributions*, each containing three chapters.

**Background.** This first part details some of the background information needed to understand the work carried out in this thesis. Chapter 1 introduces basic concepts of DL with a detailed presentation of the anatomy and training mechanisms of CNN models. Chapter 2 briefly reviews the different DNN compression methods available, emphasizing *quantization*. We also describe various quantization formats widely used in ML and detail the procedures used to quantize DNN models. Chapter 3 introduces hardware architectures and accelerators that can be used to implement DNNs.

**Contributions** The second part presents our contributions. Chapter 4 introduces AdaQAT, an optimization-based method for mixed-precision quantization of both weights and activations. Chapter 5 introduces a minifloat encoding scheme optimized for inference based on a learnable asymmetric exponent bias. Chapter 6 studies the impact of long skip connections in U-shaped models used for semantic segmentation and introduces a specialized dataflow architecture implementation of this model.

Part I

# Background

Chapter 1

# Deep Neural Networks for Computer Vision

The domain of Artificial Intelligence (AI) is vast. It covers all the algorithms and methods used to implement a system capable of simulating intelligence. A large part of AI is covered by Machine Learning (ML), which enables computers to learn how to carry out tasks that are not directly programmed. Its mathematical foundation is that of statistical models and algorithms that can learn from data and that can generalize to new, unseen data. Here, we focus on the evolution of ML methods based on neural networks as these have become a cornerstone of modern ML and have proven to be a very effective approach in a wide range of applications over the last few years.

The study of Artificial Neural Networks (ANNs) originates from scientists' ambition to build a computer system simulating the human brain. Building such a system requires understanding how our cognitive system functions. The genesis of ANNs actually stems from philosophy questions in associationism theory[1] [Bai73], rather than a computer science or a biology problem. Early work on ANNs introduced simple models that can be viewed as a single layer. The depth of these early models was mainly limited by the training methods used at the time. In the last decades, a series of developments allowed Neural Networks (NNs) with many layers to be trained effectively, thereby removing previous limitations on the capabilities of these early models, setting the premise for what is known as a DNN. The area of study for such networks is called DL, a sub-field of the larger family of ML [LBH15].

This chapter aims to introduce the basic concepts of DL that will facilitate the comprehension of the work carried out in this thesis. In Section 1.1, we briefly review the history of DL, covering some of the major breakthroughs. Then, in Section 1.2, we describe in detail the anatomy of DL models, starting with a description of perceptrons and how they

[1] Associationism is a theory states that the mind is a set of conceptual elements that are organized as associations between these elements.

interlock to form neural networks. In Section 1.3, we explain the training mechanisms of neural networks. whereas in Section 1.4, we discuss two popular computer vision tasks, classification and segmentation, along with modern DL models that perform well on these tasks. These models will be used in later chapters to validate the approaches introduced by the thesis.

## 1.1 History of Deep Learning

The first model of an artificial neuron can be traced back to 1943 when Warren McCulloch and Walter Pitts speculated on the inner workings of biological neurons and modeled a primitive artificial neural network using electrical circuits [MP43]. Their model, known as the McCulloch-Pitts (MCP) neural model, consists of an aggregation function that sums inputs weighted with binary values followed by a thresholding function, which determines the neuron's action. Following the success of the MCP neural model, Frank Rosenblatt introduced perceptrons [Ros58], an electronic device that has proved capable of association learning. Rosenblatt's perceptron is very similar to the structure of modern artificial neurons in a neural network.

After several years without any major breakthroughs, Kunihiko Fukushima proposed, in 1980, the Neocogitron [Fuk80], a hierarchical, multilayered artificial neural network used for handwriting recognition and other pattern recognition problems. It is generally seen as the inspiration for the CNN model. Not too long after, David Ackley, Geoffrey Hinton, and Terrence Sejnowski showed in 1985 that backpropagation in neural networks could yield interesting distributed representations [AHS85].

The very first practical demonstration of CNNs appeared in the late 1980s with the contributions from Yann LeCun, John Denker, and Sara Solla on the LeNet network [LeC+89], a combination of convolutional layers and backpropagation for handwritten digit recognition. However, it was not until 2012 that DL became mainstream with the outstanding performance of AlexNet [KSH12] in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) image classification challenge. AlexNet is the first CNN model that could perform well on the challenging (at the time) ImageNet-1K dataset. The success of AlexNet was due not only to its unique architectural design, but also to the clever mechanisms used in its training [Alo+18b].

Since then, DL has gone from strength to strength, with the number of publications

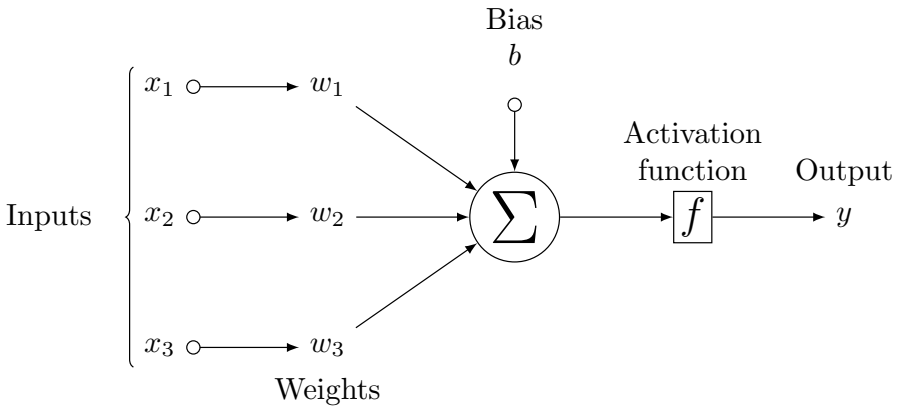



Figure 1.1: Illustration of a three-input artificial neuron. The weighted inputs are affinely combined with a bias term. The result is then passed through a nonlinear activation function to form the output.

severalfold in the last decade. This success is mainly explained by the availability of increased computing capability, particularly the benefits of GPUs for training and the amount of available data needed to train modern networks. DL has been applied with great success in many fields and for many applications such as image generation [Gre+15], strategy board games [Moz17], protein folding [Jum+21], and natural language processing [Ach+23], to name but a few.

## 1.2 Anatomy of a Deep Neural Network

Like many brain-inspired algorithms, the main element of an ANN is the artificial neuron or perceptron (Figure 1.1). Concretely, the output of an $n$-input neuron is defined as:

$$y = f(\mathbf{x}^\intercal \mathbf{w} + b) = f\left(\sum_{i=1}^{n} x_i w_i + b\right), \tag{1.1}$$

where $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}^\intercal \in \mathbb{R}^n$ is the input feature vector (also known as input activation vector), $\mathbf{w} = \begin{bmatrix} w_1 & w_2 & \cdots & w_n \end{bmatrix}^\intercal \in \mathbb{R}^n$ is the associated weight vector, $b \in \mathbb{R}$ is a bias term and $f : \mathbb{R} \to \mathbb{R}$ a nonlinear *activation* function. The weights and bias are the neuron parameters optimized during the training stage (see Section 1.2). The activation function $f$ usually acts as a (soft) threshold on the weighted input combinations.

By connecting multiple neurons to one another, we get a neural network. Doing it

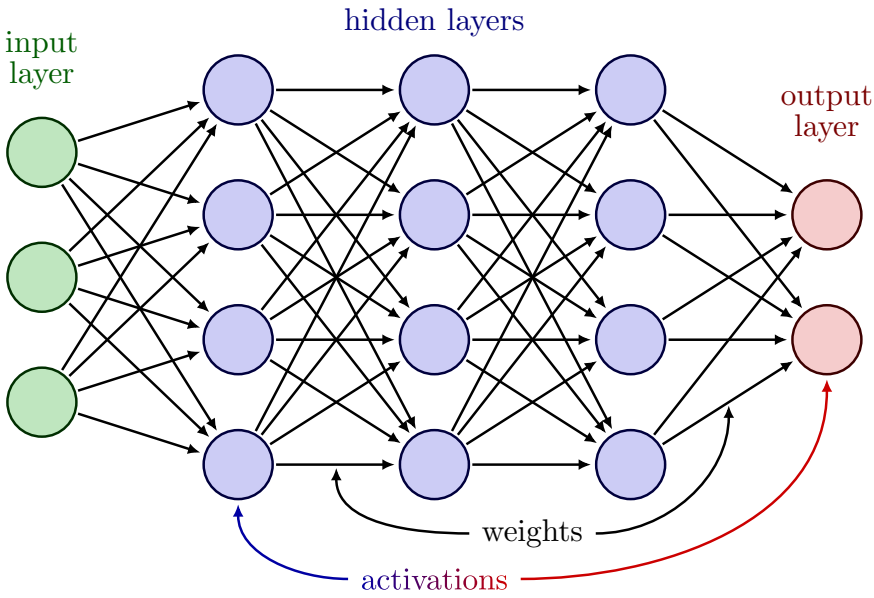



Figure 1.2: Simple feed-forward neural network example and basic layer terminology.

in a directed acyclic graph fashion, we get so-called feed-forward neural networks. A prototypical example is shown in Figure 1.2, which corresponds to what is known as a Multi-Layer Perceptron (MLP), one of the most basic types of neural network. Far from random, such networks follow a layerwise architecture. The neurons in the *input* layer receive input data and propagate them to the neurons in the network's subsequent *hidden* layers. The activations from hidden layers are ultimately propagated to the *output* layer, which makes, for instance, the final predictions in case of a classification problem.

The need for depth, *i.e.,* a network with many layers, is long known. In 1985, [Yao99] had already shown the limitations of shallow circuit functions. Later, [ES16] presented more thorough proof that the depth of a neural network is exponentially more valuable than its width for a standard MLP when using common activation functions such as rectified linear units, sigmoids or thresholds. The idea of ANNs with a big number of layers (modern networks can reach depths going into the thousands) is central to DL. DNN-based methods differ from traditional ML techniques in that they can automatically learn representations from data such as images, video, or text without introducing hand-coded rules. DNNs have the ability to extract powerful high-level features from the raw input more successfully than shallower neural networks. However, the increase in layers is limited by the requirement of exponentially more training data to ensure their

generalization ability. Nevertheless, DNNs have been widely adopted in practice, and DL is today the most popular branch of AI.

## Fully Connected Layers

*Fully Connected (FC)* layers, also known as *dense* or *linear* layers are the backbone of MLPs and many modern DL architectures. An $m$-neuron FC layer where each neuron processes the same $\mathbf{x} \in \mathbb{R}^n$ input activation vector produces an output activation vector $\mathbf{z} \in \mathbb{R}^m$. Each neuron has its own weight vector $\mathbf{w}_i \in \mathbb{R}^n$ and bias term $b_i \in \mathbb{R}$, such that

$$z_i = \mathbf{x}^\intercal \mathbf{w}_i + b_i = \sum_{k=1}^{n} x_k w_{ik} + b_i, \qquad i = 1, \ldots, m.$$

In matrix form, we have

$$\mathbf{z}^\intercal = \mathbf{x}^\intercal \mathbf{W} + \mathbf{b}^\intercal, \tag{1.2}$$

where $\mathbf{W} = \begin{bmatrix} \mathbf{w}_1 & \mathbf{w}_2 & \cdots & \mathbf{w}_m \end{bmatrix} \in \mathbb{R}^{n \times m}$ is the corresponding weight matrix and $\mathbf{b} = \begin{bmatrix} b_1 & b_2 & \cdots & b_m \end{bmatrix}^\intercal \in \mathbb{R}^m$ is the bias vector. We can also define the right hand side of (1.2) as an *affine* function

$$\varphi(\mathbf{x}) := \mathbf{z}^\intercal = \mathbf{x}^\intercal \mathbf{W} + \mathbf{b}^\intercal. \tag{1.3}$$

An $L$-layer MLP network $g$ is thus a chain of affine & activation function pairs $\{\varphi_i, f_i\}_{i=1}^{L}$ such that the network output is

$$\mathbf{y} = g(\mathbf{x}) := \left(f_L \circ \varphi_L \circ \ldots \circ f_2 \circ \varphi_2 \circ f_1 \circ \varphi_1\right)(\mathbf{x}). \tag{1.4}$$

In convolutional models, FC layers are typically found towards the end of a neural network architecture and are responsible for producing final output predictions. In Transformer-based architectures, FC layers are present all thoughout the model and account for much of the compute time.

## Hidden Layer Activation Functions

Activation functions $f$ determine whether or not a neuron should be activated. They are differentiable nonlinear operators for transforming input signals into output signals. Activation functions are typically applied after each fully connected or convolution layer. Figure 1.3 illustrates popular activation functions used in DL.

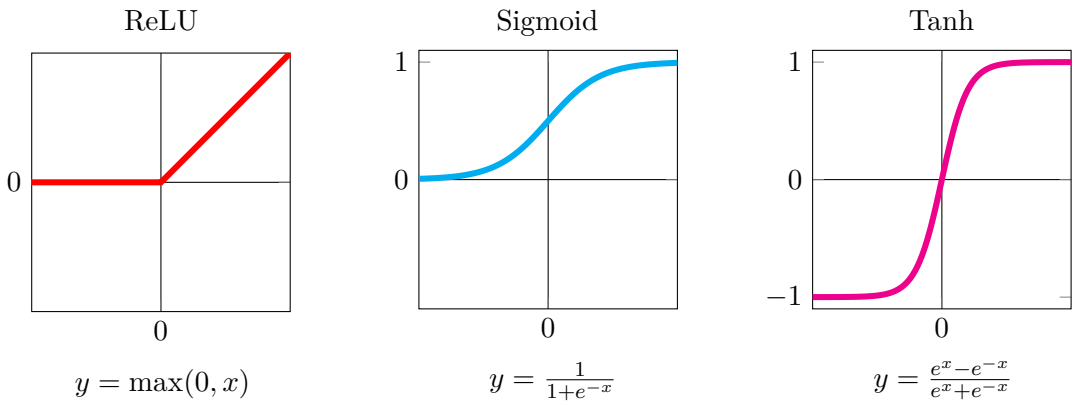



Figure 1.3: Overview of three popular activation functions: ReLU, Sigmoid and Tanh.

Sigmoid and hyperbolic tangent activations were amongst the first to be used in gradient-based learning approaches. However, using them in deep architectures poses numerical challenges during gradient computations [MOM12], which is why they were superseeded by simpler and easier to train with alternatives. Rectified Linear Unit (ReLU) [NH10] activations (and variations such as leaky ReLU [MHN+13]) have emerged as the most popular alternative, especially in the context of CNNs and image processing applications. Other alternatives that lead to better accuracy have also been proposed in recent years, with functions like Swish [RZL17] in reinforcement learning contexts and GeLU [HG16] for Transformer-based architectures in Natural Language Processing.

## 1.3 Training and Inference

There are two distinct stages in the use of neural networks: *training* and *inference*. In the context of *supervised learning* (the type of ML problems we consider in this thesis), we start with a labeled *training dataset* $\{(\mathbf{x}_i, \mathbf{t}_i)\}_{i=1}^{N}$ consisting of input (features) $\mathbf{x}_i \in \mathbb{R}^n$ and output (labels) $\mathbf{t}_i \in \mathbb{R}^m$ pairs. During training, the network will *learn* from this dataset how to compute outcomes $\mathbf{y}_i$ (*i.e.,* predicted labels) for the given training inputs $\mathbf{x}_i$ by iteratively updating a vector of learnable parameters, which we denote with $\mathbf{W}^g$, containing for instance weight and bias terms like we saw in (1.1) and (1.2). In more abstract terms, we are searching for a model $g := g(\mathbf{x}; \mathbf{W}^g)$ such that for each $i \in \{1, \ldots, N\}$, $\mathbf{t}_i$ and $\mathbf{y}_i = g(\mathbf{x}_i; \mathbf{W}^g)$ are *close* to one another. This notion of closeness is

captured through a *loss function* $\ell : \mathbb{R}^m \times \mathbb{R}^m \to [0, \infty)$. The goal is, in fact, to minimize a *global loss function* $\mathcal{L}$, taken as the mean loss over the entire training dataset:

$$\mathcal{L}(\mathbf{W}^g) := \frac{1}{N} \sum_{i=1}^{N} \ell \left( g(\mathbf{x}_i; \mathbf{W}^g), \mathbf{t}_i \right). \tag{1.5}$$

Since we cannot feasibly train the network on all the data it will see in practice, this loss acts as an approximation for the loss computed on all inputs, and in practice, we measure the performance of the trained model on an unseen *test dataset*. Both datasets should be representative of the inputs the network will see in practice.

When experimenting with various models for a particular task, a *validation dataset* is also used to compare the models between them and fix hyperparameters. Because we target only one model at a time for a particular task in our experiments, we omit the use of explicit validation datasets.

Once the model has been trained, it is used during the inference phase to make predictions. At this stage, the learnable parameters are fixed, and the network is used with real world input data not seen during the training process.

In this section, we discuss the basics of neural network training using *gradient descent*-based optimization algorithms. For a more comprehensive overview and further references on NN training methods, see [Zha+23, Ch. 12] and [BB23, Ch. 7–8].

### 1.3.1 Gradient Descent-Based Learning

Gradient descent is a popular optimization strategy used in Deep Learning. It iteratively increments model parameters $\mathbf{W}^g$ in the direction of the negative gradient of the loss function $\mathcal{L}$, with the goal that $\mathbf{W}^g$ will converge to a global minimizer of $\mathcal{L}$. Going from iteration $T$ to $T+1$, the gradient descent update rule takes the form

$$\mathbf{W}^g_{T+1} = \mathbf{W}^g_T - \eta_T \frac{\partial \mathcal{L}(\mathbf{W}^g_T)}{\partial \mathbf{W}^g_T}, \tag{1.6}$$

where the learning rate $\eta_T > 0$ determines how big the update step is at each iteration. The update rule in (1.6) corresponds to what is known as *full-batch gradient descent*.

**Stochastic Gradient Descent.** When the training dataset is large (*e.g.* $N$ in the order of millions of samples), the compute and memory requirements for determining the gradients of the loss function in (1.5) with respect to all $N$ training samples can become

prohibitively large. A more compute and memory-friendly approach is to do a *stochastic* weight update in which $\mathcal{L}$ is replaced at each iteration $T$ with $\ell$ on one training sample $r_T \in \{1, \ldots, N\}$, chosen uniformly at random. This Stochastic Gradient Descent (SGD) update rule corresponds to

$$\mathbf{W}^g_{T+1} = \mathbf{W}^g_T - \eta_T \frac{\partial \ell(\mathbf{y}_{r_T}, g(\mathbf{x}_{r_T}; \mathbf{W}^g_T))}{\partial \mathbf{W}^g_T}. \tag{1.7}$$

While these updates are extremely fast, they generally lead to noisy gradients that overall can make convergence slower when compared to full-batch gradient descent. On the positive side, this noise is usually beneficial in escaping local minima of $\mathcal{L}$ and allowing the model $g$ to generalize better to unseen data [Pri23, Ch. 9.2].

In practice, a hybrid approach between (1.6) and (1.7) is usually taken, retaining the advantages of the two. *Mini-batch gradient descent* splits the training data set into batches $\mathcal{B}_T$ of size $B \ll N$, with the loss estimate being the mean on the batch samples

$$\mathcal{L}_{\mathcal{B}_T}(\mathbf{W}^g) := \frac{1}{|\mathcal{B}_T|} \sum_{i \in \mathcal{B}_T} \ell(\mathbf{y}_i, g(\mathbf{x}_i; \mathbf{W}^g))$$

with the associated update rule

$$\mathbf{W}^g_{T+1} = \mathbf{W}^g_T - \eta_T \frac{\partial \mathcal{L}_{\mathcal{B}_T}(\mathbf{W}^g_T)}{\partial \mathbf{W}^g_T}. \tag{1.8}$$

At the start, the entire training dataset is partitioned (randomly) into $\lceil N/B \rceil$ batches, the $\lceil N/B \rceil$ iterations of (1.8) required to go over the entire set corresponding to an *epoch* of training. The process of going over the entire dataset through these batches is then repeated. In the rest of the document, we will usually report on the time required to train a network in terms of the total number of epochs. Power of 2 mini-batch sizes are frequently used. The SGD moniker is frequently used with the mini-batch version of gradient descent as well, a practice we will also follow in this document.

**Accelerated Optimization.** This basic SGD update rule can be modified in many ways, leading to various so-called *accelerated* optimization algorithms. Some of the more popular ones include momentum [Pol64], Nesterov momentum [Nes83], Adam [KB14] and AdamW [LH17].

**Computing Gradients through Backpropagation.** The most time-consuming part of the training procedure is computing the gradients of the loss with respect to the parameters. The most efficient algorithm in use today is *backpropagation*, devised during the 1980s [RHW86]. It makes heavy use of the chain rule from Calculus to back-propagate gradients in the network from the last one to the first.

**Learning Rate Scheduling.** When training DNNs, adjusting the learning rate as the training progresses is crucial to convergence. One main reason is that optimization will diverge if the learning rate is too large, and it will either stagnate or converge to suboptimal results if it is too small. Learning rate schedulers seek to adjust $\eta_T$ during training by reducing the learning rate according to a pre-defined schedule. There is a wide variety of learning rate schedulers, but amongst the most popular ones, we find step decay, cosine [LH16], and cyclical [Smi17].

**Parameter Initialization.** Another crucial aspect of training is how model parameters are initialized at the start of gradient descent-based learning. Although it is common practice to use randomly distributed values, DNN models struggle to converge when this is not done carefully. If the variance of the parameters is not properly taken into account and controlled, it can lead to gradient magnitudes exploding or vanishing during backpropagation. This problem worsens as the depth of NNs increases. Using a specific initialization strategy is recommended to avoid bad starting points. Two such popular strategies that we will use are Xavier [GB10] and Kaiming [He+15] initialization.

There are cases when it is not necessary to start training from scratch. This is exemplified, for instance, by generalist models that have been pre-trained on huge labeled datasets and that now need to be adapted to a more specialized context. Such models are then trained again (*i.e., fine-tuned*) on a usually new, smaller dataset, a process known as transfer learning. Such re-training is also common when adding new constraints to the problem, such as optimizing for memory size (through quantization or removing redundant parameters) or other hardware performance metrics. Quantization, in particular, has benefited immensely from fine-tuning approaches, a topic we will cover in-depth in later chapters. Two major advantages of fine-tuning are that it generally leads to much faster training times than starting from scratch and improved accuracy, but if not done carefully (*e.g.* if the new dataset is vastly different from the one used in the pre-training stage), can lead to overfitting.

Overfitting happens when a model performs very well on the training set, but poorly on new data, in particular the test dataset. During training this means that the loss function on the training dataset continues to decrease, whereas on the test set it will increase. One potential way of avoiding this overfitting phenomenon is to increase the training set, allowing for a greater variety of samples. This increase can be obtained through so-called data augmentation, *e.g.* geometric transformations such as translation or rotation, color space changes, or noise injection.

**Regularization Techniques.** If enriching the training dataset is not an option, *regularization techniques* such as $L_1$, $L_2$ and Dropout [Sri+14] are also applicable. Not limited by such use cases, regularization in general is a process that leads to *simpler* models. This can mean various things. We have talked about avoiding overfitting, but it can also be about promoting models that are less memory-intensive (*e.g.* quantizable to very small bit-widths or promoting sparse weights). In modern ML, regularization can be either *explicit* or *implicit*. Explicit regularization is characterized by the addition of a penalty term $R$ to the loss function, which becomes

$$\mathcal{L}(\mathbf{W}^g) + \lambda R(\mathbf{W}^g). \tag{1.9}$$

The value of $\lambda > 0$ in (1.9), the regularization coefficient, controls the strength of the regularization term, and is usually a small value, the idea being that regularization should not impede $\mathcal{L}$ from decreasing, still the main goal of the training process.

One popular choice is $L_2$ regularization [HK70], also known as ridge regression or weight decay in the ML literature, which takes the form

$$R(\mathbf{W}^g) = \|\mathbf{W}^g\|_2^2 = \sum_{i=1}^{P} w_{g,i}^2,$$

where $P \in \mathbb{N}$ is the size of the $\mathbf{W}^g$ model parameter vector. It reduces overfitting by encouraging weight values to decay towards zero. Another choice that has enjoyed wide usage is $L_1$ regularization [Tib96], also known as least absolute shrinkage and operator selection Least Absolute Shrinkage and Selection Operator (LASSO). It uses the sum of the parameter magnitudes in the regularizer term

$$R(\mathbf{W}^g) = \|\mathbf{W}^g\|_1 = \sum_{i=1}^{P} |w_{g,i}| \,.$$

Similar to ridge regression, it results in parameters that decay to zero, but additionally promotes sparsity, leading to it being widely used in sparse neural network design methods. In Chapter 4 we will use explicit regularization as a means to produce mixed-precision quantized DNNs with small bit-widths.

Implicit regularization accounts with all other forms of regularization. In DL, this includes approaches like early stopping of the optimization process, normalization techniques such as Batch Normalization (BN) [IS15], and randomly dropping parameters during training with Dropout. BN, in particular, is credited with accelerating convergence by improving the numerical behavior of DNNs during training. For a minibatch $\mathcal{B}$, the input $\mathbf{x}$ to a batch normalization layer is transformed as follows:

$$\mathrm{BN}(\mathbf{x}) = \gamma \frac{\mathbf{x} - \mu_{\mathcal{B}}}{\sigma_{\mathcal{B}}} + \beta,$$

where $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$ denote the mean and standard deviation of the minibatch, and $\gamma$ and $\beta$ are two learnable parameters. At the end of this process, the resulting minibatch has zero mean and unit variance. BN implementations for FC and convolutional layers differ slightly. For FC layers, BN is inserted right after activation functions, while for convolutional layers, it is applied before the nonlinear activation function. In computer vision, BN layers are generally used exclusively for convolutions.

### 1.3.2 Inference

After a neural network has been trained, it can be used to make predictions on new data, using the patterns and relationships it has learned during training. This process is called *inference*. During this phase, data is passed only in the forward direction through the network (like in (1.4)), also known as *forward propagation*, with the network parameters being essentially constants. Avoiding the backward propagation computations that are required for training leads to faster and less resource-intensive computations. However, inference represents a non-negligible proportion of a neural network's lifecycle. Companies such as Nvidia and Amazon estimate that $\sim$ 90% of ML demand in the cloud is due to inference [Pat+21]. Thus, the impact of these computations, particularly on power consumption, should not be underestimated. In this work, we will focus exclusively on improving inference quality metrics. Our goals are reducing energy consumption and computing complexity of DNN models during inference.

## 1.4 Convolutional Neural Networks

Computer Vision is a field of AI that focuses on the automatic analysis and interpretation of image data. Historically, computer vision has been based on statistical methods and three-dimensional projective geometry. However, since the DL revolution, it has shifted to CNN models that achieve state-of-the-art results on specific tasks. Recently, alternative NN architectures based on transformers have become competitive with CNNs [Kha+22].

Machine Learning has many applications in Computer Vision. Some examples include image classification, object segmentation, super-resolution, and image colorization. In this section, we overview some core elements of CNN-based Computer Vision, focusing on the problems, models, and datasets that will be extensively used in later chapters.

### 1.4.1 Image Data

In a digital context, an image consists of a rectangular array of pixels formed by one or more channels. In particular, color images are composed of a triplet of red, green, and blue channels, each with its own intensity values. These intensities are encoded with non-negative integers. Images generally have large sizes, with typical cameras capturing images comprising tens of megapixels (a unit equivalent to one million pixels expressing the total number of pixels in the image). This is even more the case with high-resolution images acquired by Earth observation satellites, with images that reach hundreds of megapixels and large bit encodings (typically 10 to 16 bits). This large size makes it difficult to use networks comprised of only fully connected layers, which would require a vast number of learnable parameters. Furthermore, nearby image pixels are statistically related, something that fully connected layers can struggle with since they do not have any a priori notion of vicinity and they interpret every input equally.

### 1.4.2 Convolutional Layers

This is not the case with convolutional layers, the backbone of CNN architectures. They process each local image region independently, using parameters shared across the whole image. They use fewer parameters than fully connected layers, exploit the spatial relationships between nearby pixels, and do not have to explicitly learn relationships between the pixels at every position.

Original image | $3 \times 3$ Gaussian blur | Sharpening | Edge detection

Figure 1.4: Examples of blurring, sharpening, and edge detection effects achievable by convolving kernels and images.

**Convolutions.** In digital image processing, two-dimensional convolutions consist of combining an image, known as an *input feature map*, with a filter whose size and values determine the nature of the effect. The result is a filtered image, an *output feature map*. Depending on the element values, a kernel can produce a wide range of effects, including blurring, sharpening, or edge detection, as illustrated[2] in Figure 1.4.

The convolution window (*i.e.,* the image area affected by the filter) slides over the entire input feature map, from left to right and top to bottom. At the corresponding location, the convolution window and the kernel matrix are multiplied elementwise, and the resulting matrix is summed up, yielding a single scalar value. This result gives the value of the output matrix at this particular location. Figure 1.5 illustrates how convolution works for a $9 \times 9$ raster image of digits and a $3 \times 3$ filter. The orange portion is the convolution window. Convolved with the kernel, it produces the result of the blue portion in the output feature map:

$$1 \times 1 + 1 \times 0 + 0 \times 0 + 0 \times 0 + 0 \times 1 + 1 \times 1 + 0 \times 1 + 0 \times 0 + 1 \times 0 = 2.$$

The general expression for a two-dimensional convolution can be stated as

$$\mathbf{I}'(x, y) = \mathbf{K} * \mathbf{I}(x, y) = \sum_{i=0}^{K_x - 1} \sum_{j=0}^{K_y - 1} \mathbf{K}(i, j) \cdot \mathbf{I}(x + i, y + j), \tag{1.10}$$

where $\mathbf{I}'$ is the output feature map, $\mathbf{I}$ is the input feature map (image), $\mathbf{K}$ is the filter, or kernel, which is represented by a $K_x \times K_y$ matrix, and $(x, y)$ are the coordinates of the pixels in the output feature map.

[2]Image credit: https://en.wikipedia.org/wiki/Kernel_(image_processing).

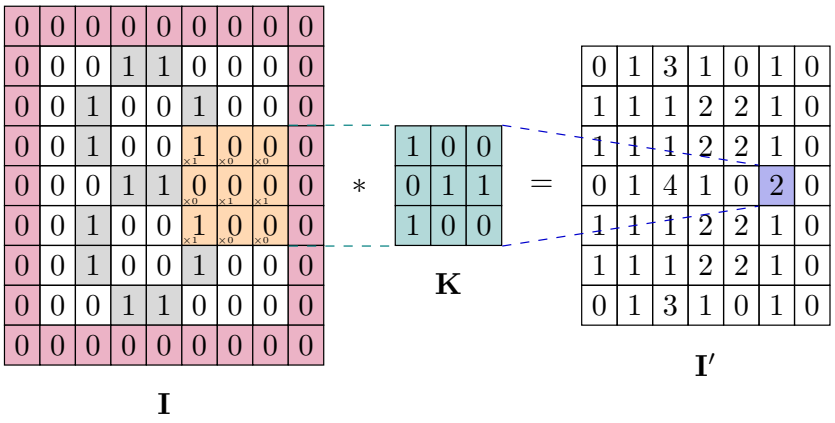



Figure 1.5: An example of a two-dimensional convolution operation with a $3 \times 3$ kernel with zero padding on the input feature map and a stride of one.

**Padding.** Convolutions tend to lose pixels on the perimeter of the feature map, effectively reducing the output size compared to the original input. In some cases, keeping the exact same size as the input is preferable. In such cases, padding is added to the input to preserve the output size. It consists of adding extra filler pixels around the boundary. A typical choice, called *zero padding*, is to set the padding values to zero (see Figure 1.5 for an example). Padding can be applied to any input feature maps of convolutional layers inside the network.

**Stride.** Size reduction of convolution outputs is sometimes desired when images or feature maps are very large, ensuring a certain degree of flexibility in the design of CNN architectures. One way to achieve this reduction is to use strided convolutions: instead of stepping the filter over the image one pixel at a time, it is moved in large steps. For example, if we have a stride of two, the output feature map generated by the convolution will be twice as small as the input.

**Multi-Dimensional Convolutions.** As previously mentioned, when working with color images, inputs are not two-dimensional objects but rather three-dimensional, characterized by height, width, and channel as depicted in Figure 1.6. While the first two axes concern spatial relationships, the third can be regarded as assigning a multidimensional representation to each pixel location. In the case of network inputs, the channel dimension usually corresponds to a color (red, green, and blue). The convolutional expression

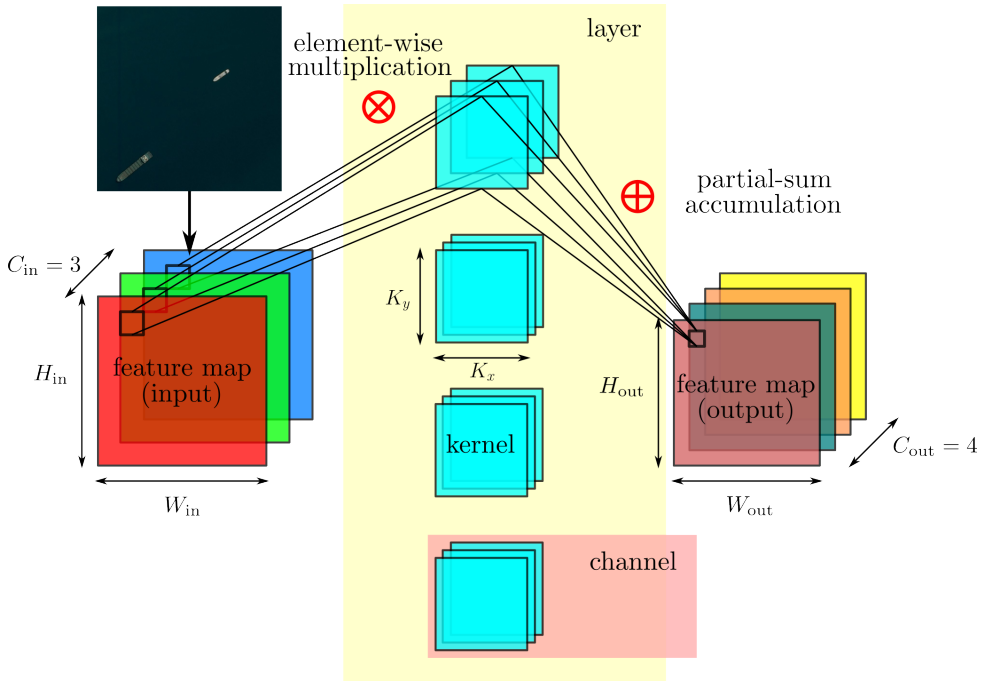



Figure 1.6: Expanded view of a typical two-dimensional convolutional layer.

(1.10) transforms into

$$\mathbf{I}'(c,x,y) = \sum_{k=0}^{C_{in}-1} \sum_{i=0}^{K_x-1} \sum_{j=0}^{K_y-1} \mathbf{K}(c,k,i,j) \cdot \mathbf{I}(k,x+i,y+j),$$

where $\mathbf{I}'$ is the output channel indexed by $c \in [\![0, C_{out}-1]\!]$, $C_{out}$ is the number of output channels, $\mathbf{I}$ is the input image, $C_{in}$ is the number of input channels, $\mathbf{K}$ is the kernel tensor[3] of size $C_{out} \times C_{in} \times K_x \times K_y$, $(x,y)$ are the output pixel coordinates at channel $c$.

**The Complete Layer.** The purpose of having multiple channels and associated filters is to build more flexible models, where each filter has an independent set of learnable parameters (*i.e.,* the elements of $\mathbf{K}$) that can extract relevant pieces of information from the inputs. The network learns distinctive patterns that trigger an activation function upon encountering specific learned features at particular spatial locations within the input. Equipped with an optional set of bias parameters, a complete convolutional layer takes

[3]In our setting a tensor is a multidimensional array with real-valued elements.

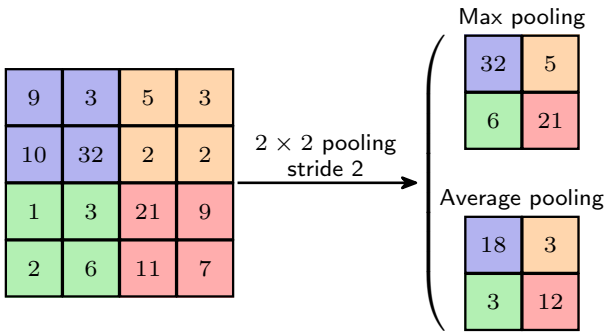



Figure 1.7: Illustration of max and average pooling operations for a $2 \times 2$ window and a stride of 2 on a $4 \times 4$ input feature map.

the form

$$F(c, x, y) = f\left(b_c + \sum_{k=0}^{C_{in}-1} \sum_{i=0}^{K_x-1} \sum_{j=0}^{K_y-1} \mathbf{K}(c, k, i, j) \cdot \mathbf{I}(k, x+i, y+j)\right),$$

where $F$ is the $C_{\text{out}}$ channel output feature map, $f(\cdot)$ is an activation function and $b_c$ is the scalar bias associated with the output channel $c$. Each output channel will have its own associated bias parameter.

### 1.4.3 Pooling

Pooling layers are another essential operation that reduces the size of an input feature map but does so in a simple and compute-efficient way using a fixed function of locally spaced inputs without any learnable parameters. It is frequently used to down-sample feature maps and produce new ones with a condensed resolution. Pooling operators also provide a form of spatial transformation invariance[4] that is useful in image classification tasks.

The most popular pooling variants are maximum and average pooling. They compute the maximum or average value of the elements in a pooling window over the input, respectively. A simple example of maximum and average pooling is shown in Figure 1.7. Pooling layers change the output shape. As with convolutions, they can be adjusted to achieve a desired output shape by padding the input and adjusting the stride.

[4]The output should be invariant to translations of the input.

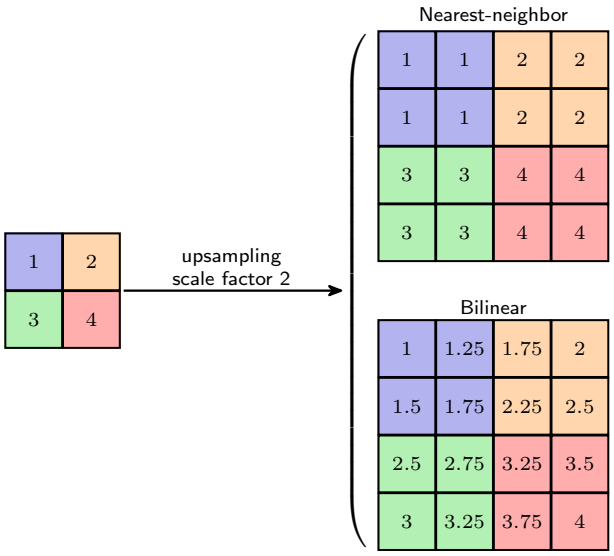



Figure 1.8: An example of two-dimensional nearest-neighbor and bilinear interpolation operations with a scale factor of two.

### 1.4.4 Unpooling

Both convolutional and pooling layers progressively reduce the spatial dimensions of inputs and intermediate feature maps. In image segmentation tasks that perform classification at the pixel level, it is convenient to reverse the process, gradually restoring the spatial dimensions of the feature maps while reducing the number of channels. This process is known as *unpooling*. Two main methods to increase the size of a feature map are *upsampling interpolation* and *transposed convolution*. Upsampling interpolation is based on mathematical transformations, while transposed convolution is similar to the convolution layer with parameters (weights and bias) optimized during the training stage.

**Upsampling Interpolation.** Upsampling interpolation is based on mathematical transformations from image processing used to rescale, translate, shrink, or rotate images. It uses known pixels to estimate the pixel values of the new upsampled image. Two interpolation algorithms are frequently used among a wide variety of existing interpolation methods (Figure 1.8):

- **Nearest-neighbor interpolation** is the simplest and fastest method as it involves duplicating existing pixels to create a new upsampled image. This makes each pixel bigger, which may result in a blocky or pixelated appearance.

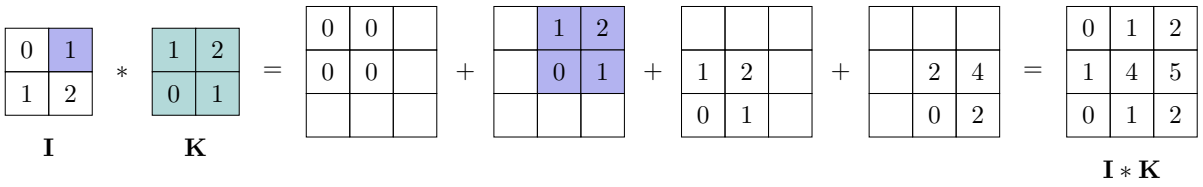



Figure 1.9: Example of a two-dimensional transposed convolution operation with a $2 \times 2$ kernel and stride of one.

- **Bilinear interpolation** computes the value of new pixels by taking a weighted average of the nearest four pixels. On the other hand, it gives rise to a smoother image than nearest-neighbor interpolation.

**Transposed Convolution.** Transposed convolution, also referred to as fractionally-strided convolution, is another method used to increase the size of output feature maps. They broadcast input elements via a kernel, producing a larger output. This layer has the same attributes as a standard convolution, including learnable parameters (weight and bias) optimized during training. Instead of applying a kernel to a convolution window as a standard convolution does, a kernel is applied to each location of the input feature map with a stride that determines the distance between successive kernel positions. Each kernel output is then added to the corresponding location in the output feature map as illustrated in Figure 1.9. Note that, unlike interpolation, it is possible to impose a user-defined number of output channels for a transposed convolution layer.

## 1.5 Representative Computer Vision Tasks

### 1.5.1 Image Classification

Image classification, sometimes also called image recognition, involves assigning a label to an image from a predefined set of categories. Since it is a well-known and much-studied task widely adopted as a benchmark, it is one of the main candidates in comparing our methods with the state-of-the-art.

**Datasets.** Among the wide variety of image classification datasets, we chose to start with the CIFAR-10 dataset for fast experimentation before moving on to the larger and more complex ImageNet dataset. Both datasets can be summarized as follows:

- **CIFAR-10** [KH09] consists of 60,000 color images of size $32 \times 32$ divided into ten classes of animals and vehicles. Each class contains 6,000 images and is entirely mutually exclusive. The dataset is split into 50,000 training images and 10,000 test images. While relatively small by current standards, it is still widely used for benchmarking ML-based computer vision algorithms.

- **ImageNet** [Den+09] contains 14,197,122 hand-annotated color images divided into 21,841 classes. The project does not own the copyright of the images, only thumbnails and URLs of images are provided. Since 2010, the dataset has been used in the ILSVRC, an annual Computer Vision contest on image classification and object detection. The initial dataset being extremely large, various subsets have been proposed, such as Tiny ImageNet [LY15] with 100,000 images of 200 classes downsized to 64×64. The most highly used subset is referred to as ILSVRC2017 or ImageNet-1K. This subset contains 1,281,167 training images, 50,000 validation images, and 100,000 test images divided into 1,000 classes. It is widely used in the research literature to benchmark DL algorithms. Note that the images have different resolutions, with an average resolution of $469 \times 387$, and are usually resized to $256 \times 256$ in an image preprocessing step.

**Accuracy Metrics.** The accuracy score is the most straightforward way to measure a classifier's performance. This metric measures the proportion of correct predictions made by the model out of all the predictions.

The top-$k$ accuracy score is a generalization of this. The difference is that a prediction is considered correct if the true label is associated with one of the $k$-highest predicted scores:

$$\text{Top-}k = \frac{1}{N_s} \sum_{j=1}^{k} TP_j,$$

where $TP_j$ is the number of true positive images in $j$-th position of the best prediction and $N_s$ is the total number of samples. Note that the accuracy score is the special case of $k = 1$. Top-5 accuracy, popularized by the ImageNet challenge, is another common metric in image classification evaluation.

There are other metrics, such as Precision & Recall or F1-Score, but top-$k$ and, more specifically, top-1 is by far the most widely used to evaluate the performance of a classification algorithm.

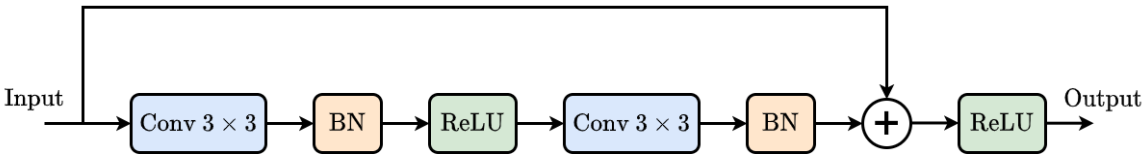



Figure 1.10: Illustration of a standard residual block.

**Network Architectures.** With the success of LeNet [LeC+89] and LeNet-5 [LeC+98] on the MNIST [Den12] hand-written digit recognition dataset, and the breakthrough of AlexNet [KSH12] on ImageNet-1K, CNNs have shown great potential in solving Computer Vision tasks. These successes and others have attracted much research, with many CNN models having been introduced in the last decade. For instance, the top two architectures in the ILSVRC the year after AlexNet won, Visual Geometry Group (VGG) [SZ14] and GoogLeNet [Sze+15] introduced notable concepts: the idea of shrinking spatial dimensions and increasing depth for VGG and the introduction of the inception module, which was later used by R-CNN [Gir+14; Ren+15] for GoogLeNet.

Today, two other CNN families of models are widely used as benchmarks for image classification: Residual Networks [He+16] and the MobileNet family of models [How+17; San+18; How+19], in particular MobiletNet-V2 [San+18]. Since they are extensively used in our experiments, we briefly overview them next.

- **Residual Networks** [He+16], frequently shortened to ResNets, introduced a residual learning block through an identity mapping shortcut connection, facilitating the training of much deeper models (with numbers of layers in the hundreds) than previously possible. In ResNets, each residual block contains a BN operation, a ReLU activation function, and a convolutional layer. This is followed by the same sequence before being added back to the input, as illustrated by Figure 1.10. Although the names might be misleading, the ResNet-20 architecture described in the original paper and targetted towards the CIFAR-10 dataset is smaller than the ResNet-18 one, which is intended for ImageNet-1K classification, with 0.27 million versus 11 million parameters, respectively. This can be explained by the deeper residual blocks of the ResNet-18 model.

- The **MobileNet** family of models consists of several efficient CNN architectures, with the first version developed by Google in 2017. The defining feature of these networks is the use of *depthwise separable convolutions* [MG12] that significantly

Figure 1.11: Semantic segmentation applied to ship detection. Class 0 ( ) corresponds to background pixels, and class 1 ( ) to ship pixels.

reduce the computation complexity and the number of parameters. A depthwise separable convolution decomposes a regular convolution into two separate, simpler convolutions: *depthwise convolution* and *pointwise convolution*. Depthwise convolution is configured with the number of groups equivalent to the number of input channels, while pointwise convolution denotes a traditional convolution operation employing a kernel size of $1 \times 1$. MobileNet-V2 is probably the most widely used inside benchmarks among the different versions.

### 1.5.2 Semantic Segmentation

Semantic Segmentation consists of assigning a label to every pixel in the image. This can extract fine-grained information and objects from imagery data. Figure 1.11 illustrates semantic segmentation applied to the problem of ship detection.

**The Airbus Ship Dataset.** A wide variety of tasks require semantic segmentation of satellite images, including cloud [MS19] and forest fire [Alm+21] detection, or land-surface processes [JWL18]. In this work, we consider ship detection since it is fairly simple to state (*i.e.,* a binary problem with only two classes), but still challenging because of the presence of small details on ship objects. Most of the work described in this thesis can be applied to other satellite image processing workloads requiring segmentation tasks. We use the Airbus Ship Dataset, consisting of $768 \times 768$ pixels RGB satellite images of the Earth, mainly seas and coasts. This dataset was introduced as part of a Kaggle

competition in 2018[5] to improve the detection of ships from satellite images in the context of maritime traffic monitoring. The dataset comprises a labeled training set and a test set. Labeling is done through a CSV file containing the image ID, the positions, and the number of consecutive ship pixels. The test set can only be used in the context of the competition as the labels are not provided, so we will not consider it further here.

Although instance segmentation[6] is probably the most intuitive choice for this task today. We chose to use semantic segmentation since it holds practical interest for our CNES partners, and it was the most commonly used approach at the time of the competition and performed well for this task, including for the winners. Nevertheless, what will be presented in this manuscript should also be relevant in designing compressed DNN models targeted towards instance segmentation tasks.

**Dataset Analysis** The training data is composed of $192,555$ labeled images, including $42,555$ images with ships and $150,000$ empty images, which represent a 1:857 ship to non-ship pixel ratio. As it stands, the training dataset is very unbalanced, so we decided to remove $130,000$ background images, resulting in a more balanced 1:278 ratio. We split the remaining $62,555$ images into 80% for training (50,043 images) and 20% for validation (12,512 images). There is also a huge disparity in the ship sizes. Small ship detection [Zha+19], as well as inshore [Nie+18] detection, are challenging topics, both subject to active research. In the following sections, we will refer to these two subsets when referring to the Airbus dataset.

**Accuracy Metrics.** The accuracy of semantic segmentation models is measured by the ratio between the correctly segmented area and the ground truth. A very simple metric, known as pixel accuracy, calculates the ratio between the amount of adequately classified pixels and their total number. However, this metric does not consider False Positives (FPs), which makes it less informative. The most commonly used metric is known as Intersection over Union (IoU) [UA19]. This statistical metric computes the ratio between the number of true positives (intersection) over the sum of true positives, false negatives, and false positives (union), illustrated[7] by Figure 1.12. For multi-class segmentation tasks, we use the mean Intersection over Union (mIoU), which is the average

[5] Kaggle Airbus Ship Detection Challenge (July 2018): https://www.kaggle.com/c/airbus-ship-detection

[6] Instance segmentation associates each pixel with an instance/object belonging to a class. Image segmentation on the other hand just associates with each pixel the class it belongs to, without differentiating between different instances from the same class.

[7] Image credit: https://wikidocs.net/177706


Ground Truth Mask | Predicted Mask | IoU Components

Figure 1.12: Illustration of the different components used to compute the IoU metric and, in particular, the True Positives (TPs), False Negatives (FNs), and FPs elements.

IoU computed per class and defined as:

$$\text{mIoU}_k = \frac{1}{N_c} \sum_{i=1, i \neq k}^{N_c} \frac{TP_k}{FP_{k,i} + FN_{k,i} + TP_k},$$

where $TP_k$ is the number of true positive pixels for class $k$, $FP_{k,i}$ and $FN_{k,i}$ are the number of false positive and false negative pixels for class $k$ respectively, and $N_c$ is the total number of classes.

In the context of the Airbus ship dataset, since it is a binary task, we will always refer to the ship class when mentioning the model accuracy.

**Neural Network Architectures.** Moving from conventional methods such as thresholding, clustering, or region growing, the utilization of DL, more specifically that of Fully Convolutional Network (FCN) models, was a breaking point for segmentation tasks. The first FCN segmentation model proposed upsampling the output activation maps from which the pixel-wise output is calculated [LSD15]. The same year saw the introduction of a new model family called *encoder-decoder* networks [NHH15]. The role of the decoder network is to map the low-resolution encoder feature to full input resolution feature maps for pixel-wise classification. One of the most popular models in this family is the U-Net [RFB15] architecture. This model consists of a contracting path that captures context and a symmetric expanding path that enables precise localization. It introduced skip connections to the encoder-decoder image segmentation networks, which improved the model's accuracy and addressed the problem of vanishing gradients. A few years later, the DeepLabV3+ model combined the advantages of both dilated convolutions and fea-

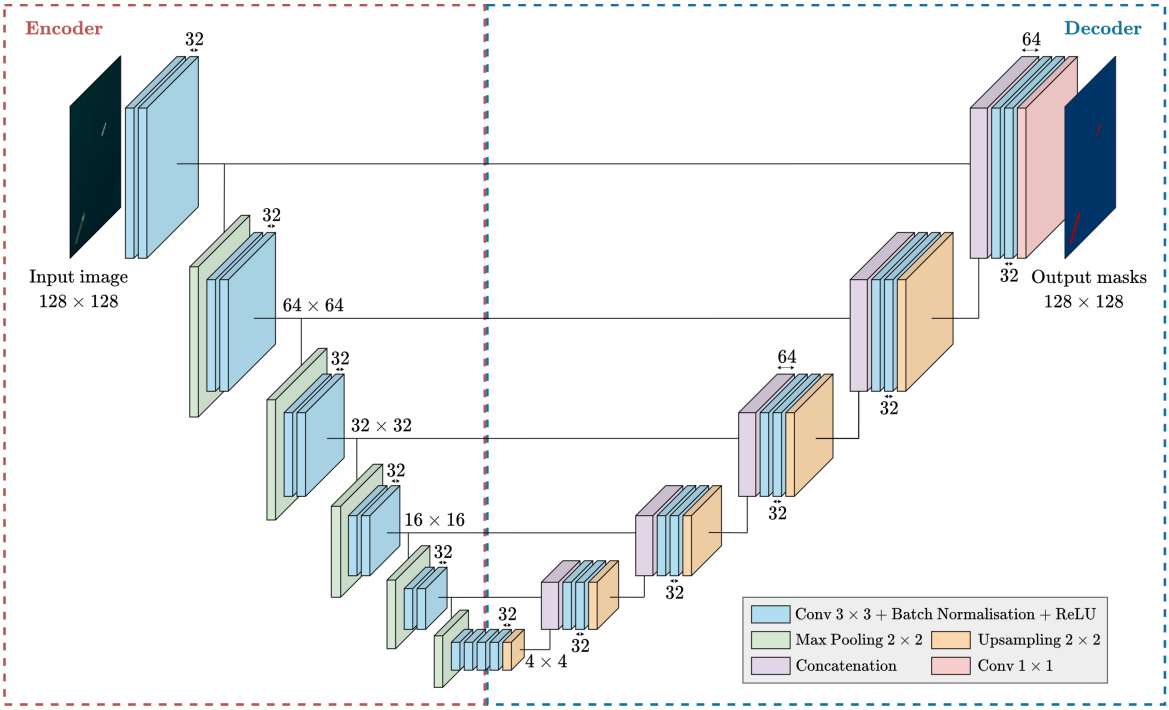



Figure 1.13: The Thin U-Net 32 architecture consists of a 5-stage encoder, followed by a 5-stage decoder, with skip connections between each corresponding stage pair in the encoder-decoder blocks.

ture pyramid pooling [Che+18a]. Although these models achieve very high accuracy, their speed and memory requirements remain a significant limitation for their deployment on embedded systems. For instance, the U-Net model is composed of 31M parameters for a size of 288MB. This is why lightweight models inspired by U-Net have emerged [Meh+18; VXN20; BJ20; Ans+22].

In our case, we use a Thin U-Net [VXN20] model (see Figure 1.13), a smaller version of the U-Net architecture. More specifically, our version is a Thin U-Net 32 model, where 32 refers to the number of channels for each convolution. Its memory size is smaller by a factor of 290 compared to a standard U-Net architecture. The smaller size has little impact on accuracy, making it an interesting candidate for embedded deployment.

Chapter 2

# Deep Neural Network Quantization for Inference Acceleration

Training a DNN model requires a large amount of data and significant computing resources. In addition, the training stage can take up to multiple days or weeks, which is why training is typically carried out in the cloud. Inference, on the other hand, while mostly done in the cloud, can also be performed on edge devices such as mobile phones, wearable devices, or satellites. Running DNN inference on-device addresses several concerns, including privacy, security, latency, and limitations in communication bandwidth. However, modern DNNs contain at least millions of parameters and require billions of arithmetic operations. In many cases, memory and computational costs make deployment on embedded devices difficult, if not infeasible. To mitigate these issues, recent research has focused on neural network compression. Various techniques, such as pruning, weight sharing, distillation, and quantization, can be used to reduce the computational intensity to make it compatible with on-device processing. With the emergence of hardware platforms offering better support for low (*e.g.,* recent Nvidia GPUs) and custom precision computing, quantization is at the forefront of methods used to increase the efficiency of DNN model inference [Dup+22].

This chapter introduces the basic concepts of NN quantization that will facilitate the comprehension of the work carried out in this thesis. In Section 2.1, we briefly review various NN compression methods with an emphasis on quantization. Then, in Section 2.2, we describe several quantization formats that are widely used in ML, including floating-point formats and affine quantization for integer mapping. In Section 2.3, we detail the procedures used to quantize neural network models that will be used in later chapters.

## 2.1 Deep Neural Network Compression Methods

The millions of parameters constituting a neural network are generally stored in single precision (i.e., 32-bit floating-point). This large format has a non-negligible impact on computing resources and energy consumption. In this section, we discuss different approaches to dealing with this issue.

### 2.1.1 Quantization

Quantization is the process of mapping input values in a large (often continuous) set to output values in a smaller finite set. This process is an old concept that was used long before being applied to NNs. The oldest mention of quantization is rounding off, which was first used to estimate densities by histograms [She97]. It was not until 1948, with the advent of computers, that the effect of quantization and its use in coding theory started being explored [Sha48]. Since then, quantizer design algorithms have been developed and tested for speech, images, video, and other signal sources. For example, in computer graphics, color quantization is a process that reduces the number of distinct colors used in an image, usually with the intention that the new image should be as visually similar as possible to the original image (see Figure 2.1 for an illustration). More recently, the scalability issues that plague modern DNNs have brought a lot of interest in quantization techniques. NN quantization emerged as a field of study in the 90s, during a resurgence of NN research [Ham90; HF91; HH93]. At the time, one of the main limitations to developing NNs was the long training time on the available hardware. Therefore, the main motivation for quantization was to reduce bit-widths to speed up training. With the increasing popularity of GPUs for training in the early 2010s, quantization has become more mainstream, especially as the model depth continues to increase every year [Xu+18]. Both training and inference of large models are computationally intensive, making efficient representation of numerical values particularly important. Moreover, most modern DNN models are heavily over-parameterized [Den+13], so there is ample opportunity for reducing bit-width without impacting accuracy. Most DNNs show remarkable robustness to quantization.

However, low bit-width quantization introduces noise in the network, potentially leading to a drop in accuracy. While some networks are robust to such noise, others require extra work to exploit the benefits of quantization. The rest of this chapter highlights core quantization topics such as number representations & quantization formats and the DNN

Figure 2.1: Effect of color quantization with a 24-bit image on the left and a 4-bit image on the right. credit: https://en.wikipedia.org/wiki/Color_quantization

quantization methodologies of *post-training quantization* and *quantization-aware training* that will be of interest in later chapters. For a more comprehensive overview and other details, we recommend two excellent DNN quantization surveys [Gho+22; Nag+21].

### 2.1.2 Other Compression Methods

Quantization can often be applied in conjunction with other methods for DNN compression. For completeness, the rest of this section briefly reviews such methods.

#### Pruning

Pruning reduces model complexity by removing redundant connections or neurons that do not significantly contribute to accuracy, resulting in a sparse computational graph. The pruning concept is inspired directly by the human brain, mimicking the natural process of synaptic pruning observed in neuronal development [LDS89]. Pruning techniques can be broadly categorized according to structure granularities, from *unstructured* to *structured* pruning. In unstructured pruning, individual weights are pruned independently [Han+15; FC18; LAT18; Tan+20], whereas, in structured pruning, a group of weights is removed at once, for instance, in a channel [Mol+16; Liu+17a; HZS17] or filter [Mol+19; AIZ19] wide level. Structured pruning often results in higher accuracy loss compared to unstructured pruning. To improve the flexibility of structured pruning and lower the accuracy drop, less aggressive methods, sometimes called semi-structured pruning, can be used, such as pattern-based pruning [Ma+20; Niu+20]. Model pruning is usually carried out either by

zeroing out weights lower than a certain threshold (magnitude-based methods) [Han+15; JV22] or by using a regularization term in the loss function (regularization-based methods) [LL16; HZS17; Wan+19a].

Pruning requires dedicated hardware architectures that can handle sparsity to take advantage of the resulting compression. Although pruning can be used in conjunction with quantization, we did not consider it because of the specific hardware required.

## Knowledge Distillation

Knowledge Distillation is a compression method that mimics the behavior of a large DNN model, called the teacher, into a smaller, lightweight model, called the student, by transferring learned knowledge. In the context of image classification, instead of using class labels during the training of the student model, the key idea is to leverage the probabilities produced by the teacher, as these probabilities can contain more information about the input. The concept of knowledge distillation [BCN06] was first introduced as a method for transferring information from a large model to train a small model. It can be applied at various points in the network, such as for activations [Heo+19], neurons [HW17], or features of intermediate layers [Rom+14]. Knowledge transfer was later used in semi-supervised learning to transfer information between a fully supervised teacher and student models using unlabeled data [USB11]. This method was later augmented with the student model being trained to imitate the class probabilities of the softmax output of the teacher model [HVD15]. Since then, much work has been carried out to improve knowledge distillation [Li+17; PPA18; Ahn+19; AL20b; Gou+21; AAA21].

Although knowledge distillation can improve the effects of quantization [Kim+19; SBS20; Boo+21], we do not consider this technique because of the significant additional training costs it requires, but also for a fairer comparison with state-of-the-art quantization methods, many of which do not employ knowledge distillation.

## Efficient Model Architectures

Searching for efficient DNN model architectures is another surefire way of reducing computation time and the memory footprint. To obtain a more compact model, it is possible to modify either its micro-architecture [How+17; Ioa+17; Cho17; San+18] (*e.g.,* using simplified convolutions such as $1 \times 1$ kernels or depth-wise convolutions in the case of CNNs), or its macro-architecture [Ian+16; Hua+17; Zha+18b; San+18] (*e.g.,* with resid-

ual connections or inception modules).

Appropriate network architectures are mostly found through manual search, which is often a time-consuming process of traversing a large search space. As such, significant effort has been invested into automated approaches for this task, now known as Neural Architecture Search (NAS) methods [CZH18; Cai+19; How+19; XCS23]. These aim to automate the search for new model architectures under given model size, depth, and/or width constraints. As discussed in Chapter 1, we consider such efficient models in our experiments, particularly the MobileNet-V2 family of models for image classification and the Thin U-Net 32 architecture for semantic segmentation.

### Weight Sharing

Weight sharing exploits redundancy in the model parameter space by identifying clusters of weights that can take the same value, reducing the number of unique weights. The first weight-sharing approach used the K-means clustering algorithm [Ja79] to group weights and assign the cluster centroid values to each weight within a group [Gon+14]. K-means is an unsupervised learning clustering algorithm that achieves iterative grouping of the samples using a distance computed between the samples and their centroids. It has been used in many approaches from the literature [Raz+17; Wu+18b; SNL18; Dup+20]. Some variations can be observed, such as investigating multiple cluster initialization techniques [HMD15] or minimizing the accumulated error across multiple layers to improve the centroid selection [Wu+16]. Other approaches have been used to group weights, including random-grouping [Che+15], mixture components [UMW17], dynamic programming [Yan+20] or, more recently, dictionary-based learning techniques [Pik+22].

Like pruning, weight sharing requires specific hardware architectures to take advantage of the resulting compression, such as Huffman coding-based accelerators [HMD15]. Weight sharing can also be seen as a form of quantization, with the number of clusters corresponding to the number of quantization levels.

## 2.2 Number Formats for Deep Neural Network Quantization

Real numbers are central in AI and ML models, being used to represent feature values, weights, and biases, facilitating pattern recognition and decision-making. The represen-

tations of real numbers in computer systems can be divided into two families: *uniform* and *non-uniform* formats. For uniform formats, as the name suggests, numbers are uniformly spaced and consist of integer-based representations. In contrast, the distances between the numbers vary in non-uniform formats, including the most used class of formats: floating-point representations. Additional non-uniform number representations, such as logarithmic [Joh18; CTN18; CDP22] and block floating-point [Kös+17; Fan+19; Lia+19] or posits [ZHK19; Lan+20; RTR21], have been explored in the literature but, in this thesis, we have focused only on uniform and floating-point formats, mainly due to their wider and more far-reaching availability.

### 2.2.1 Affine Quantization

Integer formats are widely used for efficient ML inference. 8-bit integer formats are now widely used to reduce model size without any significant accuracy drop. Most current research is focused on reducing the bit-width to sub-8-bit levels while limiting the impact on accuracy. In particular, there are some approaches targeting 4-bit integers that show very little to no performance degradation on certain models.

Affine quantization is a popular scheme that maps floating-point values to lower-precision integer counterparts. It allows efficient implementation of fractional numbers by using scaled and shifted integers. This format type is sometimes referred to as integer quantization since it involves integer numbers in actual computations (for instance, matrix multiply operations). Fixed-point arithmetic is also a special case of affine quantization [SM22]. Affine quantization comes in two variants, asymmetric and symmetric, which we detail next.

#### Integer Arithmetic

In a binary number system, integers are commonly represented as a vector of bits, with each element having twice the weight of the previous one. In this manner, an $N$-bit positive integer is expressed as:

$$\hat{x} = \sum_{i=0}^{N-1} b_i \cdot 2^i, \tag{2.1}$$

where $\hat{x}$ is an integer and $b_i \in \{0, 1\}$ is the binary value at index $i$.

One way to extend this representation to negative integers is using a *sign-and-modulus* encoding, *i.e.,* use one bit $s$ to represent the sign, generally the most significant one, and

the remaining $N-1$-bit to store the magnitude of the integer. The encoding corresponds to:

$$\hat{x} = (-1)^s \cdot \sum_{i=0}^{N-2} b_i \cdot 2^i.$$

However, it is possible to use a more clever representation called *2's complement*, which allows only one representation for zero and gains an extra encoded number:

$$\hat{x} = -b_{N-1}2^{N-1} + \sum_{i=0}^{N-2} b_i \cdot 2^i.$$

## Asymmetric Affine Quantization

Asymmetric affine quantization is characterized by three parameters: the *scaling factor*, the *zero-point*, and the *bit-width*. The scaling factor specifies the step size of the quantizer values, and the zero-point ensures zero is correctly encoded while the bit-width determines the size of this grid. The asymmetric affine quantization function, according to these three parameters, is defined as

$$\hat{x} = \left\lfloor \frac{x}{s} \right\rceil + \hat{z}, \tag{2.2}$$

where $x \in \mathbb{R}$ is the floating-point number to quantize, and $\hat{x} \in \mathbb{Z}$ its mapped integer, $s \in \mathbb{R}$ is the scaling factor, $\hat{z} \in \mathbb{Z}$ is the zero-point, and $\lfloor \cdot \rceil$ indicates rounding to the nearest integer. The scaling factor $s$ and zero-point $\hat{z}$ are determined according to the bit-width and the dynamics of values, such as

$$s = \frac{\max(\mathbf{X}) - \min(\mathbf{X})}{2^N - 1},$$

$$\hat{z} = -\left\lfloor \frac{\min(\mathbf{X})}{s} \right\rceil,$$

where $\mathbf{X}$ is a matrix/tensor of floating-point numbers, and $N \in \mathbb{N}_+^*$ is the bit-width.

A clamping function is commonly used to ensure that values outside the quantization range will be clipped to the range limits. The clamping function is defined as

$$\texttt{clamp}(x, \alpha, \beta) = \min(\max(x, \alpha), \beta) = \begin{cases} \alpha, & x < \alpha, \\ x, & \beta \leq x \leq \alpha, \\ \beta, & x > \beta, \end{cases} \tag{2.3}$$

where $\alpha$ and $\beta$ are the lower and upper bounds, respectively. Hence, the boundary values used in the clamping function determine whether the quantization format is signed or unsigned, such that

$$x_{\text{uint}} = \texttt{clamp}(\hat{x}, 0, 2^N - 1) \text{ and } x_{\text{int}} = \texttt{clamp}(\hat{x}, -2^{N-1}, 2^{N-1} - 1).$$

To recover the real quantized value $\tilde{x}$ from the integer value $\hat{x}$, an operation that is often referred to as *de-quantization* is used:

$$\widetilde{x} = s\left(\hat{x} - \hat{z}\right). \tag{2.4}$$

Note that the recovered real values $\widetilde{x}$ will not exactly match $x$ due to the rounding operation. The value $x - \tilde{x}$ is called the *rounding error*.

By combining Eq. (2.2),(2.3) and (2.4), the general definition for the asymmetric affine quantization function is

$$\widetilde{x} = s\left[\texttt{clamp}\left(\left\lfloor\frac{x}{s}\right\rceil + \hat{z}, 0, 2^N - 1\right) - \hat{z}\right]. \tag{2.5}$$

The clamping function introduces a *clipping error*. There is usually a trade-off. Reducing the clipping error is possible by increasing the scaling factor, which expands the quantization range. However, increasing the scaling factor increases the rounding error.

## Symmetric Affine Quantization

Symmetric quantization is a simplified version of asymmetric quantization with the zero-point set to zero, thus reducing the computational overhead of dealing with zero-point offset. Consequently, (2.5) simplifies to

$$\widetilde{x} = \texttt{q}(x) = s\left[\texttt{clamp}\left(\left\lfloor\frac{x}{s}\right\rceil, \alpha, \beta\right)\right]. \tag{2.6}$$

However, the lack of an offset restricts the mapping between integer and floating-point data. Therefore, the signed or unsigned integer grid must be chosen carefully. Typically, unsigned symmetric quantization is well suited for ReLU hidden activations and signed symmetric quantization for weights quantization.

The scaling factor must also be adjusted accordingly, leading to

$$s_{\text{int}} = \frac{\max(|\mathbf{X}|)}{2^{N-1}-1} \text{ or } s_{\text{uint}} = \frac{\max(|\mathbf{X}|)}{2^{N}-1}.$$

### Power of Two Scaling Factors

Using a power of two scaling factor corresponds to fixed-point arithmetic [CBD14; SM22]. In this case the scaling factor $s$ is defined as

$$s = 2^{\left\lceil \log_2\left(\frac{\max(|\mathbf{X}|)}{2^{N-1}-1}\right)\right\rceil}.$$

This format provides better hardware efficiency as scaling to a power of two can be handled with a simple bit shift. However, the restricted expressiveness of the scaling factor introduces additional quantization error, sometimes resulting in a loss of accuracy.

## 2.2.2 Binary and Ternary Quantization

Going to the extreme, it is sometimes possible to quantize weights and activations using one and two bits encodings, aka *binary* and *ternary* quantization. Previous work has shown that this is a promising approach to better take advantage of bit-wise operations, which are extremely cheap to compute in hardware.

### Binary Quantization

The concept of binarization was first introduced in BinaryConnect [CBD15], which constrains the weights to either $-1$ or $+1$. In this approach, the weights are kept as real values during the quantized training process and are only binarized during the forward and backward passes to simulate the binarization effect. The real-valued weights are converted and stored as $\pm 1$ only after training has finished, using the sign function:

$$w_b = \texttt{sign}(w) \begin{cases} +1 & \text{if } w \geq 0, \\ -1 & \text{otherwise.} \end{cases} \tag{2.7}$$

Later, this concept was extended to both weights and activations, leading to the first full Binary Neural Networks (BNNs) [Hub+16]. The computationally heavy matrix multiplication operations can be replaced with light-weight bitwise XNOR and *bitcount* opera-

tions, such as:

$$x \cdot y = \texttt{popcount}(\overline{a_b \oplus w_b}), \tag{2.8}$$

where `popcount` counts the number of ones in the binary representation of the parameter.

Another major contribution is XNOR-Net[Ras+16], which achieves higher accuracy by incorporating a scaling factor to the weights and using $\pm\alpha$ instead of $\pm 1$. Here, $\alpha \in \mathbb{R}_+$ is the scaling factor chosen to minimize the distance between the real-valued weights and the resulting binarized weights. This same approach is extendable to activations, enabling the matrix multiplication simplification of Eq. (2.8), all while improving accuracy thanks to the use of scaling factors.

Since this pioneering work on BNNs, several solutions have been proposed to reduce the accuracy degradation introduced by extreme quantization. According to [Qin+20], such solutions can be broadly categorized as: *quantization error minimization* [LZP17; Mis+17; BT19], *improved loss function* [HYK16; Din+19], and *improved training method* [Dar+18; Liu+18; Gon+19b].

## Ternary Quantization

Despite all inroads made in binary NNs, they can still suffer from high accuracy loss. Inspired by the observation that many learned weights are close to zero, some works have looked at using a ternary set $\{-1, 0, +1\}$ of values to constrain the weights and the activations, thereby explicitly encoding the zero [Lin+15; Li+16a]. Ternary Neural Network (TNN) also drastically reduce the inference latency by simplifying costly matrix multiplications as binarization uses XNOR operators [Ale+17].

TWN [Li+16a] adopt the Euclidean distance to find the scaling factor and formats the weights into $-\alpha, 0$, and $+\alpha$ with a threshold generated by an assumption that the weights are uniformly distributed:

$$w_t = \begin{cases} +\alpha & \text{if } w > \Delta, \\ 0 & \text{if } |w| \leq \Delta, \\ -\alpha & \text{if } w < \Delta, \end{cases} \tag{2.9}$$

where $\Delta \in \mathbb{R}_+^*$ is the threshold parameter and $\alpha \in \mathbb{R}_+$ is the scaling factor.

TTQ [Zhu+16] extends the approach by learning both the scaling factor and the ternary assignments during training. They further introduce an asymmetric ternary format with positive $\alpha^+$ and negative $\alpha^-$ scaling factors. TNNs [Ale+17] extend TWN by

also quantizing activations into ternary values using knowledge distillation.

### 2.2.3 Floating-Point Arithmetic

Floating-point representations are based on the base-2 scientific notation, which is more suitable for the binary representation used in Computer Arithmetic than the more traditional base-10 encodings. This notation, for a nonzero real number $x$, is expressed as

$$x = (-1)^s \times M_x \times 2^{E_x},$$

where $s$ is the sign bit, the exponent $E_x$ is an integer, and the mantissa $M_x$ is a (quantized) real such that $1 \leq M_x < 2$. It is always possible to satisfy the condition $1 \leq M_x < 2$ by choosing an appropriate $E_x$. In 1985, to address the problem of floating-point support on computing machines and to simplify software compatibility, the IEEE-754 [IEE85] floating-point standard was ratified, with binary32 (or single precision) and binary64 (or double precision) formats becoming mainstays on computing platforms ever since. This standard has undergone two revisions in 2008 and 2019, and is still the reference for floating-point formats today.

The IEEE-754 standard defines formats for 16-bit, 32-bit, and 64-bit floating-point numbers. These formats are known as binary16 (`binary16`), binary32, and binary64 or, more colloquially, half, single, and double precision, respectively. Two of these formats, binary16 and binary32, are widely used for training neural networks. This is due to the long-term support available for these formats on the hardware used for training.

#### Normalized Numbers

In IEEE-754 formats, a non-zero floating-point value $X$ with $m$-bit of mantissa and $e$-bit of exponent is represented, in binary notation, as

$$X = (-1)^s \times 1.\underbrace{x_1 \dots x_m}_{M_X} \times 2^{E_X - E_B},$$

where $s$ is the sign bit, which is 0 for positive numbers and 1 for negative numbers, $M_X \in [0, 1)$ is the $m$-bit fractional mantissa, $E_X \in \{0, 1, \dots, 2^e - 1\}$ is the integer exponent and $E_B = 2^{e-1} - 1$ is an integer exponent bias term. The exponent offset-binary representation ensures the representation of numbers smaller than zero as it covers an almost symmetric

range of positive and negative values. Numbers that can be written in this representation are called normalized numbers. We note that a normalized number's leading bit $x_0$ is said to be hidden, *i.e.,* this bit is not stored. This allows the precision to increase by one bit without expanding the bit-width.

## Precision and Range

The precision of floating-point systems is defined by the number of bits in the mantissa, including the hidden bit. For example, in binary32, the precision $p$ is 24, with 23 stored bits in the fractional part of the mantissa and one leading hidden bit. At the same time, the dynamic range is expressed by the number of bits in the exponent, or, in other words, how wide our representable number range will be. Any fixed bit-width floating-point number system must make a trade-off between the dynamic range of representable values ($e$) and the precision ($m$).

## Zero Encodings and Subnormal Values

The number zero is special since it cannot be normalized in the same way. The approach in the standard is to use a special encoding when the exponent value is set to 0 to signal that the number is subnormal or zero. In this case, the exponent value is implicitly set to 1, and the previous expression becomes

$$X = (-1)^s \times 0.\underbrace{x_1 \dots x_m}_{M_X} \times 2^{1-E_B}.$$

In the standard, zero is signed, meaning that there exists both a positive zero (+0) and a negative zero (−0). The two values behave as equal in numerical comparisons, but some operations return different results for +0 and −0. For instance, $1/(-0)$ returns negative infinity whereas $1/(+0)$ positive infinity.

Besides allowing the exact representation of 0, the standard extends the range below the smallest normalized number using subnormal numbers. Subnormals fill the gap between the smallest normalized number and zero, leading to a smooth rounding towards zero. They can be useful when training DNN models, where it is common to represent near-zero values for gradients. Subnormals can also be helpful to represent random values pulled from certain distributions. For example, model weights are often initialized to small random values at the start of training.

### Exceptions

The IEEE-754 standard also defines multiple exceptions, including invalid operation, division by zero, overflow, or underflow, and a standard response to each. The standard specifies that an exception must be signaled by setting an associated status flag such as Not-a-Number (NaN) or Infinity (Inf).

**NaN and Inf** The response to an invalid operation, usually when infinity and zero are involved, such as $0/0$ or $\infty \times 0$, is to set the result to NaN. This exception is helpful during training to debug code running on accelerator hardware. The appearance of Inf occurs when there is a division by 0 or an overflow and is not usually treated as an error value but similar to any ordinary numerical value. These two exceptions are encoded when the exponent is set to the upper limit $E_{\max}$. Infinities correspond to a zero mantissa and are signed according to the sign bit. The rest of the range is reserved for NaN encodings, *i.e.,* when the mantissa is not zero. Thus, according to the Institute of Electrical and Electronics Engineers (IEEE) standard, there are $2^{m+1} - 2$ NaNs encoded, *e.g,* $16,777,214$ for binary32, compared with one positive and one negative infinity.

**Overflow and Underflow** An overflow occurs when the exact result of a floating-point operation is finite but so big that its correctly rounded value is larger than the largest representable floating-point number $N_{\max}$. The standard response to overflow is to provide the correctly rounded result, either $\pm N_{\max}$ or $\pm\infty$, depending on the rounding mode. An underflow occurs when the exact result of an operation is non-zero but with an absolute value smaller than the smallest normalized floating-point number $N_{\min}$. The standard response to underflow is to return the correctly rounded value, which may be a subnormal number, $\pm 0$, or $\pm N_{\min}$. Subnormal numbers fill the relatively large gap between $\pm N_{\min}$ and zero, leading to what is known as gradual underflow.

### 2.2.4 Low-Precision Floating-Point Formats for Deep Learning

In recent years, smaller floating-point formats have emerged, mainly for neural network applications [SM22]. First introduced to reduce training complexity, these formats are becoming increasingly popular for accelerating inference, with Nvidia being one of the prominent figures in this regard. Figure 2.2 illustrates the different formats which are further detailed in this section.

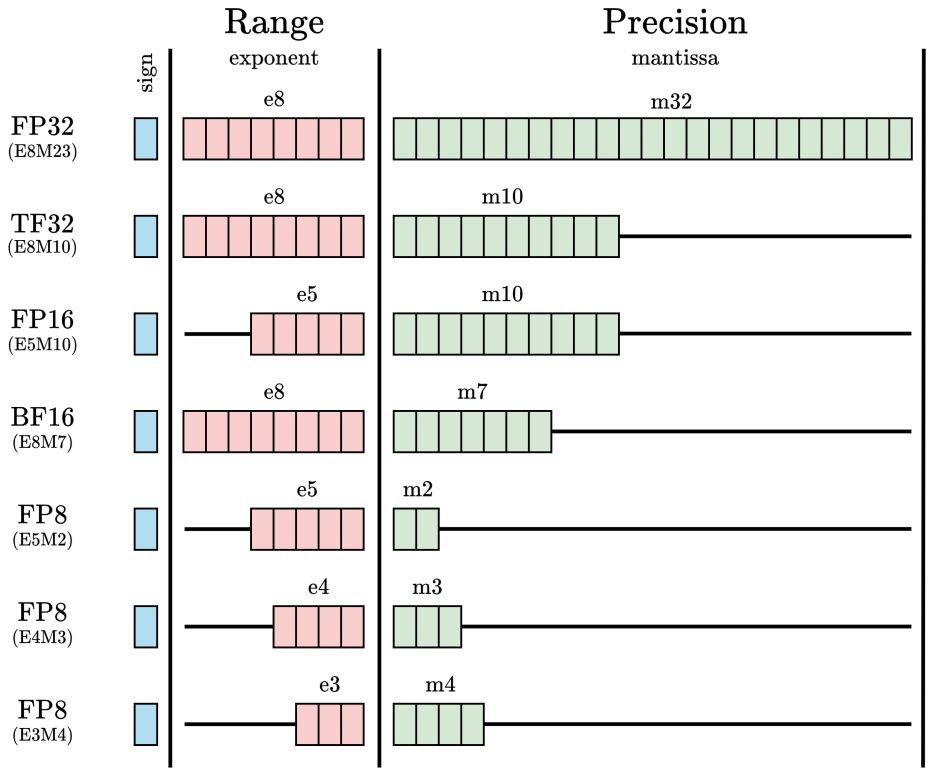



Figure 2.2: Comparison of different floating-point formats used in ML. We use the E$e$M$m$ notation to designate a floating-point format with $e$-bit of exponent and $m$ mantissa bits.

## binary16/BF16

binary16, or `binary16`, has been defined in the 2008 revision of the IEEE-754 standard [IEEE-754]. It follows the same specifications as binary32 with an implicit leading one, subnormal support, NaN and Inf implementation, and an exponent offset-binary representation. binary16 is encoded using 5 bits of exponents and 10 bits of mantissa. This format was initially designed as a storage image format that could handle a wide dynamic range without the memory cost of single or double precision [CA20]. Today, the binary16 format is widely supported by most hardware vendors, including ARM, AMD, Intel, and Nvidia.

BF16, or `bfloat16`, is more recent and was first introduced as part of distributed training frameworks DistBelief [Dea+12] and Tensorflow [Aba+16] as a low-precision storage format before becoming an alternative format to binary16 for training acceleration. As opposed to binary16, BF16 is encoded using 8 bits of exponents and 7 bits of mantissa.

BF16 is therefore designed to maintain the numeric range of binary32, providing a wider dynamic range than binary16, at the expense of precision, illustrated in Figure 2.2. It also enables faster conversion from and to binary32, compared with binary16, by simply truncating the significant field. Hardware support is available in Google TPUs since v2, Nvidia GPUs since the A100, ARMV8, and modern Intel Central Processing Units (CPUs).

Half-precision binary16 floating-point number formats and BF16 are mainly used for DNN training acceleration, with numerous studies showing that 16-bit formats are sufficient for DNN convergence [Mic+17; Kal+19; Hag+19]. Both formats have not received much attention for inference on edge devices, with existing works limited to low-power CPUs [Tor+22] or embedded GPUs [Ji19]. This is due to the fact that lower bit-widths are usually sufficient in inference settings.

### TF32

TF32, or `tfloat32` despite its name, is a floating-point-based format with a 19-bit bit-width that was first introduced in the Nvidia Ampère architecture. TF32 adopts the same 8-bit exponent as binary32, so it can support the same numeric range, and uses the same 10-bit mantissa as binary16. TF32 is generally used in convolutions and matrix multiplications to convert their 16-bit inputs to TF32 right before multiplication, all memory storage and other operations remaining completely in binary32. This format is used to speed up training and is only available in Nvidia GPUs.

### binary8

Recently, the idea of using 8-bit floating-point formats for neural network training has gained interest from the Deep Learning community, with Nvidia, Arm, and Intel in the lead [Mic+22]. The IEEE P-3109 working group is investigating specifications for standardizing binary8 formats for ML [P3109]. Different binary8 formats were identified as potential candidates for DNN training and inference[1]: E5M2[2] (binary8p3), E4M3 (binary8p4), and E3M4 (binary8p5). Unlike binary16 or binary32, IEEE binary8 formats use a single bit-sequence to represent NaN at the expense of the negative zero ($-0$) and an exponent bias of $2^{e-1}$. For $\pm\infty$ and subnormal values, binary8 formats follow IEEE-like encoding rules. Specifications of the different formats covered, including binary32,

[1] As of September 2024.
[2] E$e$M$m$ means $e$-bit and $m$-bit to represent exponent and mantissa, respectively.

| | binary32 | BF16 | binary16 | E5M2 | E4M3 | E3M4 |
|---|---|---|---|---|---|---|
| Exponent Bias ($E_B$) | 127 | 127 | 15 | 16 | 8 | 4 |
| Max value | $3.4028235 \times 10^{38}$ | $3.40 \times 10^{38}$ | $65,504.0$ | $49,152$ | 224 | 15 |
| Min value | $1.4 \times 10^{-45}$ | $1.18 \times 10^{-38}$ | $5.96 \times 10^{-8}$ | $8 \times 10^{-6}$ | $9.77 \times 10^{-4}$ | $7.813 \times 10^{-3}$ |
| Subnormals | Yes | No | Yes | Yes | Yes | Yes |
| NaN | all | all | all | single | single | single |
| Infinity | Yes | Yes | Yes | Yes | Yes | Yes |

Table 2.1: Configuration of different floating-point binary formats: The E$e$M$m$ notation represents bit allocation for Exponent ($e$) and Mantissa ($m$) respectively. All binary8 formats listed here follow the specifications of the IEEE working group P-3109. All formats shave a sign-bit and an implicit leading bit in the mantissa.

binary16, and BF16, are summarized in Table 2.1.

As with half-precision, binary8 formats were first discussed and designed to make neural network training more efficient. Initial studies focused on the E5M2 format for training tasks due to its wider dynamic range [Wan+18; Mel+19a], which is necessary for representing gradient values, followed by a hybrid training approach using E4M3 and E5M2 formats for the forward and the backward paths, respectively [Sun+19].

However, binary8 has attracted significant interest for inference acceleration from the outset. binary8 quantization schemes can be divided into two distinct approaches: calibration and scaling. Calibration is generally associated with Post-Training Quantization algorithms since it consists of batch normalization calibration and, more specifically, readjusting pre-trained model statistics (mean and variance parameters). By theoretical analysis of quantization errors, it has been demonstrated that batch normalization statistics could be fine-tuned to increase the accuracy of quantized models [Sun+19] and can be combined with numerical range scaling techniques [She+23] to fully recover binary32 baseline model accuracy. Normalizing activations can also help reduce quantization error, thus avoiding a calibration phase [Wu+20].

Scaling is a widely used strategy in affine quantization to increase accuracy by shifting the numerical range (refer to Section 2.2.1 for affine quantization scaling features). For binary8 quantization, simple tasks can be carried out successfully without scaling. However, more complex tasks require rescaling to maintain accuracy. Like with integer-based formats, a real scaling factor $\alpha$ can be used [Set+18; Met+21] to scale the numerical range. An equivalent approach moves the scaling to the exponent bias $\pm 1.M_x \times 2^{E_x - \beta}$ [Kuz+22]. A more hardware-friendly inference scaling approach consists of an integer exponent bias, which corresponds to allowing only power of 2 scaling factors [Sun+19; Tam+20; HCH21;

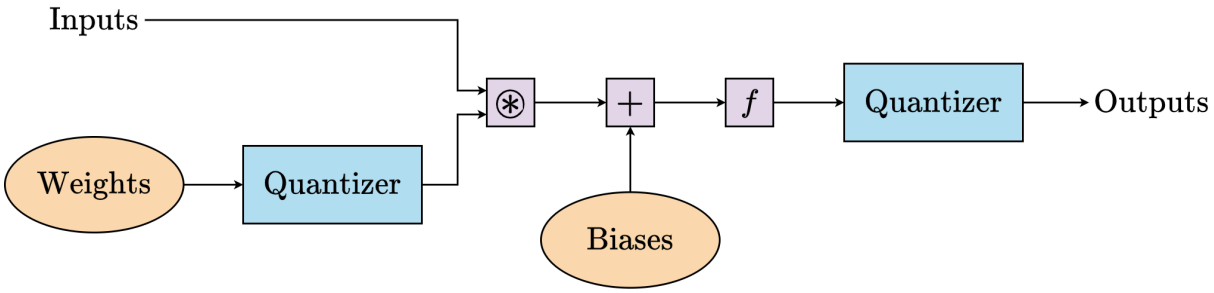



Figure 2.3: Schematic overview of a quantized forward pass for a convolutional layer. Quantizer blocks are added to the standard convolution stream to simulate the operation's performance when using a quantization format.

Nou+22].

In terms of actual availability, Nvidia has support for two binary8 formats (E5M2 and E4M3) in their Transformer engine software for the new Hopper architecture GPU [3]. These formats are described in [Mic+22]. Both have subnormal value support, but the E5M2 format follows IEEE-754 conventions with the same special values, whereas the E4M3 format includes only one mantissa encoding for NaNs and no infinities. Specific hardware such as ASIC [Wu+20] or FPGA [Met+21] custom processors have been designed to further address the problem of hardware efficiency caused by floating-point adders by introducing a hybrid Multiply-ACcumulate (MAC) with a floating-point multiplier and a fixed-point accumulator.

## 2.3 DNN Quantization Algorithms

GPUs are widely adopted for training DNNs, thanks to their computational parallelization capabilities. Although modern GPUs support multiple arithmetic formats, they are usually limited to the most common data types. Quantizer blocks are often introduced to *test* various quantization options and *simulate* the behavior of quantized models with non-standard formats, as illustrated in Figure 2.3. Quantizer blocks are applied to the weights to simulate weight quantization and at the output of activation functions to simulate activation quantization. Biases are often not quantized because they are stored in higher precision. The quantizer block implements a quantization function, and each quantizer is defined by a set of quantization parameters of the target format. The quantizer input

[3] https://www.nvidia.com/en-us/data-center/h100/

and output are usually stored in binary32, but the output belongs to the quantization grid. Adding such blocks makes it possible to approximate many formats using binary32 arithmetic while benefiting from the training acceleration provided by GPUs.

### 2.3.1 Quantization Procedures

Quantization generally requires changing DNN parameter values for it to be valid. This can either be performed by retraining the model, a process that is called Quantization-Aware Training (QAT), or done without re-training, a process that is often referred to as Post-Training Quantization (PTQ). The overhead of PTQ is generally low and often negligible, making these algorithms very effective and fast to implement as they do not require network retraining with labeled data. However, this often comes at the cost of lower accuracy, especially for low-precision quantization [Nah+21]. In these cases, one can resort to QAT, which finds more accurate solutions than PTQ. However, the higher accuracy comes with the usual costs of neural network training, *i.e.,* longer training times, the need for labeled data, and, possibly, hyper-parameter search.

#### Post-Training Quantization

Post-Training Quantization (PTQ) algorithms take a pre-trained single-precision network and use a small set of calibration data to help choose the best quantization parameters, *e.g.,* scaling factor. Multiple approaches have been proposed to mitigate the accuracy degradation of PTQ-based methods. A common issue is that quantization error is often biased, *i.e.,* the expected outputs of the original and quantized layer or network are different. The authors of [BNS19; FAG19] have observed an inherent bias in the mean and variance of the weight values following their quantization and introduce methods to correct for the expected shift in distribution. To overcome this problem, the authors of [BNS19] analytically compute an optimized clipping range. The authors of [Mel+19b; Nag+19] show that equalizing the weight ranges between different layers or channels reduces quantization errors. OMSE [Cho+19] proposes to minimize the $L_2$ distance between the quantized tensor and the corresponding floating-point tensor. AdaRound [Nag+20] shows that quantizing with Round-to-Nearest (RN) results in sub-optimal solutions. They propose an adaptive rounding method that better reduces the loss. Further calibration of PTQ weights using integer programming can also be used [Hub+20].

In certain cases, access to the original training dataset is not possible during the

quantization procedure. Various methods have been proposed to address this problem by introducing quantization approaches without calibration data. Synthetic data that does not take internal statistics into account may not properly represent the real data distribution [Har+20]. To address this, a number of subsequent efforts use the statistics stored in batch normalization layers, *i.e.,* channel-wise mean and variance, to generate more realistic synthetic data. The work in [Har+20] generates data by directly minimizing the Kullback–Leibler divergence of internal statistics, which is then used to calibrate and fine-tune the quantized models. ZeroQ [Cai+20] shows that the synthetic data can be used for sensitivity measurement and calibration.

## Quantization-Aware Training

Quantization-Aware Training (QAT) involves training a model with quantized parameters to learn and correct any quantization bias that often results from rounding and clamping errors. Although it is possible to train a quantized NN from scratch, applying fine-tuning to a pretrained model is more common as starting with optimized parameters generally reduces the time required to produce the quantized model. QAT methods stem from pioneering work on binary neural networks [CBD15; Cou+16]. At its core, a QAT method consists of using a quantized version of the network during training in both the forward and backward passes while performing updates on full-precision copies of the network parameters. These full precision parameters are then quantized to be used in the next iteration. A crucial aspect is how to perform back-propagation through quantized variables (parameters and activations). The quantization function defined in (2.6) is non-differentiable. In the binary case, the gradient be can approximated using a so-called Straight-Through Estimator (STE) [BLC13]. STE essentially ignores the rounding operation and approximates it with an identity function. STE has been extended to cover larger bit-widths while also applying quantization to gradient signals (the DoReFa-Net [Zho+16] approach). The corresponding backpropagation through (2.6) is done using STE, leading to the following rule:

$$\begin{aligned} \textbf{Forward:}\ & \widetilde{\mathbf{W}} = 2\,\mathrm{q}\left(f(\mathbf{W}) + \frac{1}{2}\right) - 1 \\ \textbf{Backward:}\ & \frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \widetilde{\mathbf{W}}} \frac{\partial \widetilde{\mathbf{W}}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \widetilde{\mathbf{W}}} \end{aligned} \tag{2.10}$$

where $\mathbf{W}$ is the unquantized weight tensor of the network, $\mathcal{L}$ is the loss function,

$$f(\mathbf{W}) = \frac{\tanh(\mathbf{W})}{2\max(|\tanh(\mathbf{W})|)},$$

and $\widetilde{\mathbf{W}}$ is the quantized version of $\mathbf{W}$. Using tanh in $f$ limits the range of weight values to $[-1, 1]$ before applying the quantization function $\mathtt{q}$ (the operation $f(\mathbf{W}) + 1/2$ leads to values in $[0, 1]$, with the final affine transformation scaling these values to $\widetilde{\mathbf{W}}$ in $[-1, 1]$).

PACT [Cho+18] proposes to learn the upper bound of a ReLU activation function to compute an appropriate scaling factor. The vanilla ReLU is thus replaced with

$$\text{PACT}(x) = \begin{cases} 0 & \text{if } x < 0 \\ \alpha & \text{if } x > \alpha \\ x & \text{otherwise} \end{cases} \tag{2.11}$$

where $\alpha \in \mathbb{R}_+$ is the learnable upper bound. The scaling factor in (2.6) is now $s = (2^N - 1)/\alpha$.

To further improve the accuracy of quantized DNNs, the STE idea can also be used to learn the parameters of affine quantizers, such as scaling factors and bias terms for weight quantization [Ess+19; Bha+20]. SAT [JYL19] investigates the gradient scales in training with quantized weights and further improves the model performance by adjusting weight scales based on the variance.

### 2.3.2 Quantization Granularity

The scaling factor can be adjusted at different granularities in CNNs, especially for convolutional layers. The scaling factor can be calculated for instance by considering all of the parameters in a layer. This approach is called *layerwise quantization* or *per-tensor quantization*. This is the most common choice of granularity due to its more straightforward hardware implementation. Each filter inside a convolutional layer can have a different range of values. As such, finer quantization granularity can be applied to improve performance further. A popular choice, known as *channel-wise quantization* or *per-channel quantization*, uses a scaling factor for each convolutional filter. This ensures a better quantization resolution and often results in higher accuracy while slightly increasing hardware costs. A less common approach called *group-wise quantization* groups multiple filters or channels of a layer to calculate the scaling factor [She+20]. This is helpful for cases where

the distribution of convolutions or activations varies significantly.

### 2.3.3 Improving the Quantization Emulation

So far, we mainly discussed convolution and dense layer quantization. However, a poor quantization of other layer types can cause an accuracy gap between the emulated format and the expected hardware performance. To improve the emulated format's performance, it is necessary to pay attention to the other layers that are part of a target DNN model.

#### Pooling Layers

The pooling algorithm determines whether outputs need to be quantized. For instance, the procedure is different between max pooling and average pooling. Max pooling layers do not need to be quantized, as the input and output values are on the same quantization grid. Conversely, average pooling layers must be quantized, as averaging integers does not necessarily result in an integer.

#### Unpooling Layers

**Interpolation** As with pooling layers, the interpolation algorithm determines whether or not output quantization is necessary. In the case of the nearest neighbor, quantizing outputs is superfluous as all the inputs are already quantized. On the other hand, quantizing outputs is necessary for bilinear interpolation.

**Transposed convolution** Like with standard direct convolutions, transposed convolutions also require a quantization step.

#### Batch Normalization

BN operations can be folded into the previous linear or convolutional layers during inference. This removes the batch normalization operations entirely from the network, as the calculations are absorbed into an adjacent dense or convolution layer. Besides reducing the computational overhead of the additional scaling and offset, this prevents extra data movement and the quantization of the layer's output. Thus, the simulation can be

improved by combining (1.3.1) with the previous layer, leading for instance to:

$$\mathrm{BN}(\mathbf{W}\cdot\mathbf{X}+\mathbf{b}) = \left(\frac{\gamma\mathbf{W}}{\sqrt{\sigma_{\mathcal{B}}^2+\epsilon}}\right)\cdot\mathbf{X} + \left(\beta - \frac{\gamma\left(\mathbf{b}-\mu_{\mathcal{B}}\right)}{\sqrt{\sigma_{\mathcal{B}}^2+\epsilon}}\right) = \widetilde{\mathbf{W}}\cdot\mathbf{X}+\widetilde{\mathbf{b}}, \tag{2.12}$$

where $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$ denote the mean and standard deviation of the minibatch, $\gamma$ and $\beta$ are two learnable parameters, and $\epsilon$ is a small constant to avoid division by zero.

### Skip Connections

Concatenation is the operation of joining feature maps end-to-end. The two concatenated branches do not generally share the same quantization parameters, i.e., their quantization grids may not overlap, making a new quantization step necessary. Similarly, element-wise addition of residual blocks must be adjusted to ensure inputs are on the same grid.

## 2.3.4 Rounding

There are many approaches to rounding a number, *e.g., Round Down* or *Round Up*, which round to the next smaller or larger integer, respectively. In this section, we will focus on explaining two commonly used rounding modes for DNN quantization: Round-to-Nearest (RN) and Stochastic Rounding (SR).

**Round-to-Nearest** consists of rounding to the nearest integer, as its name suggests. Rounding a number $x$ to the nearest integer requires some tie-breaking rule for those cases when $x$ is exactly halfway between two integers. In the case of a tie, a common rule that we also use is to pick the value with the least significant bit equal to zero (round to nearest ties to even). Given a real number $x$, RN is defined as

$$\mathrm{RN}(x) = \lfloor x \rceil = \begin{cases} \lfloor x \rfloor & \text{if } x - \lfloor x \rfloor < 0.5 \\ \lceil x \rceil & \text{if } x - \lfloor x \rfloor > 0.5 \\ \texttt{tie}(x) & \text{if } x - \lfloor x \rfloor = 0.5 \end{cases} \tag{2.13}$$

where $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ are the round down and round up operators, respectively, and `tie` is the tie-breaking rule.

**Stochastic Rounding** has recently been explored for DNN quantization training due to substantial improvements in reducing rounding errors in low-precision numerical formats, including affine [HF92; Gup+15], binary [CBD15] and binary8 [Wan+18; Mel+19a; BFS24] formats. SR differs from other rounding modes in that the rounding decision is not deterministic, and the probability of rounding up is proportional to the residue. Given a real number $x$, SR is defined as:

$$\mathrm{SR}(x) = \begin{cases} \lfloor x \rfloor & \text{if } P \geq 1 - \frac{x - \lfloor x \rfloor}{\epsilon} \\ \lfloor x \rfloor + \epsilon & \text{if } P < \frac{x - \lfloor x \rfloor}{\epsilon} \end{cases}$$

where $\lfloor \cdot \rfloor$ is the round down operator, $\epsilon$ the quantization step ($\epsilon = 1$ when rounding to integer), and $P \in [0, 1)$ is a value drawn uniformly at random.

Chapter 3

# Hardware for Embedded DNN Inference Processing

Computing with DNNs relies on two main tasks: training and inference. Today, the complexity of training and the large amount of training data require computing capacity that can only be carried out on servers. GPUs remain the most widely used hardware platform for training DNNs, although there are a few alternatives, such as Google's TPUs. Inference, on the other hand, is less complex, as it no longer requires sophisticated training algorithms to be executed and a large amount of training data to be handled. Whereas inference is mostly performed in the cloud, it is possible to consider deploying DNNs directly on low-power edge devices, thus addressing latency and data privacy issues. However, modern DNNs contain millions of parameters and require billions of arithmetic operations. Memory and computational costs make the deployment on embedded devices difficult, if not infeasible, in many cases. Consequently, specialized architectures are essential to address the computational demands of NNs. These new opportunities have led to the development of hardware that is better suited to embedded applications, such as edge-GPUs and edge-TPUs, or with the emergence of specialized hardware running on FPGAs. Nevertheless, in terms of energy and power consumption, FPGA solutions are known to be more efficient than GPUs. As a result, numerous FPGA-based DNN accelerators have been proposed (see Section 3.2).

This chapter introduces the basic concepts of hardware architectures and accelerators that can be used to implement DNNs, which will facilitate the comprehension of the work carried out in this thesis. In Section 3.1, we briefly present popular choices of hardware used to perform inference on edge devices and justify our decision to use an FPGA. Then, in Section 3.3.2, we describe the two types of neural network accelerators frequently encountered on FPGAs. Finally, in Section 3.4, we detail architectures for parallelizing neural network computation.

## 3.1 Hardware for Embedded DNN Inference

The fundamental operation used in convolutional and dense layers is the Multiply-ACcumulate (MAC) operation, a central operation of a matrix multiplication kernel, which can be easily parallelized. It is possible to use various hardware platforms to perform such operations in the context of DNN inference. In this section, we briefly discuss the most common ones.

### 3.1.1 Central Processing Unit (CPU)

A CPU is the most common type of processor used in computer systems. It is an electronic circuit that sequentially executes a set of instructions. Traditionally, CPUs were single-core. However, today's CPUs are multi-core designs, enabling programs to be parallelized between the different cores. Although CPUs are mainly designed to be versatile, dedicated instructions can be added to speed up DNN processing. For instance, the Intel Knights Mill CPU features special vector instructions for high-performance computing explicitly designed for the acceleration of AI workloads. Micro-Controller Units (MCUs) are more appropriate for embedded applications and are also benefiting from the interest shown by researchers in accelerating calculations with specialized instructions, particularly with the emergence of the RISC-V instruction set architecture [Lou+19b; LHC19; Ass+21].

### 3.1.2 Graphics Processing Unit (GPU)

A GPU is a specialized electronic circuit initially designed to accelerate digital image processing and computer graphics. After their initial design, GPUs were found to be useful for non-graphic calculations involving considerable parallel problems due to their parallel structure. GPUs are now the most widely used hardware accelerator for training ML models. The GPU is a highly parallel processor architecture composed of processing elements and a memory hierarchy. A GPU 's highly parallel structure and high inner bandwidth drive its excellent performance in DNN training processes. However, GPUs are not a good choice for edge applications due to their high power consumption, large cost and footprint requirements, and severe thermal dissipation. An Edge-GPU (also referred to as Embedded GPU) is typically a smaller, more power-efficient version of a traditional GPU, designed for use in devices with limited space, power, and cooling capabilities. In

addition to standard GPUs, edge-GPUs have become another mainstream development platform for embedded systems. The Nvidia Jetson TX2 edge-GPU is small and has low complexity, enough to perform inference in many applications. GPU manufacturers are showing an increasing interest in supporting a wide range of numerical formats, in particular small formats, *e.g.,* binary8, INT8, INT4, and INT1 for Nvidia,[1] to reduce complexity and increase efficiency.

### 3.1.3 Application-Specific Integrated Circuit (ASIC)

ASICs are electronic circuits optimized for a particular set of algorithms or even some specific computational operations rather than intended for general-purpose use as a CPU or a GPU. Well-known ASIC products include Google's TPU, an ASIC optimized for training and inference of large AI models. The co-design of hardware and software makes ASIC chips the most efficient platform with relatively smaller Silicon areas compared to FPGAs. Google's TPUs were initially designed to improve DNN training or perform inference efficiently in the cloud. In early 2019, Google released an edge version of their TPU [2] for embedded inference applications. The Edge-TPU can only accelerate forward-pass operations and is primarily useful for performing inference with the support of 8-bit arithmetic only. However, ASICs also come with very high manufacturing costs, especially in advanced Complementary Metal–Oxide–Semiconductor (CMOS) technologies with the overheads of mask fabrication, which makes them only suitable for applications with a large number of devices produced to amortise the expensive Non-Recurring Engineering costs.

### 3.1.4 Field-Programmable Gate Array (FPGA)

An FPGA is an integrated circuit that can be repeatedly configured to perform complex combinational or arithmetic functions on custom logic for rapid prototyping and deployment. Initially designed to facilitate ASIC prototyping and validate the design, FPGAs are now widely used as part-blown circuits and excel in parallel computing. Most modern FPGAs are System-on-Chips (SoCs) with one or several CPUs integrated on-chip in addition to the FPGA fabric. Both CPUs and FPGAs work with their external memory and can access each other's memory through inner on-chip connections. The CPU cannot

[1] https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[2] https://cloud.google.com/edge-tpu/

afford massive computing operations, so it serves as a switch to launch the whole system and provides some non-linear computing capabilities, such as for computing softmax functions. FPGAs are widely used in embedded systems due to their design flexibility and high performance per Watt, offering a less expensive alternative to ASICs. FPGAs are also starting to be integrated into high-performance computing data centers thanks to their high parallel computing capabilities. They can also incorporate specialized AI cores, such as in Xilinx-AMD's Versal architecture.

## 3.2 Why Choose an FPGA?

To evaluate the efficiency of the DNN hardware, we should consider several parameters such as the cost, the throughput, and the power consumption.

Mass production is the key element in assessing the cost of a DNN hardware. While CPUs, GPUs, and FPGAs are generally the more economical choices, ASICs can be an expensive choice if they are produced in limited quantities. A second important aspect in estimating the cost is the development time. The maturity of development tools such as compilers and frameworks simplifies the deployment of DNNs on CPUs and GPUs. Design tools are beginning to emerge to simplify the deployment of DNNs on FPGAs, such as FINN [Umu+17; Blo+18; Ryb+18] or Vitis AI[3] from AMD-Xilinx. These tools are based on High-Level Synthesis (HLS), a design flow more accessible to software engineers. ASIC development time is the longest if we consider the chip design and fabrication process. Still, once the proper compilation tools have been developed, it is possible to get closer to CPU and GPU development times.

Regarding throughput, ASIC and FPGA have similar performance levels. Embedded GPUs show excellent capacity for computing floating-point data. MCUs contain a small number of processing cores and, as a result, only a small number of processes can be performed in parallel, thus limiting throughput.

The configuration flexibility of FPGAs compared with other hardware platforms is a major advantage. CPUs, GPUs, and ASICs all have fixed designs, so their flexibility is limited to the software that can run on them. As we aim to accelerate unconventional arithmetic formats and various precisions, this flexibility is a key criterion in our use of FPGAs.

[3] https://www.amd.com/en/products/software/vitis-ai.html

Moreover, FPGAs are used in space applications from low Earth orbit to deep space, enabling communications, sensors, instruments, and systems for a new generation of space missions. As the space industry has grown and matured, satellite lifetimes have increased exponentially, much longer than the validity of telecommunication standards, and programmability in-flight has become a stringent requirement. Some satellites are now being repurposed, reconfigured, and reprogrammed for different applications after their initial mission is complete.

Contrary to ASICs or one-time (anti-fuse) programmable FPGAs, the configuration of a reprogrammable FPGA is stored in an Static Random-Access Memory (SRAM), which is sensitive to radiation. With the arrival on the market of major manufacturers such as AMD-Xilinx and Microchip with radiation-tolerant alternatives and the emergence of FPGA manufacturers specialized for space conditions, such as NanoXplore, FPGAs have become real candidates for many space applications such as communication [Man+21], data compression [Jos+24] and image processing [Xu+22].

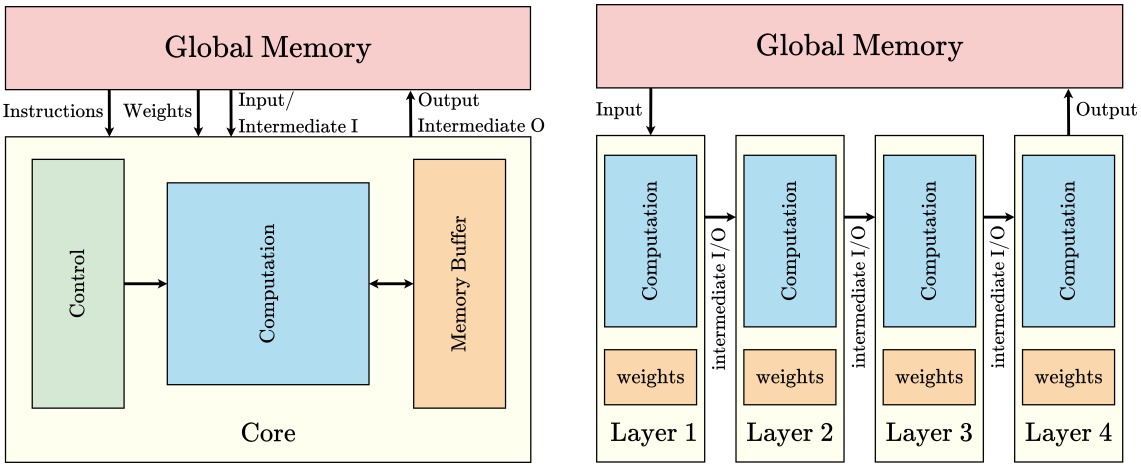
Considering these different criteria, FPGAs seem to be the best candidates for embedded processing on satellite and was also the preferred choice of our partner in this thesis, the CNES.

## 3.3 FPGA-based DNN Accelerators

On FPGA chips, DNN accelerator designs are divided into two distinct categories, as showcased in Figure 3.1. A *sequential accelerator* consists of a single processing engine that processes each layer sequentially. The second category, *dataflow accelerators*, is a streaming architecture consisting of one processing engine per network layer, where inputs are streamed through the dataflow architecture and all layers are computed in parallel.

### 3.3.1 Sequential Accelerators

Sequential accelerators are similar to general-purpose processors with a single processing engine but specialized to handle only the operations present in layers of the DNN models to be executed, as depicted in Figure 3.1a. Only one layer is computed at a time. The processing of the next layer begins when the computation of the previous layer is completed. During layer processing, the full input feature map is available to the accelerator.




(a) Sequential Accelerator (b) Dataflow Accelerator

Figure 3.1: Illustration of sequential and dataflow accelerators. Sequential accelerators include a control unit dedicated to the type of layer used. Only one layer is computed at the time of processing. The next layer's processing begins when the previous layer's computation is complete. During layer processing, the full input feature map is available. This enables access to every element of the input data. The processing time of the whole network depends on the sum of the times for all layers. Dataflow accelerators are organized as a stream of layers executed in parallel. Each layer is directly connected to its predecessor and successor, separated by First In, First Outs (FIFOs). The throughput of the entire network depends on the slowest layer.

This enables access to every element of the input data.

The authors of [And+16; Zha+15; Ovt+15] describe a systolic array style architecture, using theoretical roofline models to design accelerators optimized for the execution of each layer. The authors of [Mos+17] implement a systolic array, but specifically designed for Binary Neural Networks (BNNs). In [CES16], 16-bit fixed-point is used rather than floating-point, and the hardware combines several different data reuse strategies. Stripes [Jud+16] implements a bit-serial processor capable of handling multiple precisions on a single compute array. FP-BNN [Lia+18] implements a single processing engine for BNNs, utilizing an XNOR-popcount datapath. The authors of [Far+09] implement a RISC-like programmable ConvNet processor that computes partial sums of different outputs with an identical input in parallel via multiple Digital Signal Processors (DSPs), and shifts inputs to accomplish convolutions to process an output.

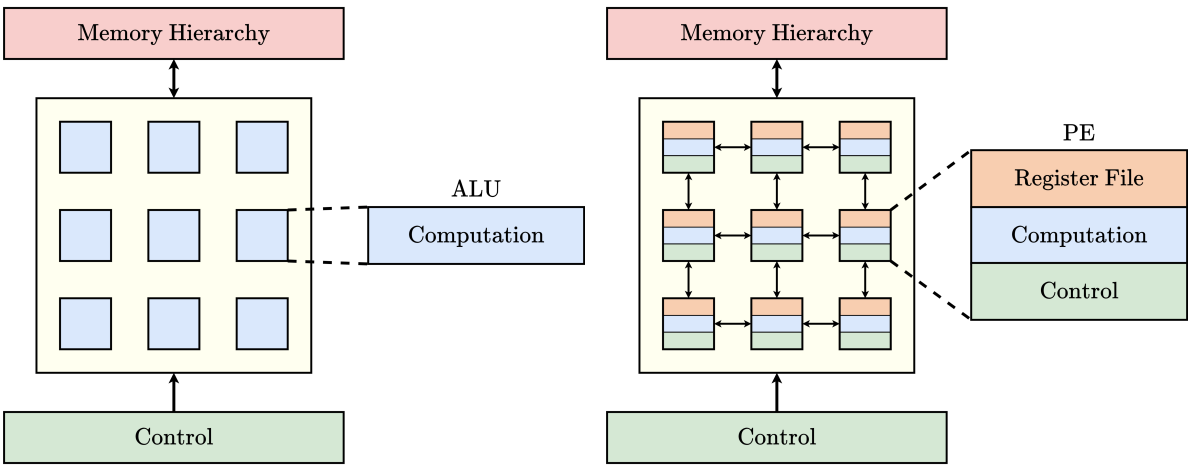
These accelerators are highly versatile, enabling different DNN models to be accelerated without having to change the mapped bitstream or design. However, large models require feature maps and weights to be stored in external memory, which has limited

bandwidth and a higher energy cost than internal memory.

### 3.3.2 Dataflow Accelerators

Feed-forward NNs are, by nature, a streaming-based application in which the execution is purely data-driven. This motivated multiple approaches to investigate the applicability of the dataflow model to accelerate DNNs on FPGAs. Dataflow architectures, also known as *streaming architectures*, parallelize layer computations on the device by pipelining all layers of the DNN, as illustrated in Figure 3.1b. This type of architecture requires fewer external memory accesses, as each layer has its resources on the device and offers higher throughput than sequential architectures. However, deploying large models is limited by the available resources, especially the memory. Furthermore, even if each layer operates in parallel, their complexity, latency, and throughput can differ. This variation in latency requires larger FIFOs memory buffers between layers, which can significantly increase internal memory costs.

The fundamentals of dataflow designs have been formalized to create an architecture where multiple instruction fragments can process data streams simultaneously [DM74]. Programs respecting dataflow semantics are described as Dataflow Process Network. Each node of this network corresponds to a fundamental processing unit called an actor, and each edge corresponds to a communication FIFO channel. Actors exchange abstract data through these FIFOs. Each actor follows a purely data-driven execution model wherein the execution is triggered only by the availability of input operands. Applying the dataflow architecture to accelerate DNN implementations on FPGAs was first investigated by demonstrating the efficiency of the proposed lightweight dataflow methodology [She+10] by mapping convolutional layers with variable clock-domains [Li+16b]. Static Data-Flow models [LM87], a paradigm in which the number of tokens produced and consumed by each actor can be specified *a priori*, is employed in [VB16; VB17] to optimize the mapping of CNN graphs on FPGAs. The CNN graph is modeled as a topology matrix that contains the number of incoming streams, the size of tokens and the consumption rates of each actor. [Abd+17] optimize the direct hardware mapping of CNN graphs. In this approach, each actor is physically mapped on the device with its own specific instance, while each edge is mapped as a signal. [Ale+17; Pro+17] design ternary networks in a dataflow, achieving notably high accuracies and performance. FINN is a framework designed to port BNNs [Umu+17] to AMD-Xilinx FPGAs before being extended to larger bit-width integer formats [Blo+18] and Long-Short Term Memory Neural




(a) Temporal Architecture (b) Spatial Architecture

Figure 3.2: Illustration of temporal and spatial architectures. Temporal architectures use a centralized control for a large number of ALUs. These ALUs can only fetch data from the memory hierarchy and cannot communicate directly with each other. Spatial architectures form a processing chain that passes data from one Processing Element (PE) to another directly. Each PE is composed of ALU, which have their own control logic and local memory, called *register file*.

Networks [Ryb+18].

## 3.4 Efficient Architectures

To achieve high performance, highly parallel computations are commonly used for NN acceleration, including both *temporal* and *spatial* architectures (see Figure 3.2). Temporal architectures appear mainly in CPUs or GPUs, and employ a variety of techniques to improve parallelism. Spatial architectures, on the other hand, mainly concern FPGA and ASIC design using dataflow processing, *i.e.,* the Arithmetic Logic Units (ALUs) form a processing chain so that data can pass directly from one to another.

### 3.4.1 Temporal Architectures

Temporal architectures exploit parallelism by supporting a variety of techniques, such as Single Instruction, Multiple Data (SIMD) or its Single Instruction, Multiple Threads (SIMT) extension to multithreading, to perform MACs in parallel. Temporal architectures use a centralized control for a large number of ALUs, as illustrated in Figure 3.2a. These

ALUs can only fetch data from the memory hierarchy and cannot communicate directly with each other. Software libraries designed for CPUs (*e.g.,* OpenBLAS, Intel MKL) and GPUs (*e.g.,* cuBLAS, cuDNN) are optimized for matrix multiplications. The matrix multiplication is tied to the storage available in the memory hierarchy of these platforms, which are on the order of a few megabytes at the higher levels. Nvidia has combined Tensor cores with traditional CUDA cores in some of its platforms. Tensor cores are a new structure designed to accelerate large matrix operations and perform mixed-precision Matrix Multiply-and-Accumulate (MMAC) calculations in a single operation.

Another technique commonly adopted to reduce the matrix multiplication cost in certain contexts is the Fast Fourier Transform (FFT) [DF12; MHL13], which reduces complexity by operating in the frequency domain at the cost of higher memory usage. Furthermore, several techniques, such as Winograd's algorithm [LG16] and Strassen's algorithm [CX14], are used to reduce the matrix multiplications needed and thereby reduce the resource and memory requirements.

### 3.4.2 Spatial Architectures

Spatial architectures typically consist of an array of PEs for computation. Each PE possesses its own local memory (register file, buffers) and control logic, interconnected with other PEs with various topologies, as illustrated in Figure 3.2b. Spatial architectures provide an opportunity to reduce the energy cost of data movement by introducing several levels of local memory in the hierarchy with different energy costs. This includes the register files in the PEs, which store data for inter-PE movements or accumulations, the global buffers, which store enough values to feed the PEs, and the off-chip memory, usually a Dynamic Random-Access Memory (DRAM).

Memory accesses are the real bottleneck in DNN computations. Therefore, off-chip DRAM accesses must be minimized, as they have a high latency and energy cost. The memory accesses can be reduced by reusing data stored in smaller, faster, and low-energy memories (global buffer and register files). In the context of spatial computing architectures, *weight stationary*, *row stationary* and *output stationary* variants are designed to improve data reuse from memories in the memory hierarchy and to reduce energy consumption.

- Weight Stationary: the weights are kept fixed and are stored in the register files of the PEs, whereas the inputs and partial sums are distributed across the PEs. These

accelerators maximizes filter and convolutional reuse of weights. Weight stationary examples are found in [Cav+15; Par+15; San+09; Sri+10].

- Output Stationary: each partial sum is kept fixed in a PE, and accumulation is performed until the final sum is reached. At the same time, the PEs' weights and inputs are dispersed in a variety of ways. These accelerators are designed to minimize the energy consumption of reading and writing partial sums. Popular output stationary accelerators are [Chi+20; Cho+10].

- Row Stationary: the operations of a row from a convolution are mapped to the same PE in a row-stationary dataflow, while the weights are kept stationary inside the register file of the PEs. These accelerators maximize the convolutional reuse of input feature maps, weights, and partial sums. Row stationary examples are found in [Che+14; Che+19].

- No-Local Reuse: no local storage is allocated to the PE. Instead, all that area is allocated to the global buffer to increase its capacity. This accelerator style maximizes the storage capacity and minimizes the off-chip memory bandwidth. Examples of No-Local Reuse architectures are found in [CES16; Zha+15].

# Summary

In recent years, AI has received much attention in scientific research and industry in various domains, including Image Processing. This is especially the case with DL algorithms. Like many brain-inspired algorithms, the main element of DL algorithms is the artificial neuron modeled by a linear combination of weighted inputs and a bias term. By connecting multiple neurons, we get a neural network organized in a sequence of layers, also known as a DNN. Model parameters are adjusted during training using labeled data to minimize prediction errors via gradient descent. Techniques like mini-batch optimization, proper parameter initialization, learning rate scheduling, and regularization (e.g., $L_1$, $L_2$, Dropout) help improve training efficiency and prevent overfitting. Once training is complete, the trained model predicts new data through forward propagation during inference. Initially relying on statistical methods, computer vision is shifting to DL, particularly CNNs, which have become dominant for tasks like image classification, segmentation, and super-resolution. CNNs use convolutional layers to process local image regions with shared parameters, exploiting spatial relationships while reducing the number of learnable parameters compared to fully connected layers. Padding, striding, and multi-dimensional convolutions enhance CNN efficiency while pooling layers downsample feature maps. For tasks like image segmentation, unpooling via upsampling or transposed convolution restores spatial dimensions, aiding pixel-level classification.

Today, the complexity of training and the large amounts of training data require computing capacity that can only be carried out on servers. Inference, on the other hand, is less complex, making it possible to consider deploying DNNs directly on low-power edge devices, thus addressing latency and data privacy issues. However, modern DNNs contain at least millions of parameters and require billions of arithmetic operations. Memory and computational costs often make deployment on embedded devices difficult, if not infeasible. This problem can be addressed using either a software or a hardware approach. From a software point of view, recent research has focused on neural network compression to mitigate these issues. Quantization is a popular technique that involves mapping a large set of input values to a smaller, finite set, reducing computational demands while maintaining performance. Other compression techniques, such as pruning, knowledge

distillation, and weight sharing, can complement quantization, though each comes with specific challenges and hardware requirements. Representations of real numbers fall into two categories: uniform formats, which consist of evenly spaced integer-based representations, and non-uniform formats, including prevalent floating-point representations and other alternatives like logarithmic and block floating-point formats. While GPUs are widely used for training DNNs due to their ability to parallelize computations, they typically support standard data types, meaning that they often require quantizer blocks to explore the effects of various quantization options. These blocks simulate weight and activation quantization by applying specific functions, allowing for approximations of different formats using single precision arithmetic while leveraging the computational acceleration of GPUs. Quantization typically involves modifying parameter values, which can be accomplished through two main methods: PTQ and QAT. PTQ is a quick and efficient approach that incurs minimal overhead, as it does not require retraining with labeled data. Still, it often results in lower accuracy, particularly with low-precision quantization. Conversely, QAT generally yields more precise outcomes by retraining the model, albeit at the cost of increased training time, the necessity for labeled data, and potentially extensive hyperparameter tuning.

On the hardware side, unlike fixed-designs such as CPUs and GPUs, FPGAs are highly suitable for embedded DNN processing due to their flexibility, power efficiency, and adaptability. They also constitute a more cost-effective alternative to ASICs. FPGA accelerator designs are categorized into sequential accelerators, which process each layer one after the other, and dataflow accelerators, where layers are processed in parallel, offering higher throughput. Efficient architectures can be broadly categorized into temporal and spatial designs. Temporal architectures use centralized control and techniques such as SIMD or SIMT to enhance parallelism. In contrast, spatial architectures consist of an array of PEs, interconnected with each other, with local memory and control logic. Spatial architectures can adopt different data reuse strategies—weight stationary, output stationary, row stationary, or no-local reuse—each optimized to reduce memory access costs and improve data locality.

In the second part of the thesis, we will present our contributions to neural network quantization. Many models can be uniformly quantized below 8 bits. However, using an identical bit-width for the entire network is sub-optimal as different layers typically exhibit different sensitivities to quantization. Thus, Chapter 4 introduces an optimization-based

method for mixed-precision quantization of both weights and activations. Integer quantization is widely used for DNN model compression and inference acceleration. Other arithmetic formats, such as small precision floating-point formats (minifloats), have not been extensively studied in the DNN quantization literature. Chapter 5 therefore investigates minifloat quantization and, more precisely, a minifloat encoding scheme optimized for inference based on an asymmetric exponent bias. One key feature of U-shaped models, a popular semantic segmentation architecture that serves as a use-case for our work, has received little attention considering its memory footprint: the skip connections. Therefore, Chapter 6 studies the impact of long skip connections in U-shaped models used for semantic segmentation and introduces a more suitable model for dataflow architecture.

Part II

# Contributions

Chapter 4

# A Gradient-Based Method for Learning the Bit-Width During Training

Quantization is a very effective method for reducing the computational complexity and memory footprint of NNs. Many models can be uniformly quantized below 8 bits [Zho+16] and, in some cases, even with binary or ternary representations. However, using an identical bit-width for the entire network is sub-optimal as different layers typically exhibit different sensitivities to quantization [ZBS22]. This forces layers that are less sensitive to quantization to use the same bit-width as quantization-sensitive layers, missing the opportunity to reduce the average bit-width of the whole network. In the same way, as the scaling factor granularity can be adapted (as explained in Section 2.3.2), finer-grained bit-widths can be used at different levels of the model, *i.e.,* layers or kernels. In the literature, these quantization methods are referred to as *mixed-precision quantization*. Thus, refined bit-width granularity provides a more flexible accuracy-efficiency trade-off adjustment than uniformly quantized models. Mixed precision is particularly interesting for dataflow accelerators, with more effective usage of available resources. However, allocating different bit-widths for each layer is challenging. The search space for per-layer parameters' bit-widths is an exponential function of the number of layers. It grows as the granularity becomes finer, which renders manual assignments of these bit-widths unfeasible. This calls for automatic, specialized methods that can efficiently determine mixed-precision allocations.

This chapter introduces an efficient optimization-based method for mixed-precision uniform quantization of both weights and activations. In Section 4.1, we review state-of-the-art methods used to address this challenging problem. Then, in Section 4.2, we present our gradient-based approach that uses relaxed fractional bit-widths updated us-

ing a gradient descent rule. Finally, in Section 4.3, we compare this approach with other gradient-based methods and discuss some key conceptual choices. Part of the work presented in this chapter has been published in [Ger+24]. The AdaQAT project is available on Gitlab.

## 4.1 Bit-Width Search Strategies

Finding bit-width allocations that improve inference efficiency has been addressed using various approaches that can be divided into four main families: *Gradient*, *Reinforcement Learning (RL)*, *Neural Architecture Search (NAS)*, and *Heuristic*.

### 4.1.1 Gradient-based Optimization Methods

Several problems arise when the precision is considered as a learning parameter to be optimized via gradient descent. The number of bits assigned to weight or activation tends to be large as higher precision naturally leads to lower losses. Furthermore, discrete non-differentiable quantization cannot be integrated with vanilla backpropagation techniques.

The authors of [LTA18] formulate the mixed-precision problem as constrained by stochastically allocated precision via modified gradient descent. They introduce a new layer, called the precision allocation layer, into the forward pass. This layer, used during the training phase, is responsible for constructing a Gumbel-Softmax distribution [JGP16] [1] to handle the discrete stochastic operations used to sample the bit-widths. A precision budget is the sum of allocated bits of each layer in the model. The process is repeated until the budget is reached.

BSQ [Yan+21] proposes to decrease the precision of weights from a sparsity perspective, *i.e.,* the precision can be taken as forcing one or a few bits, most likely the least significant bit, to be zero. Reducing the precision of a layer is equivalent to zeroing out a specific bit of all the weight parameters of the layer. By considering the bit-widths of DNN parameters as continuous trainable variables during training, they can utilize a sparsity-inducing regularizer to explore the bit-level sparsity with gradient-based optimization, dynamically reducing the layer precision and leading to a series of mixed-precision quantization schemes.

[1] Gumbel-Softmax is a continuous distribution that has the property that it can be smoothly annealed into a categorical distribution.

DQ [Uhl+19] proposes to use differentiable quantization to learn all quantization parameters of weights and activations using per-layer quantization with a global memory constraint via gradient descent. The quantization parameter is defined by the step size, the maximum value and the number of bits used to encode the quantized value. To learn the optimal quantization parameters, they define the gradient using STE to differentiate non-differentiable floor operations. In order to use simple stochastic gradient descent solvers, they add a regularizing penalty term to the loss function driven by the memory impact of the model.

FracBits [YJ21] proposes treating bit-widths as continuous values by interpolating quantized weights or activation values using the two neighboring bit-widths. They apply a first-order expansion around one of its nearby integers and approximate the derivative at this integer by the slope of the segment joining the two adjacent grid points, making it learnable through SGD. BitPruning [Nik+20] optimizes bit-widths by defining a quantization scheme that starts with integer bit-widths and extends to fractional bit-widths for gradient-based training. It uses interpolation between integer bit-widths to enable differentiable learning, allowing bit-widths to be adjusted during training. A parameterizable loss function penalizes larger bit-widths to balance accuracy and bit-width, with adjustments made based on the importance of different network layers. The final bit-widths are rounded to the nearest integer for practical hardware implementation. BitPruning includes training with interpolated bit-widths, using a straight-through estimator for gradient propagation, and fine-tuning to recover accuracy post-quantization.

SDQ [Hua+22] introduces Differentiable Bit-width Parameters (DBPs) to manage bit-width assignment for each layer. During training, weights can be quantized to varying bit-widths based on DBPs, with gradients computed using a Straight-Through Gumbel SoftMax Estimator. The quantization strategy includes a regularizer to penalize high quantization errors, particularly in layers with many parameters. After the quantization strategy is fixed, training focuses on knowledge distillation and entropy-aware bin regularization to minimize the performance gap and maintain weight distribution integrity.

DDQ [Zha+21a] learns the different quantization parameters of each layer, including bit-width, quantization level, and dynamic range. They formulate quantization as a low-precision matrix-vector product, where the quantization levels are learned through SGD. By leveraging Kronecker products, they reduce the number of parameters, making it scalable and adaptable for different layers within neural networks, allowing for flexible quantization levels and bit-widths while maintaining efficient hardware performance.

DMBQ [ZYH21] incrementally quantizes a neural network to an expected average bit-width during training by adjusting it. The loss sensitivity of each layer is accumulated and used to adjust bit-widths using a loss-guided bit-width allocation strategy. The process continues until the target average bit-width is reached, optimizing for minimal quantization error under a Laplace distribution model.

Bayesian Bits [Van+20] recursively quantizes the residual error to progressively higher bit-widths by adding quantized residuals. This approach enables a stepwise increase in precision while maintaining a hardware-compatible format. Learnable gating variables are introduced to control the effective bit-width. Each gate either activates or deactivates the addition of residuals for higher bit-widths, allowing the model to optimize bit-widths based on the data. This can also be extended to prune values by treating a zero-bit-width as quantization to zero. The method leverages a shared gating mechanism for structured pruning across filters in convolutional layers. They explore learning the gating variables using Bayesian methods, specifically variational inference. A prior favoring lower bit-widths is introduced, and the gates are optimized through an efficient gradient-based approach. The objective function is designed to regularize the model towards lower-bit-widths while ensuring good predictive performance. An approximation using the reparametrization trick is employed to reduce the variance of gradients during training, and a thresholding method is used for gate pruning at test time.

DNAS [Wu+18a] transforms NAS into a gradient-based optimization problem, leveraging a stochastic super net and Gumbel-Softmax to differentiate precision assignments. The architecture search space is represented as a stochastic super net, where nodes are intermediate data tensors, and edges are operators like convolution layers. Any candidate architecture is a sub-graph of the super net, and the edges are executed stochastically based on architecture parameters. The goal is to find the optimal architecture parameters that yields the best expected performance. To achieve this, the super net is trained using SGD for both its weights and architecture parameters. The Gumbel SoftMax function handles the discrete nature of the stochastic edge execution, making the super net fully differentiable and trainable with SGD. EBS [Li+20] considers jointly reducing the memory and computational cost. They maintain a single meta weights tensor that can adapt to branches of different quantization precision. EBS uses a softmax function over learnable strength parameters to determine the optimal bit-width, while stochastic search techniques such as Gumbel-Softmax are applied to improve precision. Additionally, the Binary Decomposition technique is introduced to efficiently handle mixed-precision com-

putations during inference by using binary operations.

HMQ [HJN20] learns a quantization scheme parameterized by a threshold and bit-width, with the threshold restricted to powers of two for hardware efficiency. They use the Gumbel-Softmax estimator to search for optimal pairs of threshold and bit-width over a discrete set to enable gradient descent. During training, it samples and updates quantization parameters smoothly through backpropagation. The optimization process involves two phases: the first phase trains both model weights and HMQ parameters using a gradually increasing compression target, while the second phase fine-tunes only the model weights with fixed quantization parameters.

### 4.1.2 Reinforcement Learning (RL)-based Methods

Reinforcement Learning (RL) allows an agent to learn in an interactive environment by trial and error using feedback from its own actions and experiences. Unlike previously introduced supervised learning methods, where the feedback provided to the agent is the correct set of actions for performing a task, RL uses rewards and punishments as signals for positive and negative behavior.

The very first works to use RL algorithms for mixed-precision quantization are HAQ [Wan+19b] and Releq [Elt+19]. HAQ leverages RL to automate the quantization policy, which dictates the bit-widths of weights and activations per-layer while considering the feedback of the latency and energy of FPGA hardware accelerators. They use two representative hardware architectures that support mixed-precision: BitFusion [Sha+18] (spatial) and BISMO [URS18] (temporal). The RL agent is the DDPG [Lil+15], an actor-critic algorithm. Later, an extension is proposed to target the tight computational and memory constraints of tiny MCU devices [Rus+20].

Releq [Elt+19] exploits the sample efficiency of PPO [Sch+17], an actor-critic technique whereby the agent comprises both policy and value networks, to automate the exploration of weight bit-widths of each layer. They formulate the problem as an end-to-end Long-Short-Term Memory-based RL approach that optimizes a policy for per-layer bit-width allocation by exploring the hyper-parameter quantization space. To consider the effects of previous layers' quantization levels, the agent steps sequentially through the layers and chooses a bit-width from a predefined set, one layer at a time. The agent consequently receives a reward signal proportional to its accuracy after quantization and its computation and memory cost benefits.

AutoQ [Lou+19a] relies on hierarchical deep RL to automatically assign mixed-precision

for weights kernels and activations of each layer. In order to choose the bit-widths of weights and activations, AutoQ relies on two controllers: a High-Level Controller (HLC) and a Low-Level Controller (LLC). The task of the HLC is to choose a bit-width for activation or the average bit-width of all weight convolution kernels of each layer while the LLC produces a bit-width for each weight kernel in a single layer. Both HLC and LLC are implemented using a HIRO [Nac+18] agent, an off-policy correction, and learn at the same time by trial and error.

ADLR [Nin+20] uses a modified actor network to generate multiple candidate actions from a given state and employs a refinement function to select the most promising action. This method aims to reduce variance and accelerate convergence during training by effectively handling discrete bit-widths and leveraging domain-specific knowledge. Training with DDPG, ADRL refines its approach by evaluating actions through expanded actor functions and a deterministic refinement process. They optimize network configurations through a combination of profiling and distance-based indicators.

DQMQ [Wan+24] searches for an optimized mixed-precision model from the perspective of data. DQMQ is modeled as a hybrid RL task to dynamically select optimal bit-widths based on input data and layer sensitivity. The system includes a Precision Decision Agent, which automatically determines the optimal layer-wise bit-widths during forward propagation, and a Quantization Auxiliary Computer, which mitigates the accumulation of quantization bias, enabling one-shot mixed-precision training.

### 4.1.3 Neural Architecture Search (NAS) Methods

Neural Architecture Search (NAS) is the process of automating all steps in the ML pipeline, from data cleaning to feature engineering and selection to hyperparameter and architecture search. This method can also find mixed-precision bit-width configurations by integrating precision into these objectives.

JASQ [Che+18b] is a genetic algorithm-based approach that simultaneously searches for optimized neural network architectures and precision settings by using the validation accuracy of quantized models as the fitness metric. The process begins with an initial population of models, which evolves iteratively based on fitness. In each evolutionary step, models are randomly selected from the population according to their fitness scores, with the top-performing model chosen as the parent, while the lowest-performing one is removed. The parent is mutated by altering its architecture and bit-width, and the resulting child is trained, quantized, evaluated for fitness, and reintroduced into the population.

In [Gon+19a], a controller generates various neural network configurations, each evaluated on the accuracy and energy cost, estimated from a physical simulator based on BitFusion [Sha+18], using Monte Carlo sampling and optimization techniques. The dimensions of the architecture search space on MobileNetV2 backbone are kernel size and expansion ratio. The depthwise and bottleneck layers within a single MobileNet convolutional block are assigned different precisions. The algorithm employs Gumbel-Softmax for the differentiable search method to find an optimal configuration of neural architecture and bit-width. The optimization process iteratively samples configurations, updates network weights, and refines the controller until convergence.

APQ [Wan+20] uses an evolutionary architecture search that optimizes the variables directly based on the measured latency and energy of the target hardware.

EdMIPS [CV20] and BP-NAS [Yu+20] adopt a differentiable search architecture based on DARTS [LSY18] to enable search without proxies. EdMIPS simplifies DARTS bi-level optimization by updating both architecture and quantization parameters in a single forward-backward pass. To prevent the trivial selection of the highest bit-width, a complexity-constrained optimization is introduced and reformulated as a Lagrangian for balancing accuracy and complexity. BP-NAS proposes incorporating bit operations constraint into the loss function and binding the search process within the constraint. They also propose a Prob-1 regularizer to aid in node selection during the search and suggest using a uniform-like pre-trained model for more stable training of mixed-precision models.

SSPS [Sun+21] involves a differentiable single-path search cell that constructs a super net and integrates constraints into the loss function to guide the search process. It uses Gumbel-Softmax for differentiable sampling and optimizes quantization bit-width, considering both model size and computational complexity. The search process iteratively refines the quantization bit-width for each layer, leveraging entropy to evaluate and guide the bit-width selection, ultimately producing a quantization model that meets predefined requirements.

### 4.1.4 Heuristic-based Optimization

Heuristic methods include all approaches that use the dynamics of the data to compute the sensitivity of layers and assign precision accordingly.

HAWQ [Don+19] proposes to use second-order information from the Hessian matrix to address the problem of mixed-precision quantization. HAWQ quantizes blocks of layers rather than individual layers, reducing the overhead of applying different bit-widths.

Specifically, they compute the Hessian spectrum (eigenvalues) to assess each layer's sensitivity to quantization. Layers with smaller eigenvalues are deemed less sensitive and can be quantized to lower precision, while more sensitive layers receive higher precision. In HAWQ-V2 [Don+20], the authors propose to calculate the Hessian trace instead of computing the full Hessian matrix, which captures layer sensitivity efficiently. This reduces the complexity of obtaining second-order information. In HAWQ-V3 [Yao+21], all inference computations use integer-only multiplication, addition, and bit shifting with static quantization to better suit deployment on integer-only hardware. This includes the batch norm layers and residual connections, typically kept at floating point precision. In further mode, the authors formulate the mixed-precision problem as an Integer Linear Programming with latency, model size, and/or Bit Operations taken as constraints supporting only 4- or 8-bit bit-widths.

The authors of [CWC21] formulate the mixed-precision quantization problem as a discrete constrained optimization problem. They propose to approximate the original objective function with Taylor expansion and propose an efficient approach to compute the Hessian matrix of each layer. They reformulate the problem as a particular variant of the Knapsack problem called Multiple Choice Knapsack Problem and propose a greedy search algorithm to solve it efficiently.

OMPQ [Ma+21] defines the orthogonality of a NN layer by extending the concept from function orthogonality to the entire network. The Orthogonality Metric (ORM) quantifies the importance of each layer. This metric is computed efficiently using Monte Carlo sampling to guide the optimal bit-width configuration for mixed-precision quantization. The authors construct a linear programming problem that integrates the orthogonality values. ORM-based configurations assign larger bit-widths to layers with stronger orthogonality, maximizing the model's representation capability while reducing computational resources.

ZeroQ [Cai+20] uses a Pareto frontier optimization method to automatically select the bit-precision configuration by computing the quantization sensitivity based on the Distilled Data with a small computational overhead. The authors of [Pan+23] use the signal-to-quantization noise ratio to measure the relative sensitivity of each quantizer when quantized to different bit-width configurations. They allocate different bit-widths to each layer in the network based on a desired Bit Operations budget and layer sensitivity. FILM-QNN [Sun+22a] chose to use only 4- and 8-bit precision in their exploration space, justifying this decision to maximize the resource usage on an FPGA. They calculate

the quantization error based on the Euclidean distance to determine the layers likely to impact the accuracy and quantize them to 8 bits. A percentage of layers with high precision is given to determine the number of layers quantized to 8 bits. Only the weights are quantized with different bit-widths, while the activation bit-widths are specified before training.

With the exact requirements, MSP [Cha+20] proposes a mixed-precision algorithm targeting FPGAs. In each layer, they compute the average quantization error between 8-bit weight and its nearest 4-bit weight to determine the 5% of weights left to 8 bits and those reduced to 4 bits. The authors of [Vas+21] use an activation density metric to determine the optimal bit-precision per-layer. This activation density is the proportion of non-zero activations in a layer calculated by passing the training set through the network. During training, they monitor the activation density for all the layers. Once the activation density stabilizes across all layers, they break the training process and quantize weight and activation to the same precision. The quantization bit-width for each layer is obtained by multiplying the initial bit-width by the activation density. BMPQ [Kun+22] extend the approach to QAT. The authors of [Vas+21] propose to use Activation Density as a distinguishing metric that enables the computation of the optimal mixed-precision configuration during training. They iteratively modify bit-widths in every epoch according to the current bit-width and Activation Density and repeat the process until the Activation Density is stable.

The authors of [SR21] propose to allocate the bit-widths using an MLP model trained to predict the bit-width configuration. Kullback-Leibler (KL) divergence between the output of the full precision and the quantized model pairs with bit-width configurations is used to create a custom dataset to train the MLP model. The model intended to be compressed is quantized with multiple mixed-precision bit-width configurations determined using Monte Carlo sampling of the search space. For a desired KL divergence the MLP model predicts the bit-width configuration the network should be quantized with.

EvoQ [Yua+20] uses an evolutionary algorithm to achieve mixed-precision quantization with limited data. They evaluate the quantization policy by measuring the output difference between the quantization model and the pre-trained full-precision model using a set of samples. They analyze the per-layer quantization sensitivity and utilize the analysis results to optimize the mutation operation to improve the search efficiency and help escape from local minimums. Finally, they use a set of samples to calibrate the outputs and intermediate features of the quantization model with the pre-trained full-precision

model.

EMQ [Liu+21] formulates the mixed-precision quantization problem as a genetic problem, where a variable is attached to the quantization interval of weights or activations in one layer. They treat each encoding regarding a quantization network as an individual and stack them to constitute the population. To balance the accuracy and the latency of the quantized model, they consider both model performance and computation costs.

GMPQ [Wan+21] focuses on preserving attribution rank consistency between full-precision and quantized models, which improves the generalizability of the quantization strategy across different datasets. Instead of requiring consistent datasets for training and validation, GMPQ aims to find a quantization policy that performs well across various data distributions. The key is to preserve attribution rank consistency, which helps adaptively adjust the distribution of model attributions despite the reduced capacity of quantized models. This ensures that the quantized network retains the discriminative power of its full-precision counterpart, making the quantization process more efficient and generalizable.

MPQ [Kim+20] dynamically assigns extra bits to weights based on frequent bit-level changes, especially near quantization thresholds. The learning rate for weights near quantization thresholds is adjusted to control gradient noise. The authors of [Tan+22] transform MPQ into a one-time integer linear programming problem by leveraging the learned importance of each layer, increasing time efficiency without limiting the bit-width search space. HW-FlowQ [Fas+21] proposes to use a genetic algorithm to optimize layer-wise quantization strategies, balancing hardware efficiency and task accuracy through iterative refinement. The design process involves three abstraction levels: Coarse, Mid, and Fine, each progressively refining design parameters to meet application constraints. At each level, the genetic algorithm evaluates model configurations, incorporating hardware constraints like memory hierarchy and data transfer volumes. The authors of [Sun+22b] introduce a Quantization Entropy Score (QE-Score) to evaluate the expressiveness of mixed-precision quantization models. Based on the Maximum Entropy Principle, the entropy of a network's last feature map reflects its expressiveness, with Gaussian distributions maximizing entropy. They describe how full-precision model entropy is related to variance and extend this to mixed-precision quantization by considering how quantization affects variance, introducing a scaling parameter to normalize the inputs.

The major downside of using NAS and RL approaches is that they require significant time and computational resources. A potential downside of heuristic-based methods is that they can often lead to sub-optimal results. Gradient-based methods strike a good balance between time complexity and the quality of solutions obtained. This has been the primary motivating factor for investigating such methods.

In the rest of this chapter we introduce AdaQAT (from Adaptive QAT), a gradient-based mixed-precision QAT optimization method that shows good flexibility when compared to other approaches in the same vein.

## 4.2 The AdaQAT Algorithm

For simplicity, we start by presenting AdaQAT in the context of learning only one bit-width for all weights in the network and similarly for activations. We then extend the method to per-layer bit-width allocation.

### 4.2.1 Objective Function

To learn the bit-widths of the uniform quantizers for both weights and activations, we use two real-valued variables $N_{\mathbf{w}}$ and $N_{\mathbf{a}}$, respectively. The actual integer bit-widths of the quantized network will be $\lceil N_{\mathbf{w}} \rceil$ and $\lceil N_{\mathbf{a}} \rceil$.

We use a regularization approach (see Eq. (1.9)) to model the loss function that takes into account the cost of a particular bit-width configuration, which corresponds to:

$$\mathcal{L}_{\text{Total}} = \mathcal{L}_{\text{Task}} \left( \mathbf{Y}_{\text{U,U}}^{(L)}; \mathbf{T} \right) + \lambda \mathcal{L}_{\text{Hard}} \left( \lceil N_{\mathbf{w}} \rceil, \lceil N_{\mathbf{a}} \rceil \right), \tag{4.1}$$

where $\lambda > 0$ is a regularization hyper-parameter between the main task loss $\mathcal{L}_{\text{Task}}$ (*i.e.,* the usual loss for the task that needs to be addressed, like cross-entropy for image classification) and the regularizing hardware loss $\mathcal{L}_{\text{Hard}}$ that measures the complexity of the model (in this case in terms of hardware cost). Here, $\mathbf{Y}_{\text{U,U}}^{(L)}$ represents the output of the $L$-layer quantized network we want to optimize, where its weights have been quantized to $\lceil N_{\mathbf{w}} \rceil$ bits (the first U in the superscript notation) and its activations to $\lceil N_{\mathbf{a}} \rceil$ bits (the second U in the superscript) on a current minibatch of data and $\mathbf{T}$ is the expected output.

FracBits [YJ21] has reviewed various methods used to model the hardware cost of arithmetic precision choices for weights and activations. They argue in favor of memory size if only targeting weight quantization, and BitOPs (see [YJ21, Eqs. (4) and (5)]) for

joint weight and activation quantization. As an example, for a convolutional filter $f$, the BitOPs metric corresponds to

$$\text{BitOPs}(f) = \lceil N_{\mathbf{w}} \rceil \, \lceil N_{\mathbf{a}} \rceil \, |f| k_x k_y / s_f^2,$$

where $|f|$ denotes the filter's cardinality, and $k_x$, $k_y$, and $s_f$ are the filter's spatial width, height, and stride, respectively. And for a linear layer to

$$\text{BitOPs}(l) = \lceil N_{\mathbf{w}} \rceil \, \lceil N_{\mathbf{a}} \rceil \, |w_l|,$$

where $l$ is a linear layer and $|w_l|$ denotes the number of weights in the linear layer $l$. Thus, the overall BitOPs of a model corresponds to the sum each convolution or linear layer's BitOPs.

In our particular case, since we are using one bit-width per weights and one per activations, the overall BitOPs hardware cost will be linear in $\lceil N_{\mathbf{w}} \rceil \lceil N_{\mathbf{a}} \rceil$, and we can simplify the regularization term to

$$\mathcal{L}_{\text{Hard}} \left( \lceil N_{\mathbf{w}} \rceil \, , \lceil N_{\mathbf{a}} \rceil \right) = \lceil N_{\mathbf{w}} \rceil \, \lceil N_{\mathbf{a}} \rceil \, . \tag{4.2}$$

Using Eq. (4.2) will impact the value we need to consider for $\lambda$, which will be smaller than when integrating the scaling factors specific to each layer (determined by the layer type and shape) directly into $\mathcal{L}_{\text{Hard}}$.

### 4.2.2 Bit-Width Gradients and Parameter Updates

Since the task loss $\mathcal{L}_{\text{Task}}$ is not directly differentiable with respect to the bit-width parameters, we use finite difference approximations as

$$\begin{aligned} \frac{\partial \mathcal{L}_{\text{Task}}}{\partial N_{\mathbf{w}}} &\approx \mathcal{L}_{\text{Task}}(\mathbf{Y}^{(L)}_{\text{U,U}}; \mathbf{T}) - \mathcal{L}_{\text{Task}}(\mathbf{Y}^{(L)}_{\text{D,U}}; \mathbf{T}), \\ \frac{\partial \mathcal{L}_{\text{Task}}}{\partial N_{\mathbf{a}}} &\approx \mathcal{L}_{\text{Task}}(\mathbf{Y}^{(L)}_{\text{U,U}}; \mathbf{T}) - \mathcal{L}_{\text{Task}}(\mathbf{Y}^{(L)}_{\text{U,D}}; \mathbf{T}). \end{aligned} \tag{4.3}$$

The U/D superscripts in the $\mathbf{Y}^{(L)}$ network outputs denote how the weights (first superscript) and activations (second superscript) have been quantized: U represents $\lceil N_{\mathbf{w}} \rceil$ or $\lceil N_{\mathbf{a}} \rceil$ and D represents $\lfloor N_{\mathbf{w}} \rfloor$ or $\lfloor N_{\mathbf{a}} \rfloor$.

The gradients of the total loss (4.1) w.r.t. the bit-widths are then approximated as

$$\frac{\partial \mathcal{L}_{\text{Total}}}{\partial N_{\mathbf{x}}} \approx \frac{\partial \mathcal{L}_{\text{Task}}}{\partial N_{\mathbf{x}}} + \lambda \frac{\partial \mathcal{L}_{\text{Hard}}}{\partial \lceil N_{\mathbf{x}} \rceil}, \tag{4.4}$$

which are then used to update the fractional bit-width parameters. The gradient descent rule that does this takes the form

$$N_{\mathbf{x}}^{+} = N_{\mathbf{x}} - \eta_{\mathbf{x}} \frac{\partial \mathcal{L}_{\text{Total}}}{\partial N_{\mathbf{x}}}, \tag{4.5}$$

with $\mathbf{x} \in \{\mathbf{w}, \mathbf{a}\}$, $N_{\mathbf{x}}^{+}$ the new bit-width values at the next iteration, and $\eta_{\mathbf{x}} > 0$ the corresponding learning rates.

Algorithm 1 summarizes how AdaQAT works. During the *forward propagation phase* (forward pass computations through each layer are done using the `forward` function), the weights and activations are quantized according to the learned integer bit-widths, $\lceil N_w \rceil$ and $\lceil N_a \rceil$, respectively, and the quantization policy (through the `quantize_w` and `quantize_a` functions). We adopt the DoReFa [Zho+16] scheme for weight quantization and PACT [Cho+18] for activation quantization with the improvements suggested in SAT [JYL19]. In the *gradient estimation phase*, the bit-width gradients are approximated according to Eq. (4.3) and Eq. (4.4), while in the *parameter update phase*, we use Eq. (4.5) to update the bit-width parameters.

The rest of the network (*e.g.,* weights and bias terms) and quantizer parameters are updated in parallel using the SGD-like or accelerated algorithms that train the network normally, with their separate hyperparameters (see Section 1.3.1).

### 4.2.3 Convergence Behavior

When $N_{\mathbf{w}}$ and $N_{\mathbf{a}}$ reach their optimized values, continuing to decrease them will lead to a (steep) increase of the task loss $\mathcal{L}_{\text{Task}}$ and consequently of $\mathcal{L}_{\text{Total}}$. This means that their gradient estimates in Eq. (4.4) will become negative, and the gradient descent rule in Eq. (4.5) will start increasing $N_{\mathbf{w}}$ and $N_{\mathbf{a}}$. Then, an oscillatory pattern forms (for an example, see Figure 4.1). This oscillation phenomenon is not new, being present in several STE-based gradient approximation methods [Nag+22; Kuz+22; LLC23]. When this happens, we monitor the number of oscillations, and as soon as it passes a certain threshold (which we empirically set to 10), we fix the bit-widths to $\lceil N_{\mathbf{w}} \rceil$ and $\lceil N_{\mathbf{a}} \rceil$, respectively, and continue the rest of the quantization process in standard QAT fashion.

**Algorithm 1** AdaQAT iteration for updating the $N_{\mathbf{a}}, N_{\mathbf{w}}$ bit-widths of an $L$-layer model

**Require:** a minibatch of inputs $\mathbf{Y}^{(0)}$ and corresponding targets $\mathbf{T}$, weights $\mathbf{W} \in \mathbb{R}$, weight bit-width $N_{\mathbf{w}} \in \mathbb{R}_+^*$, activation bit-width $N_{\mathbf{a}} \in \mathbb{R}_+^*$, regularization parameter $\lambda > 0$, bit-width learning rates $\eta_{\mathbf{w}}, \eta_{\mathbf{a}} > 0$, loss functions $\mathcal{L}_{\text{Task}}$ and $\mathcal{L}_{\text{Hard}}$.

**Ensure:** updated parameters $N_{\mathbf{w}}^+$ and $N_{\mathbf{a}}^+$

/* **1. Forward propagation:** */

$\widetilde{\mathbf{W}}_{\text{U}}^{(1)} \leftarrow \texttt{quantize\_w}\left(\mathbf{W}^{(1)}, 8\right)$

$\mathbf{Y}_{\text{U,U}}^{(1)}, \mathbf{Y}_{\text{D,U}}^{(1)}, \mathbf{Y}_{\text{U,D}}^{(1)} \leftarrow \texttt{forward}\left(\mathbf{Y}^{(0)}, \widetilde{\mathbf{W}}_{\text{U}}^{(1)}\right)$

**for** $k = 2$ **to** $L - 1$ **do**

  $\widetilde{\mathbf{W}}_{\text{U}}^{(k)} \leftarrow \texttt{quantize\_w}\left(\mathbf{W}^{(k)}, \lceil N_{\mathbf{w}} \rceil\right), \quad \widetilde{\mathbf{W}}_{\text{D}}^{(k)} \leftarrow \texttt{quantize\_w}\left(\mathbf{W}^{(k)}, \lfloor N_{\mathbf{w}} \rfloor\right)$

  $\mathbf{Y}_{\text{U,U}}^{(k)} \leftarrow \texttt{forward}\left(\texttt{quantize\_a}\left(\mathbf{Y}_{\text{U,U}}^{(k-1)}, \lceil N_{\mathbf{a}} \rceil\right), \widetilde{\mathbf{W}}_{\text{U}}^{(k)}\right)$

  $\mathbf{Y}_{\text{D,U}}^{(k)} \leftarrow \texttt{forward}\left(\texttt{quantize\_a}\left(\mathbf{Y}_{\text{D,U}}^{(k-1)}, \lceil N_{\mathbf{a}} \rceil\right), \widetilde{\mathbf{W}}_{\text{D}}^{(k)}\right)$

  $\mathbf{Y}_{\text{U,D}}^{(k)} \leftarrow \texttt{forward}\left(\texttt{quantize\_a}\left(\mathbf{Y}_{\text{U,D}}^{(k-1)}, \lfloor N_{\mathbf{a}} \rfloor\right), \widetilde{\mathbf{W}}_{\text{U}}^{(k)}\right)$

**end for**

$\widetilde{\mathbf{W}}_{\text{U}}^{(L)} \leftarrow \texttt{quantize\_w}\left(\mathbf{W}^{(L)}, 8\right)$

$\mathbf{Y}_{\text{U,U}}^{(L)} \leftarrow \texttt{forward}\left(\texttt{quantize\_a}\left(\mathbf{Y}_{\text{U,U}}^{(L-1)}, \lceil N_{\mathbf{a}} \rceil\right), \widetilde{\mathbf{W}}_{\text{U}}^{(L)}\right)$

$\mathbf{Y}_{\text{D,U}}^{(L)} \leftarrow \texttt{forward}\left(\texttt{quantize\_a}\left(\mathbf{Y}_{\text{D,U}}^{(L-1)}, \lceil N_{\mathbf{a}} \rceil\right), \widetilde{\mathbf{W}}_{\text{D}}^{(L)}\right)$

$\mathbf{Y}_{\text{U,D}}^{(L)} \leftarrow \texttt{forward}\left(\texttt{quantize\_a}\left(\mathbf{Y}_{\text{U,D}}^{(L-1)}, \lfloor N_{\mathbf{a}} \rfloor\right), \widetilde{\mathbf{W}}_{\text{U}}^{(L)}\right)$

/* **2. Gradient Estimation:** */

$$\frac{\partial \mathcal{L}_{\text{Task}}}{\partial N_{\mathbf{w}}} \approx \mathcal{L}_{\text{Task}}\left(\mathbf{Y}_{\text{U,U}}^{(L)}; \mathbf{T}\right) - \mathcal{L}_{\text{Task}}\left(\mathbf{Y}_{\text{D,U}}^{(L)}; \mathbf{T}\right)$$

$$\frac{\partial \mathcal{L}_{\text{Task}}}{\partial N_{\mathbf{a}}} \approx \mathcal{L}_{\text{Task}}\left(\mathbf{Y}_{\text{U,U}}^{(L)}; \mathbf{T}\right) - \mathcal{L}_{\text{Task}}\left(\mathbf{Y}_{\text{U,D}}^{(L)}; \mathbf{T}\right)$$

$$\frac{\partial \mathcal{L}_{\text{Total}}}{\partial N_{\mathbf{w}}} \approx \frac{\partial \mathcal{L}_{\text{Task}}}{\partial N_{\mathbf{w}}} + \lambda \frac{\partial \mathcal{L}_{\text{Hard}}}{\partial \lceil N_{\mathbf{w}} \rceil}$$

$$\frac{\partial \mathcal{L}_{\text{Total}}}{\partial N_{\mathbf{a}}} \approx \frac{\partial \mathcal{L}_{\text{Task}}}{\partial N_{\mathbf{a}}} + \lambda \frac{\partial \mathcal{L}_{\text{Hard}}}{\partial \lceil N_{\mathbf{a}} \rceil}$$

/* **3. Parameter update:** */

$$N_{\mathbf{w}}^+ = N_{\mathbf{w}} - \eta_{\mathbf{w}} \frac{\partial \mathcal{L}_{\text{Total}}}{\partial N_{\mathbf{w}}}$$

$$N_{\mathbf{a}}^+ = N_{\mathbf{a}} - \eta_{\mathbf{a}} \frac{\partial \mathcal{L}_{\text{Total}}}{\partial N_{\mathbf{a}}}$$

### 4.2.4 Regularization Hyperparameter Impact

The regularization hyperparameter $\lambda$ dictates how the task loss $\mathcal{L}_{\text{Task}}$ and the hardware complexity $\mathcal{L}_{\text{Hard}}$ are balanced out in the total loss $\mathcal{L}_{\text{Total}}$ (see Eq. (4.1)). It indirectly

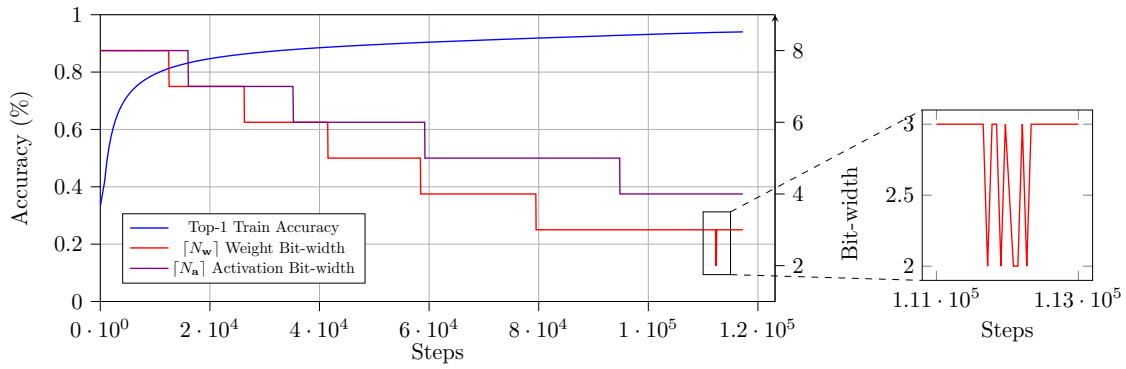



Figure 4.1: Example of applying our AdaQAT approach with a ResNet-20 network on the CIFAR-10 dataset. It showcases the evolution of the train accuracy with respect to updating the bit-width parameters $\lceil N_{\mathbf{w}} \rceil$ and $\lceil N_{\mathbf{a}} \rceil$ and how an oscillatory pattern can form (here, for the weight bit-width $\lceil N_{\mathbf{w}} \rceil$). When oscillations appear, we fix the value of the corresponding bit-width to the largest of the two oscillation levels for the rest of the QAT process, considering that the bit-width value has converged.

controls how much accuracy loss is allowed in the final DNN model compared to the weights and activations quantization levels. As can be seen in Table 4.1, a larger $\lambda$ leads to more compression but less accurate test results. Its value should be chosen carefully on a model-by-model basis, considering the application constraints (*i.e.,* how much accuracy degradation is allowed versus a certain level of attainable compression). In general, some trial and error experimentation is necessary to find an appropriate value of $\lambda$.

Table 4.1: Evolution of AdaQAT mixed-precision quantization results on CIFAR-10 with respect to $\lambda$ for a ResNet-20 model for several values of $\lambda$. The baseline `binary32` top-1 test accuracy is 92.4%. Larger values of $\lambda$ lead to a better compression ratio, but more accuracy degradation. The user has the flexibility to tweak $\lambda$ based on the application constraints to find a suitable accuracy/compression balance.

| $\lambda$ | Bit-width (W/A) | top-1 (%) |
|---|---|---|
| 0.002 | 2/4 | 91.7 |
| 0.0015 | 3/4 | 92.1 |
| 0.001 | 4/5 | 92.3 |

The values of the $\eta_{\mathbf{w}}$ and $\eta_{\mathbf{a}}$ learning rates directly impact how fast the bit-width parameters change. If they are too small, convergence to optimized values of $N_{\mathbf{w}}$ and $N_{\mathbf{a}}$ will tend to be slow and might stall beforehand. On the other hand, too big of an update

(*i.e.,* too drastic of a decrease in the bit-width values) might degrade accuracy and slow down convergence or even cause the process to diverge entirely. Through empirical testing, we settled on $\eta_{\mathbf{w}} = \eta_{\mathbf{a}} = 0.1$. In all the experiments we carried out (see Section 4.3), this bit-width refinement/exploration phase was quite fast, never exceeding 10 epochs.

### 4.2.5 Per-layer Extension

Extending the method to layer-wise precision configurations requires finding appropriate bit-width variables for each layer of the model such that, for a $L$-layer model, $\left\lceil N_{\mathbf{w}}^{(i)} \right\rceil$ and $\left\lceil N_{\mathbf{a}}^{(i)} \right\rceil$ are the bit-widths for layer $i \in [\![1, L]\!]$.

However, the method described in Algorithm 1 where the bit-width is shared by all layers cannot be straightforwardly extended to learn bit-widths by layers because the computational cost of the gradient estimation phase would grow at least linearly with the number of bit-widths to learn (*i.e.,* $\mathcal{L}_{\text{Task}}$ would need to be evaluated $2(L-2)+1 = 2L-3$ times each training iteration). Instead, it is more cost-effective to focus on updating the bit-width of only one layer's weights or activations at a time, potentially switching the layer to update at each iteration. The overall approach can thus be viewed as a quantized version of a *block coordinate descent* [Tse01] update rule. Block coordinate optimization methods update only a group of parameters at a time and have emerged as viable approaches to solve complex nonconvex optimization problems, like those involving "big data" (see for instance [XY17] and the references therein).

Careful attention must be paid when selecting which bit-width(s) to update at a particular iteration. The most straightforward approach is to take a bit-width parameter at random, but this turns out to be highly suboptimal in our case. Greedy-type approaches offer better results: at each iteration, update (*e.g.,* decrease) the bit-width most likely to degrade accuracy the least on the current training batch.

**Weights and Activations** Similarly to the per network bit-width learning context, one must decide if weight and activation bit-widths should be updated simultaneously or in sequence. Whereas in the per-network setting model accuracy is not sensitive to this choice (simultaneous and in-sequence updates tend to behave analogously), at finer granularities (*e.g.,* layer or channel) this is not the case. Taking bit-widths separately allows more freedom in exploring the search space. It is also generally the case that activations are more sensitive to perturbations than weights, meaning that model accuracy is less impacted by focusing on weights bit-width learning first and then moving on to

Table 4.2: Evolution of mixed-precision quantization results for a ResNet-20 model on CIFAR-10 with respect to the strategy used for choosing what bit-widths to update first: weights (W) or activations (A). The *simultaneous* strategy means that W and A bit-widths for a layer are updated at the same time, whereas the *separate* strategy means that all W bit-widths are updated until convergence, after which A bit-widths are optimized. The real number bit-widths correspond to the average bit-width of the whole model

| Strategy | $\lambda_{\mathbf{w}}$ | $\lambda_{\mathbf{a}}$ | Bit-width (W/A) | BitOPs ($\times 10^9$) | top-1 (%) |
|---|---|---|---|---|---|
| simultaneous | 0.02135 | | 2.78/3.26 | 0.37 | 91.5 |
| separate | 0.02135 | | 2.78/3.63 | 0.44 | 91.9 |
| separate | 0.02135 | 0.006 | 2.78/4.05 | 0.48 | 92.3 |

activation bit-widths. The fact that activation values depend on the weights (and inputs) also makes it more natural to start with weights.

A typical example of the effects from the various choices just described is given in Table 4.2. The first column indicates the weight/activation strategy employed. The second and third columns give values of the $\lambda$ parameter from Eq. (4.1), which can be the same when updating either weights or activations, or separate: $\lambda_{\mathbf{w}}$ and $\lambda_{\mathbf{a}}$. Simultaneous updates result in better compression ratios and BitOps, but accuracy is greatly degraded (1 percentage point versus the `binary32` baseline). Separate learning reduces accuracy loss (second and third rows). Having separate $\lambda_{\mathbf{w}}$ and $\lambda_{\mathbf{a}}$ introduces more flexibility in exploring the search space. Indeed, emphasizing activation bit-width optimization less (with a smaller $\lambda_{\mathbf{a}}$) improves accuracy by slightly increasing cost.

**Bit-width update selection metrics** There are various ways to measure the impact of bit-width reduction on network accuracy and/or hardware resource use. Choosing to always update the bit-width that leads to the least significant reduction (*i.e.,* with the smallest cost) is a sensible approach. During our experiments, we explored two main strategies for selecting which bit-width to update in the subsequent block coordinate descent iteration:

- *Mean Squared Error* (MSE): for two quantized tensors of the same shape $\widetilde{\mathbf{X}}_1$ and $\widetilde{\mathbf{X}}_2$ (*i.e.,* either weights or activations, before and after the corresponding bit-width parameter has been updated), the cost is their mean squared error:

$$\mathrm{MSE}\left(\widetilde{\mathbf{X}}_1, \widetilde{\mathbf{X}}_2\right) := \frac{1}{\mathrm{size}(\widetilde{\mathbf{X}}_1)} \left\|\widetilde{\mathbf{X}}_1 - \widetilde{\mathbf{X}}_2\right\|_F^2, \tag{4.6}$$

Table 4.3: Evolution of AdaQAT mixed-precision quantization results on CIFAR-10 with respect to $\lambda_{\mathbf{w}} = 0.02135$ and $\lambda_{\mathbf{a}} = 0.006$.

| Metric | Bit-width (W/A) | BitOPs ($\times 10^9$) | top-1 (%) |
|---|---|---|---|
| MSE | 2.78/4.05 | 0.48 | 92.3 |
| MSE $\times$ BitOps$^{-1}$ | 2.67/4.16 | 0.47 | 92.1 |
| BitOps$^{-1}$ | 2.00/2.84 | 0.24 | 90.5 |

Table 4.4: Evolution of AdaQAT mixed-precision quantization results on CIFAR-10 when varying $\lambda_{\mathbf{w}}$ and $\lambda_{\mathbf{a}}$.

| Metric | $\lambda_{\mathbf{w}}$ | $\lambda_{\mathbf{a}}$ | Bit-width (W/A) | BitOPs ($\times 10^9$) | top-1 (%) |
|---|---|---|---|---|---|
| MSE | 0.02135 | 0.006 | 2.78/4.05 | 0.48 | 92.3 |
| MSE $\times$ BitOps$^{-1}$ | 0.013 | 0.0042 | 2.78/4.05 | 0.47 | 92.1 |
| BitOps$^{-1}$ | 0.004 | 0.0012 | 2.78/4.05 | 0.46 | 91.5 |

where size($\widetilde{\mathbf{X}}_1$) is the number of elements in $\widetilde{\mathbf{X}}_1$ and $\|\cdot\|_F$ is the Frobenius norm.

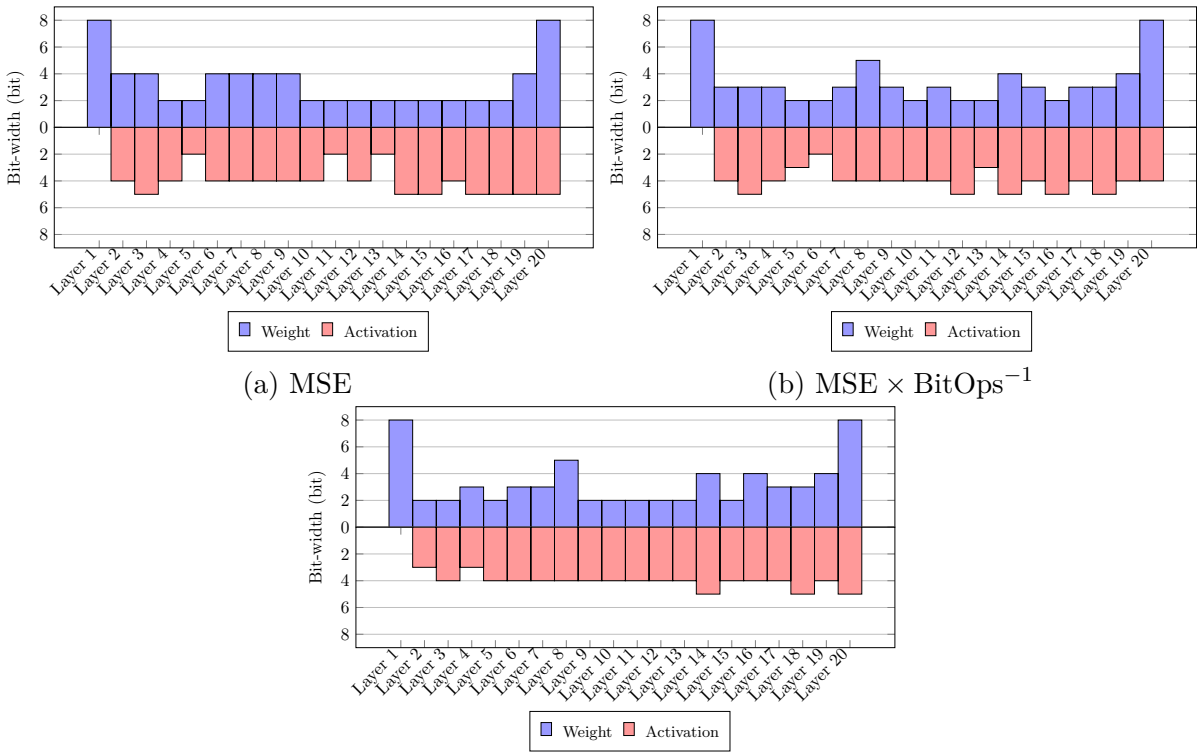
- *BitOps-Based Hardware Cost* (BitOps): the biggest hardware gains are usually correlated to the largest layers; using the bit operations metric BitOps$_i$ for each layer $i \in [\![1, L]\!]$, the corresponding cost for both weights and activations at layer $i$ is

$$\text{BitOps}_i^{-1}, \tag{4.7}$$

where the metric is computed with respect to the updated bit-width(s) at layer $i$.

We also looked at a hybrid approach in which the MSEs are weighted by the inverse of the BitOps score for the corresponding layer. The goal is to score each bit-width proportionally to the expected hardware reduction it will have on the overall model.

The effects on the same ResNet-20 + CIFAR-10 configuration as before are shown in Table 4.3 and Table 4.4. In Table 4.3, we look at the effect that the metric has on BitOps and accuracy for a fixed choice of $\lambda_{\mathbf{w}}, \lambda_{\mathbf{a}}$. The MSE configuration tends to have the highest accuracy (since it minimizes quantization errors). In contrast, the BitOps configuration leads to much more compressed models but can degrade accuracy to a considerable extent. One potential reason is that the BitOps metric does not consider any form of quantization error, choosing bit-widths that lead to the highest compression at each step but which might degrade accuracy too much. The hybrid metric tries to correct this by weighting the BitOps cost by the quantization error, leading to a better compression ratio and a




(a) MSE

(b) MSE × BitOps$^{-1}$

(c) BitOps$^{-1}$

Figure 4.2: Layer-wise mixed-precision quantization bit-widths for ResNet-20 on CIFAR-10 using the various metrics discussed in this section.

smaller accuracy drop.

By tweaking the regularization hyperparameters, one can get results with a decreased gap between the various metrics. This is exemplified in Table 4.4, where we explored choices of $\lambda_{\mathbf{w}}$ and $\lambda_{\mathbf{a}}$ for each metric that led to the best accuracy. The BitOps metric, in particular, struggles to recover accuracy, whereas the quantization error-based metrics are more robust in terms of accuracy. Figure 4.2 shows the corresponding bit-widths per layer for weights and activations. Layers at the start and end of the network tend to use higher values.

Choosing a metric is not straightforward. Quantization effects and the particularities of the target architecture should be considered. Such extensive studies of appropriate metrics are left for future work. Due to its simplicity and overall good results, we use the MSE metric for all the experiments in the rest of this chapter.

**The Algorithm** The entire procedure is listed in Algorithm 2. It follows a similar structure to Algorithm 1. In the *forward propagation* stage, weights and activations are quantized according to the current bit-width configuration (to be updated). The bit-width to be updated is first chosen among the weights if not all have converged (*i.e.,* no oscillatory pattern for any of the layers). Otherwise, an activation bit-width is chosen. This is done by examining two cost vectors initialized before calling the algorithm the first time: $\mathbf{c_w}, \mathbf{c_a} \in \mathbb{R}^{L-2}$, and selecting the corresponding layer with the minimal cost using one of the three metrics described (the `cost_metric` function), for either weights or activations. The next two stages: *gradient estimation* and *parameter update* update the bit-width configuration for the next iteration.

## 4.3 Evaluation

We use an SGD optimizer with a batch size of 256, weight decay set to $4.10^{-5}$, and momentum to 0.9. In the training from scratch scenario, weights are initialized using the Kaiming method [He+15]. We use a cosine annealing learning rate scheduler with an initial learning rate set to 0.1 for the from-scratch scenario and 0.01 for the fine-tuning scenario. Training is run for 100 epochs in the fine-tuning scenario and 150 epochs when starting from scratch. We use PyTorch 1.13 for all experiments. The ImageNet tests are run on a cluster of 8 NVIDIA V100 GPUs, whereas the CIFAR-10 ones use a single GPU configuration. Following the practice adopted by prior work regarding greater sensitivity to quantization at the input and output of a network (see for instance [YJ21]), we fix the bit-width to 8 bits in the first and last layers.

### 4.3.1 Comparison with State-of-the-Art Methods

Table 4.5 shows the results of applying AdaQAT on CIFAR-10 compared to other methods from the literature. The first line shows the `binary32` baseline result, whereas the second group of lines showcases *static* methods, where activations are not quantized and weights are quantized uniformly to 2 bits. The third group of lines corresponds to mixed-precision methods where the weight bit-width is learned, and the activations are not quantized. AdaQAT with learned weight bit-width (converged to 2 bits) and unquantized activations is on par with the best of these, both when starting from a pretrained full-precision model as well as from scratch. We should nevertheless note that the FracBits results were

**Algorithm 2** AdaQAT algorithm for training a mixed-precision $L$-layer model

**Require:** a minibatch of inputs $\mathbf{Y}^{(0)}$ and corresponding targets $\mathbf{T}$, weights $\mathbf{W} \in \mathbb{R}$, weight bit-width vector $N_{\mathbf{w}} \in \mathbb{R}_{+}^{L-2}$, activation bit-width vector $N_{\mathbf{a}} \in \mathbb{R}_{+}^{L-2}$, regularization parameters $\lambda_{\mathbf{w}}, \lambda_{\mathbf{a}} > 0$, bit-width learning rates $\eta_{\mathbf{w}}, \eta_{\mathbf{a}} > 0$, loss functions $\mathcal{L}_{\text{Task}}$ and $\mathcal{L}_{\text{Hard}}$.

**Ensure:** updated bit-width vectors $N_{\mathbf{w}}^{+}$ and $N_{\mathbf{a}}^{+}$

/* 1. Forward propagation: */

$\widetilde{\mathbf{W}}_{\text{U}}^{(1)} \leftarrow \texttt{quantize\_w}\left(\mathbf{W}^{(1)}, 8\right)$

$\mathbf{Y}_{\text{U}}^{(1)}, \mathbf{Y}_{\text{D}}^{(1)} \leftarrow \texttt{forward}\left(\mathbf{Y}^{(0)}, \widetilde{\mathbf{W}}_{\text{U}}^{(1)}\right)$

/* select the bit-width to update */

**if** all $N_{\mathbf{w}}$ did not converge **then**

  $i \leftarrow \text{argmin}\,(\mathbf{c}_{\mathbf{w}})$

**else**

  $i \leftarrow \text{argmin}\,(\mathbf{c}_{\mathbf{a}})$

**end if**

**for** $k = 2$ **to** $L - 1$ **do**

  $\widetilde{\mathbf{W}}_{\text{U}}^{(k)} \leftarrow \texttt{quantize\_w}\left(\mathbf{W}^{(k)}, \left\lceil N_{\mathbf{w}}^{(k)} \right\rceil\right)$, $\widetilde{\mathbf{W}}_{\text{D}}^{(k)} \leftarrow \texttt{quantize\_w}\left(\mathbf{W}^{(k)}, \left\lfloor N_{\mathbf{w}}^{(k)} \right\rfloor\right)$

  $\mathbf{Y}_{\text{U}}^{(k)} \leftarrow \texttt{forward}\left(\texttt{quantize\_a}\left(\mathbf{Y}_{\text{U}}^{(k-1)}, \left\lceil N_{\mathbf{a}}^{(k)} \right\rceil\right), \widetilde{\mathbf{W}}_{\text{U}}^{(k)}\right)$

  **if** k == i **then**

    **if** all $N_{\mathbf{w}}$ did not converge **then**

      $\mathbf{Y}_{\text{D}}^{(k)} \leftarrow \texttt{forward}\left(\texttt{quantize\_a}\left(\mathbf{Y}_{\text{D}}^{(k-1)}, \left\lceil N_{\mathbf{a}}^{(k)} \right\rceil\right), \widetilde{\mathbf{W}}_{\text{D}}^{(k)}\right)$

    **else**

      $\mathbf{Y}_{\text{D}}^{(k)} \leftarrow \texttt{forward}\left(\texttt{quantize\_a}\left(\mathbf{Y}_{\text{D}}^{(k-1)}, \left\lfloor N_{\mathbf{a}}^{(k)} \right\rfloor\right), \widetilde{\mathbf{W}}_{\text{U}}^{(k)}\right)$

    **end if**

  **else**

    $\mathbf{Y}_{\text{D}}^{(k)} \leftarrow \texttt{forward}\left(\texttt{quantize\_a}\left(\mathbf{Y}_{\text{D}}^{(k-1)}, \left\lceil N_{\mathbf{a}}^{(k)} \right\rceil\right), \widetilde{\mathbf{W}}_{\text{U}}^{(k)}\right)$

  **end if**

  $\mathbf{c}_{\mathbf{w}} \leftarrow \texttt{cost\_metric}\left(\widetilde{\mathbf{W}}_{\text{U}}^{(k)}, \widetilde{\mathbf{W}}_{\text{D}}^{(k)}\right), \quad \mathbf{c}_{\mathbf{a}} \leftarrow \texttt{cost\_metric}\left(\texttt{quantize\_a}\left(\mathbf{Y}_{\text{U}}^{(k-1)}, \left\lceil N_{\mathbf{a}}^{(k)} \right\rceil\right), \texttt{quantize\_a}\left(\mathbf{Y}_{\text{D}}^{(k-1)}, \left\lfloor N_{\mathbf{a}}^{(k)} \right\rfloor\right)\right)$

**end for**

$\widetilde{\mathbf{W}}_{\text{U}}^{(L)} \leftarrow \texttt{quantize\_w}\left(\mathbf{W}^{(L)}, 8\right)$

$\mathbf{Y}_{\text{U}}^{(L)} \leftarrow \texttt{forward}\left(\texttt{quantize\_a}\left(\mathbf{Y}_{\text{U,U}}^{(L-1)}, \left\lceil N_{\mathbf{a}}^{(L)} \right\rceil\right), \widetilde{\mathbf{W}}_{\text{U}}^{(L)}\right)$

**if** k == i **then**

  **if** all $N_{\mathbf{w}}$ did not converge **then**

    $\mathbf{Y}_{\text{D}}^{(L)} \leftarrow \texttt{forward}\left(\texttt{quantize\_a}\left(\mathbf{Y}_{\text{U,D}}^{(L-1)}, \left\lceil N_{\mathbf{a}}^{(L)} \right\rceil\right), \widetilde{\mathbf{W}}_{\text{U}}^{(L)}\right)$

  **else**

    $\mathbf{Y}_{\text{D}}^{(L)} \leftarrow \texttt{forward}\left(\texttt{quantize\_a}\left(\mathbf{Y}_{\text{U,D}}^{(L-1)}, \left\lfloor N_{\mathbf{a}}^{(L)} \right\rfloor\right), \widetilde{\mathbf{W}}_{\text{U}}^{(L)}\right)$

  **end if**

**else**

  $\mathbf{Y}_{\text{D}}^{(L)} \leftarrow \texttt{forward}\left(\texttt{quantize\_a}\left(\mathbf{Y}_{\text{U,D}}^{(L-1)}, \left\lceil N_{\mathbf{a}}^{(L)} \right\rceil\right), \widetilde{\mathbf{W}}_{\text{U}}^{(L)}\right)$

**end if**

/* 2. Gradient Estimation: */

**if** all $N_{\mathbf{w}}$ did not converge **then**

  $\dfrac{\partial \mathcal{L}_{\text{Task}}}{\partial N_{\mathbf{w}}^{(i)}} \approx \mathcal{L}_{\text{Task}}\left(\mathbf{Y}_{\text{U}}^{(L)}; \mathbf{T}\right) - \mathcal{L}_{\text{Task}}\left(\mathbf{Y}_{\text{D}}^{(L)}; \mathbf{T}\right) \qquad \dfrac{\partial \mathcal{L}_{\text{Total}}}{\partial N_{\mathbf{w}}^{(i)}} \approx \dfrac{\partial \mathcal{L}_{\text{Task}}}{\partial N_{\mathbf{w}}^{(i)}} + \lambda_{\mathbf{w}} \dfrac{\partial \mathcal{L}_{\text{Hard}}}{\partial \left\lceil N_{\mathbf{w}}^{(i)} \right\rceil}$

**else**

  $\dfrac{\partial \mathcal{L}_{\text{Task}}}{\partial N_{\mathbf{a}}^{(i)}} \approx \mathcal{L}_{\text{Task}}\left(\mathbf{Y}_{\text{U}}^{(L)}; \mathbf{T}\right) - \mathcal{L}_{\text{Task}}\left(\mathbf{Y}_{\text{D}}^{(L)}; \mathbf{T}\right) \qquad \dfrac{\partial \mathcal{L}_{\text{Total}}}{\partial N_{\mathbf{a}}^{(i)}} \approx \dfrac{\partial \mathcal{L}_{\text{Task}}}{\partial N_{\mathbf{a}}^{(i)}} + \lambda_{\mathbf{a}} \dfrac{\partial \mathcal{L}_{\text{Hard}}}{\partial \left\lceil N_{\mathbf{a}}^{(i)} \right\rceil}$

**end if**

/* 3. Parameter update: */

**if** all $N_{\mathbf{w}}$ did not converge **then**

  $N_{\mathbf{w}}^{(i)+} = N_{\mathbf{w}}^{(i)} - \eta_{\mathbf{w}} \dfrac{\partial \mathcal{L}_{\text{Total}}}{\partial N_{\mathbf{w}}^{(i)}}$

**else**

  $N_{\mathbf{a}}^{(i)+} = N_{\mathbf{a}}^{(i)} - \eta_{\mathbf{a}} \dfrac{\partial \mathcal{L}_{\text{Total}}}{\partial N_{\mathbf{a}}^{(i)}}$

**end if**

Table 4.5: Comparison with state-of-the-art quantization methods (ResNet-20 on CIFAR-10). Bit-width (W/A) denotes the average bit-width for weights and activation signals, whereas WCR represents the weight compression ratio w.r.t. baseline. BitOPs denotes the bit operations metric (see Sec. 4.2.1). The 4-bit activation result is learned using our method ($\lambda = 0.02135$ and $\eta = 0.1$), whereas the activation bit-widths are fixed in the 8- and 32-bit settings, with only the weight bit-widths being learned. For the mixed-precision scenario, $\lambda_w$ is set to 0.02135 and $\lambda_a$ is set to 0.006.

| Method | Bit-width (W/A) | top-1 (%) | $\Delta_{\text{acc}}$ (%) | WCR | BitOPs ($\times 10^9$) |
|---|---|---|---|---|---|
| baseline | 32/32 | 92.4 | - | - | 41.7 |
| DoReFa [Zho+16] | 2/32 | 88.2 | -4.2 | 16× | 2.7 |
| PACT [Cho+18] | 2/32 | 89.7 | -2.7 | 16× | 2.7 |
| LQ-Net [Zha+18a] | 2/32 | 91.8 | -0.3 | 16× | 2.7 |
| LQ-Net [Zha+18a] | 3/3 | 91.6 | -0.5 | 10.7× | 0.39 |
| DQ [Uhl+19] | 2.11/6.94 | 91.4 | -1.3 | 15.2× | 0.63 |
| FracBits [YJ21] | 2.00/32 | 89.6 | -2.8 | 16× | - |
| Releq [Elt+19] | 2.33/32 | - | -0.12 | 13.7× | 3.01 |
| TTQ [Jai+20] | 2.00/32 | 91.1 | -1.2 | 16× | - |
| SDQ [Hua+22] | 1.93/32 | 92.1 | -0.3 | 16.6× | - |
| HAWQ-V1 [Don+19] | 3.89/4 | 92.2 | -0.2 | 8.23× | 0.67 |
| **AdaQAT (fine-tuning)** | 2/32 | 92.0 | -0.4 | 16× | 2.7 |
| | 3/8 | 92.1 | -0.3 | 10.7× | 0.99 |
| | **3/4** | **92.2** | **-0.2** | **10.7×** | **0.51** |
| **AdaQAT (from scratch)** | 2/32 | 91.8 | -0.6 | 16× | 2.7 |
| | 3/8 | 91.8 | -0.6 | 10.7× | 0.99 |
| | **3/4** | **92.1** | **-0.3** | **10.7×** | **0.51** |
| **AdaQAT Mixed (fine-tuning)** | **2.78/4.05** | **92.3** | **-0.1** | **11.5×** | **0.48** |

obtained without fine-tuning its hyperparameters as much as possible.

The next two groups of lines in Table 4.5 illustrate the behaviour of our method when the activations are also quantized. The accuracy results are still competitive, either when starting from a pretrained model or from scratch. Even though the WCR metric is not as good as that of SDQ, it is more than compensated by the reduction in activation bit-width. It directly impacts how much memory (the BitOps column) gets transferred from one layer of the network to the next, going from 2.61 down to 0.51, more than a 5× improvement. The last groups of lines illustrate the behavior of our method when bit-widths are learned per layer.

Table 4.6 shows similar results on ImageNet compression with mixed-precision. The

Table 4.6: Comparison with state-of-the-art quantization methods on the ImageNet dataset with ResNet-18 and MobileNet-V2 in a fine-tuning setting.

| Model | Method | Bit-width (W/A) | Accuracy (%) top-1 | Accuracy (%) binary32 top-1 | WCR | BitOPs ($\times 10^9$) |
|---|---|---|---|---|---|---|
| ResNet-18 | DoReFa [Zho+16] | 4/4 | 68.1 | 70.5 | 8× | 35.2 |
| | PACT [Cho+18] | 4/4 | 69.2 | 70.5 | 8× | 35.2 |
| | LSQ [Ess+19] | 4/4 | 71.1 | 70.5 | 8× | 35.2 |
| | LQ-Net [Zha+18a] | 4/4 | 69.3 | 70.3 | 8× | 35.2 |
| | DQ [Uhl+19] | 5.11/10.4 | 70.1 | 70.3 | 6.3× | 93.6 |
| | FracBits [YJ21] | 4.00/4.00 | 70.6 | 70.2 | 8× | 34.7 |
| | SDQ [Hua+22] | 3.85/4 | 71.7 | 70.5 | 8.31× | 33.4 |
| | HAWQ-V3 [Yao+21] | 4.8/7.5 | 70.4 | 71.5 | 6.7× | 72.0 |
| | BitPruning [Nik+20] | 3.38/4.14 | 69.2 | 69.6 | 9.5× | - |
| | **AdaQAT** | **3.84/4.00** | **71.4** | **70.5** | **8.33×** | **31.4** |
| MobileNet-V2 | DoReFa [Zho+16] | 4/4 | 70.3 | 71.9 | 8× | 7.42 |
| | PACT [Cho+18] | 4/4 | 70.4 | 71.9 | 8× | 7.42 |
| | LSQ [Ess+19] | 4/4 | 70.7 | 71.9 | 8× | 7.42 |
| | SAT [JYL19] | 4/4 | 71.1 | 71.9 | 8× | 7.42 |
| | HAQ [Wan+19b] | 4.00/32 | 71.5 | 71.9 | 8× | 42.8 |
| | BitPruning [Nik+20] | 4.15/4.57 | 70.1 | 70.5 | 7.7× | - |
| | FracBits [YJ21] | 4.00/4.00 | 71.3 | 71.9 | 8× | 5.35 |
| | SDQ [Hua+22] | 3.79/4 | 72.0 | 71.9 | 8.4× | 5.07 |
| | **AdaQAT** | **3.86/3.88** | **71.3** | **71.9** | **8.28×** | **4.95** |

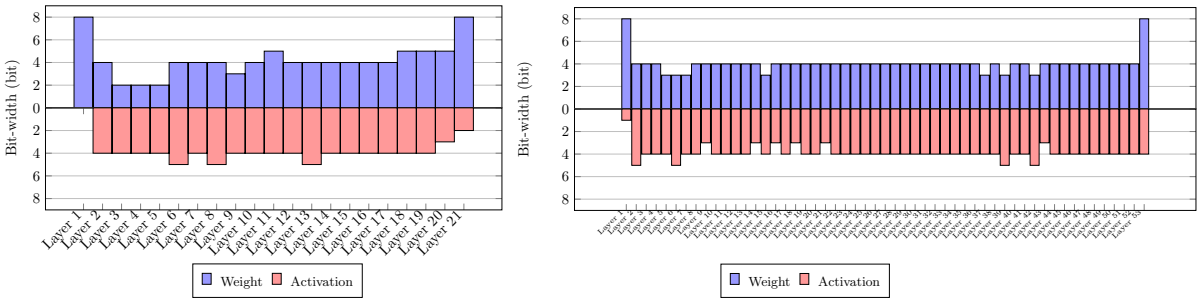
quality of the obtained quantization is comparable to other methods from the state of the art for both ResNet-18 and MobileNet-V2 models. SDQ uses knowledge distillation with a ResNet-101 teacher model and color jitter data augmentation, leading to better accuracy, but a worse BitOPs cost.

Table 4.7 compares AdaQAT with other gradient-based approaches on the exploration time needed to generate the mixed-precision configuration. While most methods require several epochs to find a suitable configuration, AdaQAT requires less than one epoch. Figure 4.3 illustrates the bit-width distribution of ResNet-18 and MobileNet-V2 models in mixed precision. We note that the bit-widths have mainly converged around a common value, due to the adopted strategy, with some variations generally of one bit.

We also tested our approach on a Thin U-Net architecture and the Airbus dataset (see Section 1.5.2). We use PyTorch 1.13 on a cluster of four NVIDIA V100 GPUs to perform these experiments. We use Adam [KB14] as the optimizer with a batch size of 32 and a Dice loss [Sud+17] as an objective function. Dice loss is a popular choice for

Table 4.7: Comparison with state-of-the-art gradient-based methods on the ImageNet dataset with ResNet-18 and MobileNet-V2 in a fine-tuning setting.

| Model | Method | # exploration epochs | # total epochs epochs | Bit-width (W/A) | Accuracy (%) top-1 | BitOPs ($\times 10^9$) |
|---|---|---|---|---|---|---|
| ResNet-18 | DQ [Uhl+19] | 50 | **50** | 5.11/10.4 | 70.1 | 93.6 |
| | FracBits [YJ21] | 120 | 50 | 4.00/4.00 | 70.6 | 34.7 |
| | SDQ [Hua+22] | 60 | 150 | 3.85/4 | **71.7** | 33.4 |
| | **AdaQAT** | **<1** | 100 | **3.84/4.00** | 71.4 | **31.4** |
| MobileNet-V2 | DQ [Uhl+19] | 50 | **50** | 5.77/- | 69.7 | 93.6 |
| | FracBits [YJ21] | 120 | 150 | 4.00/4.00 | 71.3 | 5.35 |
| | SDQ [Hua+22] | 60 | 180 | 3.79/4 | **72.0** | 5.07 |
| | **AdaQAT** | **<1** | 150 | **3.86/3.88** | 71.3 | **4.95** |




(a) ResNet-18 (b) MobileNet-V2

Figure 4.3: Layer-wise mixed-precision quantization bit-widths for ResNet-18 and MobileNet-V2 on ImageNet.

image segmentation with imbalanced datasets. The learning rate is initially set to 0.001, and we use a multi-step scheduler to divide the learning rate by 2 every 10 epochs. We train the network for 40 epochs using fine-tuning. We use random horizontal and vertical flipping, random rotation with an angle between 0 and 30 degrees, and random cropping of $256 \times 256$ pixels for position augmentation. We use random brightness (quantity of light), contrast (separation degree between darkest and brightest areas), and saturation (chromatic intensity) for color augmentation.

Table 4.8 shows the mixed-precision results of AdaQAT applied to the Thin U-Net on the Airbus Ship dataset. The two first lines correspond to quantized models with the identical manually defined bit-width for each layer. The third line is the mixed-precision Thin U-Net obtained using AdaQAT. The mixed-precision model significantly reduces the computational complexity compared with uniform bit-width quantized models while

Table 4.8: Comparison with uniform quantization methods and the mixed-precision model on the Airbus Ship dataset using a Thin U-Net 32 model.

| Quantization | Bit-width (W/A) | Accuracy (%) mIoU | Accuracy (%) binary32 mIoU | WCR | BitOPs ($\times 10^9$) |
|---|---|---|---|---|---|
| Uniform | 5/4 | 72.2 | 73.0 | 6.40× | 70.4 |
| | 4/3 | 71.4 | 73.0 | 8.00× | 42.6 |
| **Mixed** | **3.91/4.13** | **72.5** | **73.0** | **8.18×** | **54.6** |

Table 4.9: Evolution of the batch size effect on mixed-precision quantization results for ResNet-20 on CIFAR-10 for $\lambda_{\mathbf{w}} = 0.02135$ and $\lambda_{\mathbf{a}} = 0.006$.

| batch size | bit-width (W/A) | BitOPs ($\times 10^9$) | top-1 (%) |
|---|---|---|---|
| 128 | 2.78/4.05 | 0.48 | 92.3 |
| 256 | 3.39/4.53 | 0.64 | 92.6 |
| 512 | 3.39/4.95 | 0.69 | 92.4 |

maintaining a competitive accuracy.

## 4.4 Conclusion

In this chapter, we introduced a new gradient optimization-based method for mixed-precision quantization. Our method learns quantization levels for both weights and activations. Its defining characteristic is the use of relaxed fractional bit-widths that are updated using a gradient descent rule. Compared to previous approaches that are generally intended to be used in a fine-tuning setting, in early tests, our method seems to be more flexible, being capable of operating in both fine-tuning and training from scratch scenarios, producing results that are on par with state-of-the-art mixed-precision quantization approaches on CIFAR10 with a ResNet20 network. It also performs well in mixed-precision fine-tuning of ResNet18 and MobileNet-V2 on ImageNet.

Although our method is relatively powerful in finding a mixed-precision model, setting the regularization hyper-parameters $\lambda_{\mathbf{w}}$ and $\lambda_{\mathbf{a}}$ properly can be challenging. It is all the more difficult to parameterize these values correctly as mixed-precision results are widely affected by variations in the loss function. Table 4.9 shows the mixed-precision results for different batch sizes. By modifying this single hyper-parameter, we can see big changes in the final mixed-precision results. This statement is not only true for the batch size

but also applies to any change in other parameters that may affect the loss function, such as the number of epochs, the learning rate scheduler, etc. This issue is due to the finite difference approximation used to calculate the gradient of the bit-widths. A future contribution would be to work on a method for avoiding this problem, such as exploring appropriate reference values for common batch sizes [Goy17].

As another future direction, we intend to explore finer hardware complexity and energy consumption metrics, tailored for a specific target architecture (*e.g.* FPGAs), in the $\mathcal{L}_{\text{Hard}}$ term. The approach should also be extensible to finer granularities of mixed-precision quantization, such as channel-wise. We want to explore the feasibility of extending this work to other arithmetic formats as well, in particular minifloats, which are the topic of the next chapter.

Chapter 5

# Minifloat Formats for Efficient Deep Neural Network Inference

Whereas uniform quantization is widely used for DNN model compression and inference acceleration, small precision floating-point arithmetic formats, also known as *minifloats*, have been gaining much interest in recent years. One main reason is their usage in accelerating the training process of large DNNs. Another important interest is the ongoing standardization process by the IEEE (the P-3109 working group) on the usage of 8-bit floating-point formats (binary8) in Machine Learning. It is expected that such a standard will be adopted in the future by all major hardware providers that ship AI accelerator chips. While DNN training is the main motivator for adopting minifloat arithmetic, inference workloads can also take advantage of low-precision floating-point formats.

This chapter investigates using small floating-point formats for DNN inference. In Section 5.1, we start by analyzing the various features and characteristics of small floating-points and their utility in an inference setting. Based on this analysis, in Section 5.2, we introduce a floating-point encoding scheme optimized for inference based on an asymmetric exponent bias and discuss how to perform quantization with these formats. Finally, in Section 5.3, we compare this approach with uniform quantization formats in various test scenarios, highlighting its strengths and weaknesses. Part of the work presented in this chapter has been published in [Ger+23], with the corresponding codes available on Gitlab.

## 5.1 Small Floating-Point Formats

Small floating-point formats refer to many floating-point formats up to 8 bits. They have received little attention in the literature for accelerating inference on either ASICs [Wu+20; Tam+20] or FPGAs [Met+21; Agg+23]. Nevertheless, Nvidia recently announced the

support of binary6 and binary4 formats in their new Blackwell architecture without giving further details on the specifications[1]. This announcement brings new opportunities for small floating-point formats, both in terms of training and inference.

Small floating-point formats can be used following the same rules as larger, standard floating-point formats without significant modifications in the quantization flow. However, to optimize small floating-point formats for inference, special cases need to be carefully managed to limit the impact on the set of numbers that can represented. The term *minifloat* will be employed to denote the small floating-point format that incorporates our design decisions. We recall that, in IEEE-754-like formats, a non-zero normalized floating-point value $X$ with $m$-bit of mantissa and $e$-bit of exponent is represented, in binary notation, as

$$X = (-1)^s \times 1.\underbrace{x_1 \dots x_m}_{M_X} \times 2^{E_X - E_B},$$

where $s$ is the sign bit, which is 0 for positive numbers and 1 for negative numbers, $M_X \in [0, 1)$ is the $m$-bit fractional mantissa, $E_X \in \{0, 1, \dots, 2^e - 1\}$ is the integer exponent and $E_B = 2^{e-1} - 1$ is an integer exponent bias term.

**Zero encoding.** There are two distinct zero encoding strategies that are used in practice. The first approach consists of allocating an entire exponent range to encode the zero [Met+21], *i.e.,* when $E_X = 0$, which reduces the representation space in favor of simpler hardware. It is recommended to use one additional exponent bit to cover a larger dynamic range. The second approach, similar to IEEE formats, is to encode the zero in place of the smallest value [Tam+20], *i.e.,* when $E_X = 0$ and $M_X = 0$, which increases the number of representable values by $2^{m+1} - 2$ compared with the previous strategy. In Section 5.3.5, we analyze the impact of zero support and, in particular, the two strategies on accuracy. Further, in Section 5.4, we compare the hardware cost of both strategies on minifloat multiplications.

**Subnormals.** The support for subnormal numbers is directly related to the zero encoding strategy. For exponent range zero encoding, subnormals are not supported and treated as zeroes [Met+21]. In the IEEE-like zero encoding approach, choosing between supporting only normalized numbers or supporting subnormals is possible. Minifloats do not consider subnormals due to the additional hardware cost for such support. In

[1] https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/

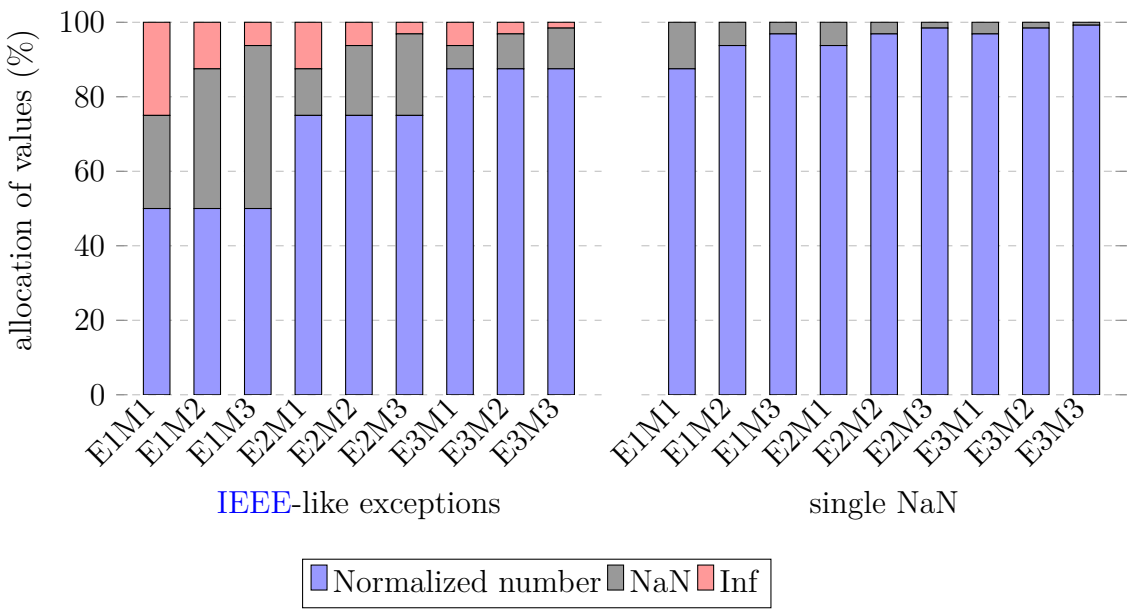



Figure 5.1: Occupancy rates of special cases NaN and Inf in various minifloat formats.

Section 5.3.5, we investigate the effect of subnormal support on accuracy.

**NaN & Inf.** Exception encoding requires extra hardware and reduces the already limited encoding space, which is extremely critical for small floating-point formats. An IEEE-type approach has a significant impact compared to supporting a single NaN, in place of the negative zero ($-0$) [Mic+22]. Figure 5.1 shows the occupancy rate of both exceptions and normalized numbers. For IEEE-like formats, a whole range of exponents is used to encode NaN and Inf, which reduces, for example, the normalized number by half for formats with a single bit of exponent. With single NaN formats, the loss of encoded numbers is greatly reduced as a single value is used to encode the NaN exception. At inference time, both encodings are not required as the model parameters learned are determined, and there can be no invalid operation or zero division. Therefore, we have chosen to not support Inf and NaN encodings. Overflows or underflows are saturated to the maximal representable number $x_{\max}$ or set to zero, respectively.

**Scaling.** As demonstrated for integer formats, scaling is essential due to the limited range of formats smaller than 8 bits. Even though small floating-points provide a larger range, scaling is critical to the small floating-point format design to be able to compete with larger formats. An exhaustive analysis of the impact of scaling and, more precisely,

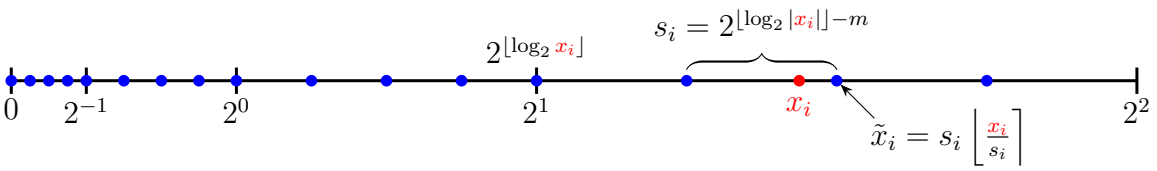



Figure 5.2: Grid depiction of the minifloat E2M2 format quantizer. $x_i$ is quantized to the nearest minifloat number in the specified format using a dynamic scaling factor dependent on the nearest integer of $x_i$.

of an asymmetric exponent range has shown the critical role of scaling in binary8 quantization [Sun+19; Nou+22]. More recent work has suggested determining this bias dynamically, such as the `AdaptivFloat` [Tam+20] method and the FFP format [HCH21], which calculate the exponent bias by examining the maximum magnitude of tensors in each layer of the network. The scaling can be better chosen by learning it using an STE [BLC13] approach. Just like the scaling factor for affine quantization, a similar method can be used to learn a real-valued exponent bias in the floating-point case [Kuz+22]. To achieve a more hardware-friendly minifloat format, we propose quantizing this learned exponent bias to an integer at layer level. In addition, we chose not to support subnormal numbers and exception encoding NaN and Inf, as well as encoding the zero in place of the smallest value.

## 5.2 Minifloat Quantization-Aware Training Scheme

Minifloat quantization can be seen as the union of $m$-bit integer quantization grids between consecutive integer powers of two [Kuz+22], as illustrated by Figure 5.2. Summarized in Algorithm 3, minifloat quantization can be performed using the same method as for uniform quantization with the distinction that each element $x_i$ in a tensor has its associated scale $s_i$. The quantized minifloat value $\tilde{x}_i$ (line 6) is computed as

$$\tilde{x}_i = s_i \left\lfloor \frac{x_i}{s_i} \right\rceil ,$$

where $\lfloor \cdot \rceil$ indicates rounding to the nearest integer. The value of $\tilde{x}_i$ must be clipped to ensure that it can be represented given $m$, $e$, and $E_B$. The authors of [Kuz+22] experimentally found that learning the maximum clipping value $c_{\max}$ instead of the exponent

**Algorithm 3** `quantize`: minifloat quantization algorithm

**Require:** real-valued tensor $\mathbf{X}$, floating-point format E$e$M$m$, learned max clipping $c_{\max}$.
**Ensure:** quantized tensor $\widetilde{\mathbf{X}}$ in the E$e$M$m$ format.

1: $E_B = 2^e - 1 + \lceil \log_2(2 - 2^{-m}) - \log_2(c_{\max}) \rceil$
2: $x_{\text{small}} = (1 + 2^{-m}) \cdot 2^{-E_B}$ *// No subnormal support here*
3: $x_{\max} = (2 - 2^{-m}) \cdot 2^{2^e - 1 - E_B}$
4: $\mathbf{X}_{\text{c}} = \texttt{clamp}(\mathbf{X}, -x_{\max}, x_{\max})$
5: **for** $i = 0$ **in X do**
6: **if** $\lfloor \log_2 |x_i| + E_B \rfloor > 0$ **then**
7: $s_i = 2^{\lfloor \log_2 |x_i| + E_B \rfloor - E_B - m}$
8: **else**
9: $s_i = 2^{1 - E_B - m}$
10: **end if**
11: **end for**
12: $\widetilde{\mathbf{X}} = \left\lfloor \frac{\mathbf{X}_{\text{c}}}{\mathbf{S}} \right\rceil \cdot \mathbf{S}$
13: $\widetilde{\mathbf{X}} = \widetilde{\mathbf{X}} \cdot \mathbb{I}_{|\widetilde{\mathbf{X}}| \geq x_{\text{small}}}$
14: **return** $\widetilde{\mathbf{X}}$

bias improves training stability. The maximum clipping is then used to calculate the exponent bias $E_B = 2^e - 1 + \lceil \log_2(2 - 2^{-m}) - \log_2(c_{\max}) \rceil$ (line 1). As maximum clipping is not in the targeted minifloat format, it must be adapted (line 3). Values of $|\tilde{x}_i|$ greater than the maximum representable value are clipped to $x_{\max} = (2 - 2^{-m}) \times 2^{2^{e-1} - 1 - E_B}$ (line 4). In the same way, values below the smallest non-zero representable value, $x_{\text{small}} = (1 + 2^{-m}) \times 2^{-E_B}$, are rounded to zero (line 7). Note that the smallest non-zero differs when subnormal numbers are supported (see section 5.3.5). To accommodate the asymmetric exponent bias, the scaling factor $s_i$ is computed as follows:

$$s_i = \begin{cases} 2^{\lfloor \log_2 |x_i| + E_B \rfloor - E_B - m} & \text{if } \lfloor \log_2 |x_i| + E_B \rfloor > 0 \\ 2^{1 - E_B - m} & \text{otherwise.} \end{cases} \tag{5.1}$$

To allow gradients to flow through each step of the quantizer, STE is used for gradients of non-differentiable rounding operations, similar to its use with uniform quantizers. The quantizer gradient that ignores the rounding operation and approximates it with an

identity function can be computed as

$$\frac{\partial \mathcal{L}}{\partial \tilde{x}_i} = \begin{cases} 1 & \text{if } x_{\text{small}} \leq |x_i| \leq x_{\text{max}} \\ 0 & \text{otherwise.} \end{cases}$$

Pruned values, *i.e.,* values below a certain threshold $x_{\text{small}}$ receive no gradient since we use the identity as suggested in some pruning-aware training techniques [VD23].

The gradient of the maximum clipping value can be computed using

$$\frac{\partial \mathcal{L}}{\partial c_{\text{max}}} = \begin{cases} \frac{2^{\lfloor \log_2 |x_i| + E_B \rfloor - E_B - m}}{x_{\text{max}}} \left( \left\lfloor \frac{x_c}{s_i} \right\rceil - \frac{x_c}{s_i} \right) & \text{if } \lfloor \log_2 |x_i| + E_B \rfloor > 0 \text{ and } |x_i| \leq x_{\text{max}} \\ \frac{1}{x_{\text{max}} \ln(2)} & \text{if } \lfloor \log_2 |x_i| + E_B \rfloor \leq 0 \text{ and } |x_i| \leq x_{\text{max}} \\ -1 & \text{if } x_i < -x_{\text{max}} \\ 1 & \text{if } x_i > x_{\text{max}} \end{cases}$$

where $\lfloor \log_2 |x_i| + E_B \rfloor$ is treated as a constant that receives no gradient. This prevents the sometimes extremely large gradients of this operation from propagating backward.

## 5.3 Experiments

For our experiments on the Airbus Ship, CIFAR-10, and ImageNet datasets, we used the same training settings as in the previous chapters.

### 5.3.1 Comparison of Quantization Formats

We have compared our minifloat formats with various uniform formats, including fixed-point and integer arithmetic alternatives, two quantization formats commonly used to accelerate inference. The results of applying minifloat quantization to ship detection and the more conventional CIFAR-10 dataset are summarized in Table 5.1. The scaling factor column indicates the method used to determine the scaling factor. The zero encoding column corresponds to the value reserved for encoding the zero. The last column is the accuracy of the models, top-1 for CIFAR-10 and mIoU for Airbus. Although all formats achieve decent accuracy on CIFAR-10, minifloats perform slightly better than uniform formats for a given bit-width. On the Airbus Ship dataset, while fixed-point quantization seems to degrade model accuracy significantly, low-precision floating-point variants are competitive with integer-based alternatives for very low quantization levels. 5-bit floating-point formats are necessary for this complex task to match single precision accuracy.

Table 5.1: Comparison of prediction accuracy for the ResNet-20 model on the CIFAR-10 dataset and for the Thin U-Net 32 model on the Airbus Ship dataset, with different arithmetic formats and bit-widths. unsigned minifloat formats are denoted here with a "u".

| Dataset | Format | Bit-width | | Scaling factor | Zero encoding | Accuracy (%) |
|---|---|---|---|---|---|---|
| | | Weight | Activation | | | |
| CIFAR-10 | binary32 | E8M23 | E8M23 | - | $M_X = 0$ & $E_X = 0$ | 92.4 |
| | Fixed-point | 3 | 3 | $2^{\lceil \log_2(\max \lvert \mathbf{X} \rvert) \rceil}$ | Zero point $= 0$ | 90.2 |
| | Integer | 3 | 3 | learn | | 91.6 |
| | Minifloat | E1M1 | E1M1 | learn | $M_X = 0$ & $E_X = 0$ | 91.2 |
| | | E1M1 | uE2M1 | | | 91.9 |
| | | E1M1 | uE1M2 | | | 91.9 |
| Airbus Ship | binary32 | E8M23 | E8M23 | - | $M_X = 0$ & $E_X = 0$ | 73.0 |
| | Fixed-point | 6 | 5 | $2^{\lceil \log_2(\max \lvert \mathbf{X} \rvert) \rceil}$ | Zero point $= 0$ | 44.5 |
| | Integer | 6 | 5 | learn | | 72.3 |
| | | 5 | 4 | | | 72.2 |
| | | 4 | 3 | | | 71.4 |
| | Minifloat | E4M2 | E4M2 | learn | $M_X = 0$ & $E_X = 0$ | 72.7 |
| | | E3M2 | uE3M2 | | | 72.7 |
| | | E2M2 | uE2M2 | | | 72.6 |
| | | E3M1 | uE3M1 | | | 72.2 |
| | | E2M1 | uE2M1 | | | 71.9 |

When single ships are present in the image, our quantized model usually does a good job of detecting them, as illustrated in Figure 5.3a. Detecting small ships, as well as side-by-side ships and inshore ships, is more challenging than detecting single large ones, leading to poorer predictions even with single-precision models. Figure 5.3b provides prediction results with minifloats (E3M2) for images with small ships in the harbor together with the original image and its associated masks in ground truth and single-precision. Figure 5.3c shows the same prediction results but for an image with multiple ships and side-by-side ships. Inshore [Nie+18] and small ship [Zha+19] detection are challenging topics, both subject to active research.

### 5.3.2 Maximum Clipping Initialization

Just like with the scaling factors [Bha+20] for uniform quantization, a good initialization of the exponent bias parameter is key to a faster convergence with good accuracy. Table 5.2 presents three strategies for initializing maximum clipping and the training loss after the first epoch. The strategy *None* is to initialize the maximum clipping to a

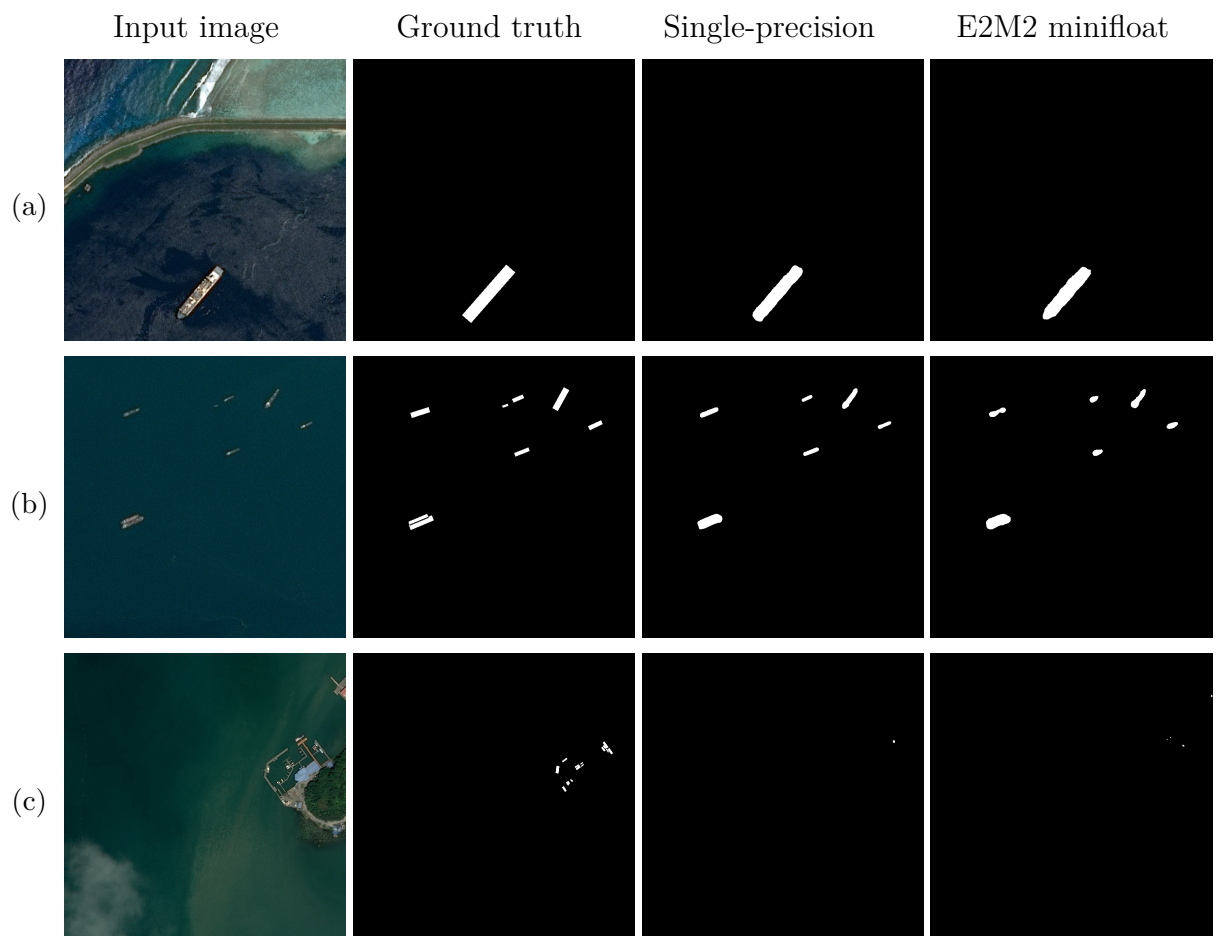



Figure 5.3: sample input images with a selection of scenarios, a) a single large ship b) multiple ships and side-by-side ships c) small ships in the harbor, and associated masks.

large value, e.g. 10. The other two strategies use the model parameters and a small set of calibration data, either using the maximum or the standard deviation as the starting point. Our results show that good initial estimates can be determined by first training the network for a small number of iterations without optimizing the maximum clipping (we did this for one epoch in our tests) and then picking it based on the standard deviation in the weight and activation tensors seen during this process, leading to

$$x_{\max} = 3 \times \sigma_{\mathbf{X}},$$

where $\sigma_{\mathbf{X}}$ is the standard deviation of tensor $\mathbf{X}$. the standard deviation initialization allows the model to ignore very large values that can impact the representable number set in the minifloat format.

Table 5.2: Dice loss scores of E2M2 Thin U-Net according to different initialization strategies after the first epoch initialization phase

| Initialization strategy | Loss |
|---|---|
| None | 0.85478 |
| max | 0.39277 |
| $\sigma_{\mathbf{X}}$ | 0.23295 |

### 5.3.3 Sensitivity of the First and Last Layers

The first and last layers of CNN models are generally more sensitive to quantization. While it was common to keep these layers in single precision, quantizing them with 8-bit formats is today the most widely adopted strategy for affine quantization, as it does not significantly affect the network's overall performance. We evaluated the impact of these layers for minifloat quantization. Our results are summarized in Table 5.3. We highlight that 6-bit quantization for both the first and the last layers is sufficient to not significantly impact the accuracy of the Thin U-Net model, leading to slightly better compression ratios. These results can undoubtedly be explained by the larger range of minifloat formats compared to uniform formats. Our results also suggest that the first layer is slightly more sensitive to quantization than the last.

Table 5.3: Comparison of prediction accuracy for the Thin U-Net 32 model on the Airbus Ship dataset regarding the quantization of the first and last layer

| Bit-width | | first layer | last layer | mIoU |
|---|---|---|---|---|
| Weight | Activation | | | |
| E8M23 | E8M23 | E8M23 | E8M23 | 73.0 |
| E3M2 | E3M2 | E4M3 | E4M3 | 72.8 |
| | | E3M2 | E3M2 | 72.7 |
| E2M2 | E2M2 | E4M3 | E4M3 | 72.5 |
| | | E3M2 | E3M2 | 72.6 |
| | | E2M2 | E2M2 | 72.0 |
| | | E3M2 | E2M2 | 72.4 |
| | | E2M2 | E3M2 | 72.1 |

Table 5.4: Comparison of prediction accuracy for the E2M2 Thin U-Net 32 model on the Airbus Ship dataset regarding different scaling strategies.

| related work | quantization scheme | scaling factor | mIoU |
|---|---|---|---|
| IEEE-like | QAT | $2^{2^{e-1}}$ | 66.7 |
| [Set+18; Met+21] | | $\alpha$ | 72.6 |
| [Kuz+22] | | $2^{\beta}$ | 72.8 |
| AdaptivFloat [Tam+20] | | $2^{\lceil \log_2(\max(\mathbf{X})) \rceil}$ | 68.4 |
| our | | $2^{\lceil \beta \rceil}$ | 72.6 |
| AdaptivFloat [Tam+20] | PTQ | $2^{\lceil \log_2(\max(\mathbf{X})) \rceil}$ | 57.7 |
| our | | $2^{\lceil \log_2(3 \times \sigma_{\mathbf{X}}) \rceil}$ | 69.6 |

### 5.3.4 Minifloat Scaling

We evaluated our scaling strategy based on learned exponent bias against existing SoA scaling methods, confirming the need for scaling when using minifloat formats. Results are reported in Table 5.4 for the E2M2 Thin U-Net 32 model on the Airbus Ship dataset. Our learning bias method outperforms techniques using weight dynamics to determine the exponent bias, such as `AdaptivFloat` [Tam+20]. We also found that an integer exponent bias learned during training remained competitive with methods based on real scale factors. Moreover, we have conducted PTQ experiments using the standard deviation instead of the maximum, improving the quantized model's performance.

Looking at each layer's learned exponent biases, we find that the values are similar except for the first and last layers, as illustrated in Figure 5.4. This expected difference is explained by the higher precision used since the maximum exponents are identical for each layer, which suggests the importance of low values for these layers. There is also a significant difference between the exponent biases learned for the weights and the activations. We notice that the maximum exponent is relatively tiny for weights, favoring small values, unlike activations, which have larger maximum exponents.

### 5.3.5 Zero Encoding and Subnormal Support

Zero representation is essential to neural network computations, particularly for activation functions. The popular ReLU activation function, which outputs zero for negative inputs and the input value for a positive input, has significantly contributed to the success of Deep Learning. Moreover, CNNs often use zero-padding around the input image to preserve the

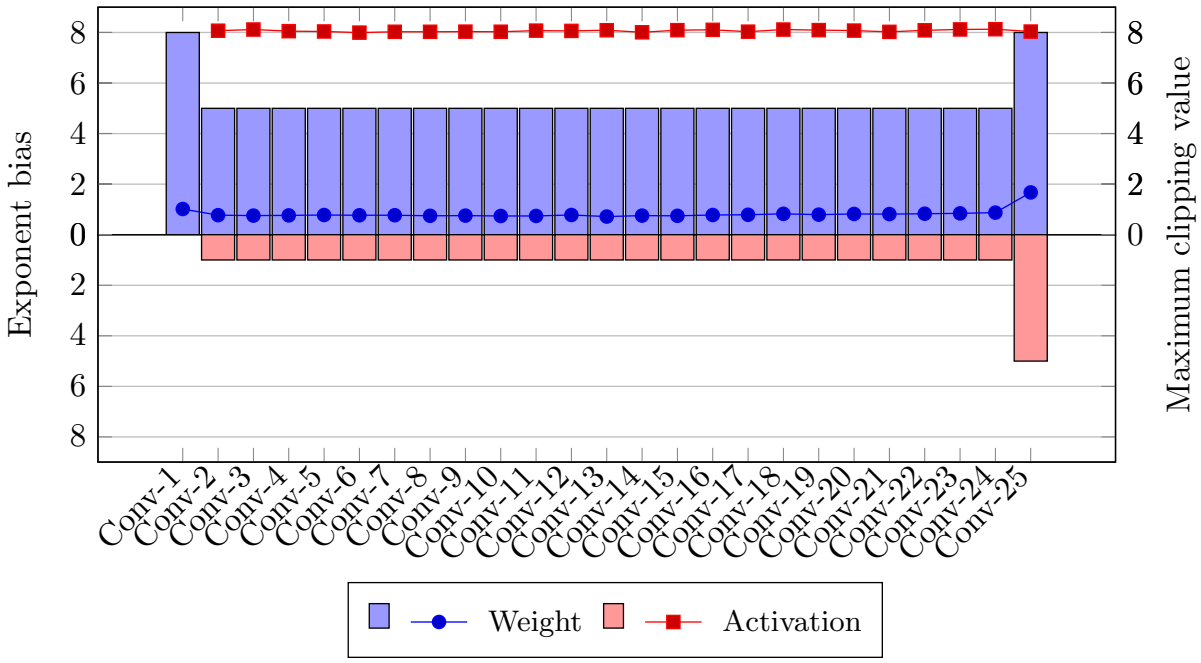



Figure 5.4: Graphical representation of the learned exponent bias values for the layers of a Thin UNet model with E2M2 minifloat quantization. The bars represent the final bias values (integer), and the curves correspond to the maximum clipping values learned during training. Large clipping values lead to small biases and vice versa.

spatial dimensions through convolutions. Our experiments, presented in Table 5.5, show that the model performs poorly when zero is not supported, confirming the importance of supporting a zero representation.

However, zero encoding is not sufficient to recover accuracy. This gap can be significantly reduced by supporting subnormal numbers, but it requires extra hardware. As outlined previously, rescaling also allows the model to recover accuracy, specifically when using an integer exponent bias. In our experience, combining rescaling with subnormals does not significantly increase accuracy compared to rescaling alone.

### 5.3.6 Impact of Stochastic Rounding

The benefits of stochastic rounding for training quantized models have already been demonstrated [HF92; Gup+15; CBD15], including work on floating-point [Wan+18; Mel+19a]-based quantization. We have evaluated the impact of stochastic rounding on minifloat-based QAT retraining. Contrary to expectations, training with stochastic rounding did

Table 5.5: Comparison of prediction accuracy for the Thin U-Net 32 model with E2M2 precision for both weights and activations on the Airbus Ship dataset with different zero encoding strategies and subnormal support.

| zero encoding | subnormals | scaling | mIoU |
|---|---|---|---|
| None | ✗ | ✗ | 35.9 |
| $E_X = 0$ | ✗ | ✗ | 50.2 |
| | ✗ | ✓ | 71.6 |
| $M_X = 0$ and $E_X = 0$ | ✗ | ✗ | 66.7 |
| | ✓ | ✗ | 71.1 |
| | ✗ | ✓ | 72.6 |
| | ✓ | ✓ | 72.8 |

Table 5.6: Comparison of prediction accuracy for the Thin U-Net 32 model on the Airbus Ship dataset with different rounding modes.

| Rounding | Bit-width | | mIoU |
|---|---|---|---|
| | Weight | Activation | |
| Nearest | E3M2 | E3M2 | 72.7 |
| | E2M2 | E2M2 | 72.6 |
| | E3M1 | E3M1 | 72.2 |
| Stochastic | E3M2 | E3M2 | 72.1 |
| | E2M2 | E2M2 | 72.1 |
| | E3M1 | E3M1 | 71.7 |

not improve the model's performance in our setting. Results comparing stochastic rounding with round-to-nearest are reported in Table 5.6 for the Thin U-Net 32 model with different precision for both weights and activations on the Airbus Ship dataset.

## 5.4 Hardware Implementation Aspects

Our results so far suggest that minifloat quantization is potentially a good choice for low-precision inference acceleration. However, floating-point addition is generally more resource-intensive than its integer/fixed-point counterpart. This is somewhat counterbalanced by the multiplier, which in the case of minifloats can be implemented efficiently using just Look-Up Tables (LUTs), as opposed to an 8-bit integer variant that would

require DSP blocks, which are much lower in number than LUTs on modern FPGAs (*e.g.*, in an AMD-Xilinx UltraScale+ VUP13 FPGA, for every DSP block there are 140 6-input LUTs [Met+21]). DSPs can then be configured to implement adder trees for the accumulation part of MAC units, improving overall logic density. For instance, the results presented by AMD-Xilinx in [Met+21] claim 60% higher performance and 12.5% memory traffic reduction (also leading to lower power usage) when using a minifloat E3M3 format as opposed to INT8 in a ResNet-50 accelerator. To achieve this performance, they implement as the basic building block a hybrid MAC operator that combines the best of both worlds, a LUT-based minifloat multiplier and a simpler fixed-point adder.

Figure 5.5 shows a minifloat E$e$M$m$ multiplier adapted from [Met+21]. Compared to our minifloat formats where zeros are encoded with $M_X = 0$ and $E_X = 0$, the formats from [Met+21] are slightly different, opting for a larger range of zero code words corresponding to $E_X = 0$. To support our zero encoding, we need to add the input mantissas to the detect zero block and set the output mantissa to zero when we detect an input zero.

While the AMD-Xilinx architecture of [Met+21] leads to a slightly more efficient minifloat multiplier design, our synthesis results using Verilog and Vivado 2022.1 with a Zynq UltraScale+ ZCU102 as target show that the differences in LUT count between the two zero encoding choices are modest. Figure 5.6 reports the synthesis results in LUT count between the zero encoding with $E_X = 0$ from [Met+21] with our proposed encoding scheme with $M_X = 0$ and $E_X = 0$. The extra encoding space saved by our choice has a positive impact on accuracy, as already stated in Table 5.1, for the E2M2 and E3M2 formats. We believe these properties make the $M_X = 0$ and $E_X = 0$ encoding a better choice in practice.

The authors of [Met+21] suggest adding an extra exponent bit to balance the reduction in value space with a zero encoded as $E_X = 0$. Considering this extra bit, our zero encoding results in an even or a lower LUT usage for the multiplier.

### Exponent Bias

The AMD-Xilinx ResNet-50 implementation uses a real-valued scaling factor requiring the incorporation of a fixed-point multiplier before converting to minifloat (Figure 5.7a). With an integer exponent bias, as we propose, the logic needed to handle its propagation (*e.g.* in convolution operations) would amount to a simpler integer exponent shift (Figure 5.7b). AMD-Xilinx did not detail the design of the fixed-to-float conversion unit, so

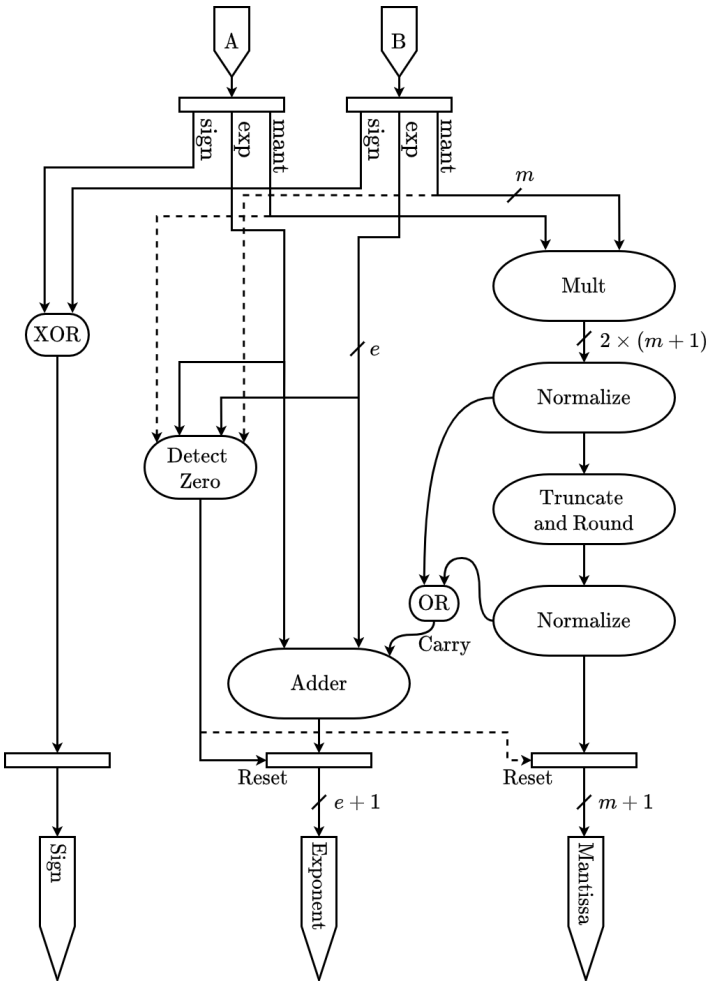



Figure 5.5: Proposed implementation of a minifloat E$e$M$m$ multiplier (based on [Met+21]). The minifloat multiplier preserves exponent accuracy by using a $(e+1)$-bit exponent output and truncates the mantissa output to $m+1$-bit, reducing the size of the logic needed to convert minifloat to fixed point in a hybrid minifloat fixed-point MAC design. To implement the $E_X = 0, M_X = 0$ zero encoding at the hardware level, we add the input mantissas as inputs to the *Detect Zero* block (dashed lines). This slightly complicates the zero detection logic compared to the $E_X = 0$ design from [Met+21].

we adopted the architecture proposed by the authors of [De +08] for our experiments.

Table 5.7 shows the resource usage (LUT and DSP) of the two architectures illustrated in Figure 5.7. We consider two alternative designs, with and without DSP, for the architecture employing a scaling factor. In the case of a hybrid MAC design, our scaling approach leads to simpler logic when converting from integers to minifloat. Suppressing

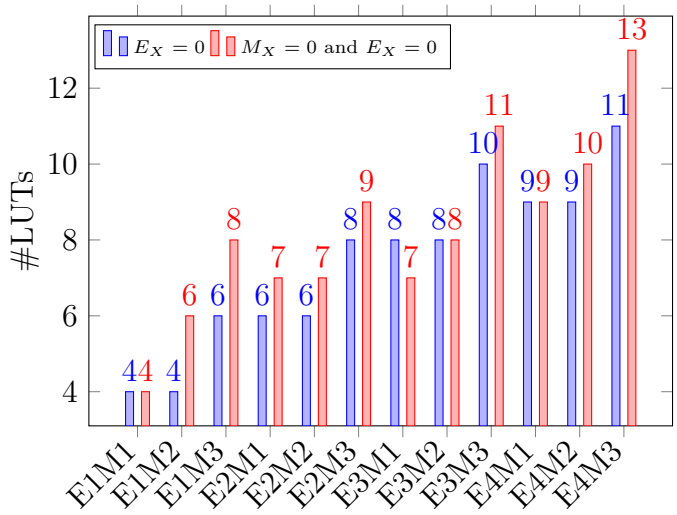



Figure 5.6: LUT consumption when implementing a minifloat multiplier with the two different zero encodings: $E_X = 0$ and our proposal $E_X = 0$ and $M_X = 0$.

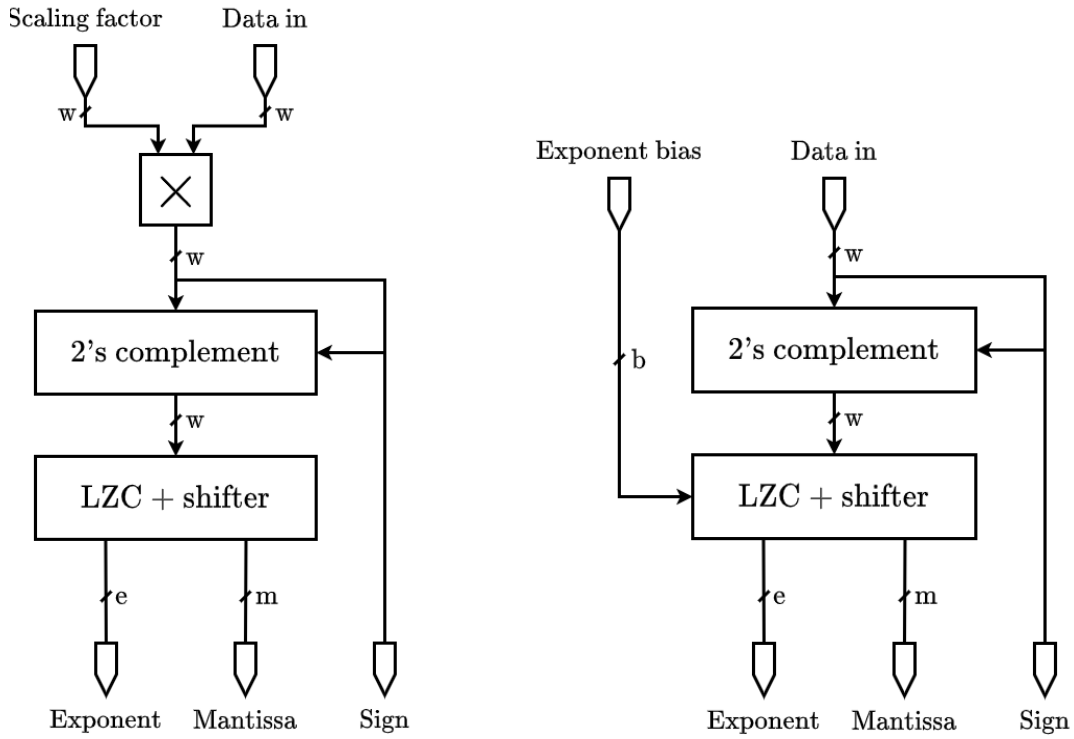
Table 5.7: FPGA resources usage for 24-bit fixed-point and 12-bit fractional inputs and E4M3 minifloat outputs.

| Resources | Xilinx | | Our exponent bias |
|---|---|---|---|
| | with DSP | without DSP | |
| LUT | 93 | 686 | 91 |
| DSP | 2 | 0 | 0 |

the fixed-point multiplier saves the use of two DSPs and a few LUTs, or even reduces the number of LUTs by 86.7% in the case of an implementation using only LUTs.

## 5.5 Conclusion

In this chapter, we analyze the features and characteristics of minifloat formats. This analysis leads us to introduce a floating-point encoding scheme optimized for inference and its quantization procedure. We have proposed a QAT algorithm for learning compressed low-precision floating-point DNN models. In addition, we learn the exponent biases of each layer for both weights and activations. Our experiments on the Airbus Ship dataset show promising results, with low-precision floating-point models being competitive with single-precision baselines and affine quantization-based alternatives.




(a) Xilinx scaling factor

(b) Our Exponent Bias

Figure 5.7: Implementation of fixed-to-float conversion architecture integrating scaling. The architecture proposed by [Met+21] manages the real scaling factor by multiplying it with the result of the matrix operation. In contrast, our proposal integrates scaling with an exponent bias into the fixed-to-float conversion unit. This unit consists of conversion from 2's complement to sign/magnitude followed by leading-zero counting and shifting and rounding

To show the potential impact of using floating-point data formats, we have also suggested an implementation of a minifloat-enabled multiplier based on [Met+21] that can be used as a basis for a full DNN inference accelerator for our models. Future work will involve designing an accelerator for the Thin U-Net 32 model based on minifloats to better assess the feasibility of using low-precision floating-point data in an on-board spatial processing context. Although quantization is a very effective compression technique, it can be associated with architectural model optimization for further compression, as we will see in the next chapter.

Chapter 6

# Lightweight Semantic Segmentation Model for Ship Detection

Over the past few years, Deep Learning has become ubiquitous for Computer Vision tasks, and semantic segmentation has been no exception. The popularisation of Fully Convolutional Networks (FCNs) with an encoder-decoder architecture has led to much progress in this field. In particular, the U-Net model [RFB15] is widely used in many semantic segmentation tasks as a baseline model. In the quest for accuracy, state-of-the-art neural networks often disregard the complexity of their architectures, which typically feature millions of trainable parameters. For instance, the U-Net model comprises 31 million parameters for approximately 288MB. Hence, these architectures require massive computing power and energy, making deployment of such models on embedded systems challenging.

Several recent research efforts have focused on reducing this complexity while maintaining good prediction performance by developing more efficient models. Models based on the U-Net architecture have been developed in this direction, adjusting depth [VXN20], convolution modules [Meh+18; Ans+22], or encoder backbone [EAA19]. However, one key feature of U-shaped architectures has received little attention considering its memory footprint: skip connections. They are even more critical since data movements are more energy-intensive than computation [Hor14], and the skip connections in U-shaped architectures imply significantly more data movements than in other, more conventional models.

In this chapter, we aim to evaluate the impact of skip connections on the accuracy of a U-shaped segmentation model and propose a more lightweight model. In Section 6.1, we briefly review specific segmentation models used for space applications with a strong focus on one feature: long skip connections. Then, in Section 6.2, we analyze the impact of

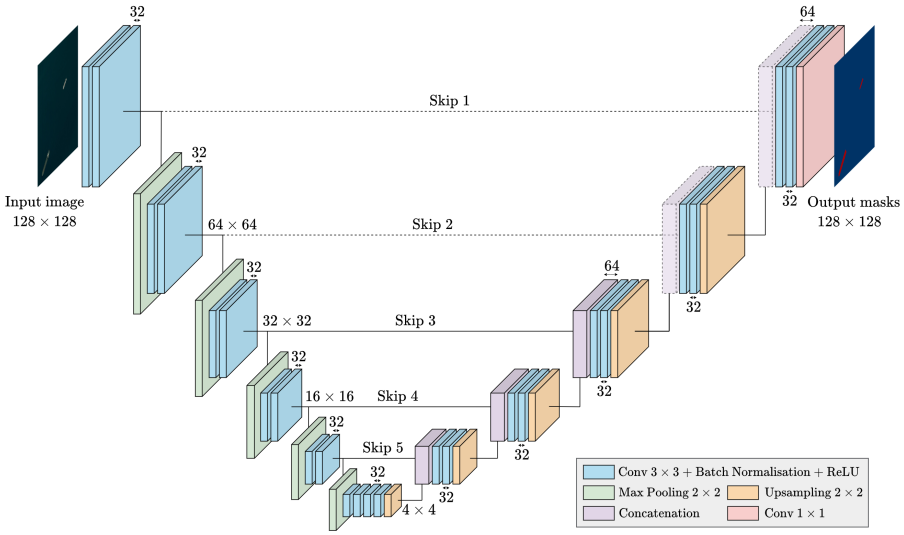



Figure 6.1: The Lightweight Thin U-Net 32 architecture. In this scenario, skips 1 and 2 are removed (dashed line) and skips 3, 4 and 5 are kept untouched (solid line).

skip connections on the Thin U-Net accuracy for ship detection to propose a lightweight version. Lastly, in Section 6.3, we present an implementation of the lightweight Thin U-Net on FPGA and analyze the gain in required resources.

## 6.1 U-shaped Networks for Semantic Segmentation

We already briefly introduced the semantic segmentation task and specific features of encoder-decoder models in Section 1.5.2. In this section, we examine in more detail U-shape models used in the space industry, some lightweight models, and one specific feature common to the different models: skip connections.

Just like with other segmentation tasks, U-shaped networks (an example was already illustrated in Figure 1.13 for the Thin U-Net 32 architecture reminded by Figure 6.1) are widely used in the space industry, just like for cloud detection [Giu+21] or urban area segmentation [ALL16]. Therefore, it is not surprising to find this popular architecture as the backbone network in optical ship detection [Li+18a; Sha+19; Mao+20; KK21].

The U-Net [RFB15] architecture has significantly impacted DL-based semantic segmentation by integrating long-range skip connections. The skip connection mechanism has proven effective in recovering fine-grained details in target objects [Dro+16; Hua+17] and particularly high-frequency components such as ship boundaries. A wide range of

architectural modifications have been proposed, and variants of the U-net architecture consistently outperform other architectures that still use long skip connections [Zho+18; Okt+18; Li+18b; Alo+18a; Hua+20].

All these models are generally large and unsuitable for embedded systems. Few lightweight models have yet been evaluated for on-board image processing. Moreover, the necessity of this skip connection is questioned as they do not always benefit the segmentation performance [Wan+22]. In addition, skip connections are memory-intensive, particularly long skips, as they require feature maps to be held in memory for a long time. Consequently, they cannot easily be deployed on devices with limited memory, especially for dataflow architecture in which they are hard-coded. As an example, C U-Net [Bah+19] is a smaller version of the U-Net model used for cloud detection with the distinctive characteristic of removing all skip connections at the cost of a significantly degraded accuracy.

## 6.2 Analysis of Skip Connection Suppression

We evaluate the impact of removing skip connections by experimenting with different configurations. We used the same training settings as in the previous chapters for our experiments on the Airbus Ship dataset. Results on the Airbus dataset are given in Table 6.1. Skip connections are numbered from 1 to 5, starting with the largest and ending with the smallest, respectively, as shown in Figure 6.1. The memory footprint corresponds to the accumulated memory of skip connections for $128 \times 128$ input images. Note that we changed the up-sampling algorithm for nearest interpolation, which slightly affected the single precision baseline accuracy compared to the previous chapter. Removing all skip connections, as suggested in the C U-Net model [Bah+19] for cloud detection (auto-encoder), significantly deteriorates the model's performance, implying that the removed connections are critical for ship detection. A more detailed analysis of the different skip connections' memory impact shows that skip connections 1 and 2 represent 94% of the memory footprint, with 75% corresponding to just the first skip connection. Hence, removing the two memory-intensive skip connections dramatically reduces the memory requirements. The reduction also depends on the input image size. For example, with input images of size $128 \times 128$, deleting skip connections 1 and 2 saves 21 Mb of memory. In comparison, the memory footprint of the model parameters is 7.9 Mb. The experiment shows that removing these two skip connections does not significantly compromise the

Table 6.1: Comparison of prediction accuracy on the Airbus Ship dataset for different skip connection removal configurations and depth, the number of convolution kernels. Skip connections are numbered from 1 to 5, starting with the largest and ending with the smallest, respectively, as shown in Figure 1.13. The memory footprint corresponds to the accumulated memory of skip connections for $128 \times 128$ input images.

| Architecture | Depth | Skip connections | | mIoU (%) |
|---|---|---|---|---|
| | | index removed | memory (Mb) | |
| Thin U-Net | 32 | - | 21.3 | 72.8 |
| | 64 | - | 40.7 | 73.0 |
| Auto-encoder | 32 | 1-2-3-4-5 | 0 | 54.2 |
| | 64 | 1-2-3-4-5 | 0 | 54.5 |
| | 128 | 1-2-3-4-5 | 0 | 54.6 |
| Lightweight Thin U-Net | 32 | 1-2 | 1.3 | 72.0 |
| | 32 | 1-2-3 | 0.3 | 62.7 |

resulting model accuracy. Figure 6.1 illustrates the proposed Lightweight Thin U-Net with removed skip connections 1 and 2. Lastly, increasing the model depth does not compensate for the suppression of skip connections and offers marginal gains in accuracy compared with expanding the model memory footprint.

Skip connections transfer high-frequency information to the decoder since the encoder tends to act as a low-pass filter due to successive convolutions. In particular, satellite images of the sea mainly comprise high-frequency information, including waves and sea foam. They are also essential for ship detection, as they form ship boundaries. Looking further into the memory reduction of skip connections by removing skip connection 3, we notice a significant drop in accuracy. This loss is mainly due to the number of small ships not detected by the further compressed model.

Figure 6.2 represents the feature maps at skip connections 3 and 4 for a small ship and a large ship. The figure also showcases the two original images (left column). Visual analysis of intermediate skip connection 3 feature maps shows sufficient preservation of ship location information, regardless of ship size (middle column). Skip connection 4 feature maps, on the other hand, are more affected by the information reduction, especially for small ships (right column). For some feature maps, the loss of information results in the deletion of small ship location information. The visual analysis of the feature maps highlights the importance of the intermediate skip connection 3 as a critical component

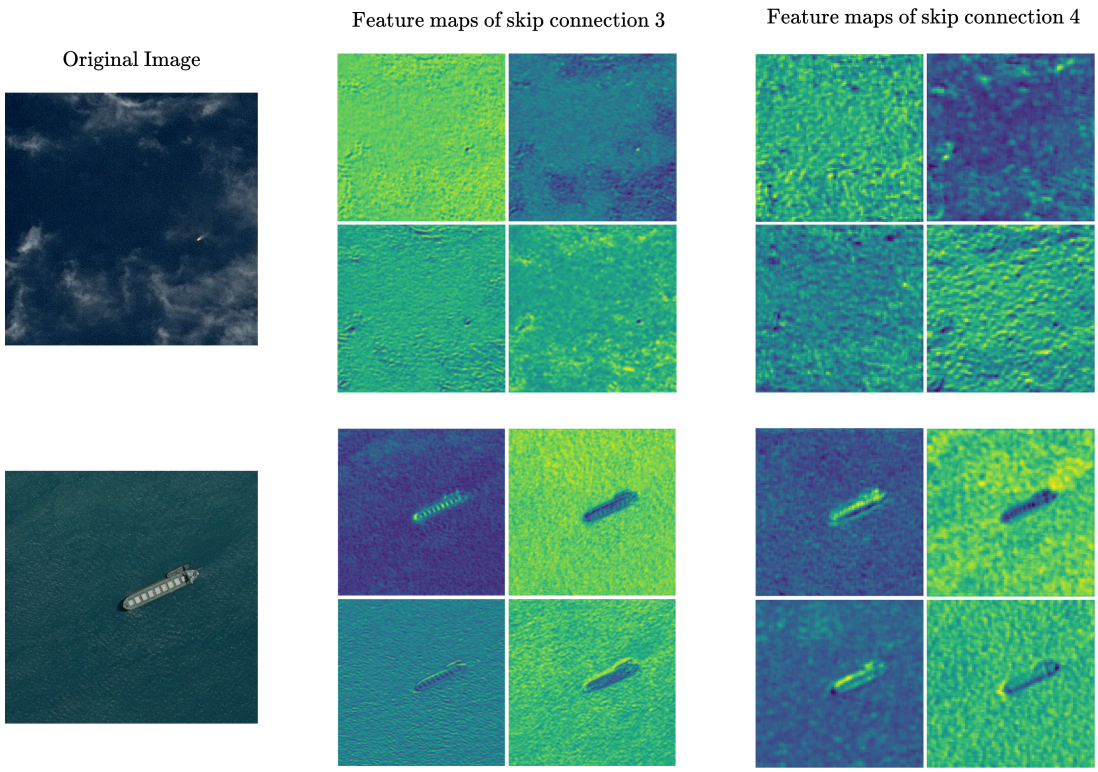



Figure 6.2: Feature maps after skip connections 3 and 4 for a small ship and a large ship.

needed to maintain the model's accuracy.

## 6.3 Thin U-Net Dataflow Architecture

AMD-Xilinx Research Labs are developing an open-source end-to-end design framework to port DNNs on FPGA, from DNN quantization training with Brevitas [Pap23] to model deployment on AMD-Xilinx FPGAs with FINN [Umu+17; Ryb+18; Blo+18]. Although several tools are available for implementing and deploying DNN models on FPGAs, we have chosen these open-source libraries for their good performance characteristics and the active community that regularly improves them and adds new features.

**Brevitas [Pap23]** is a QAT framework built as a drop-in replacement of PyTorch and that is tightly linked to FINN. It provides a set of configurable quantized building blocks for many standard PyTorch layers. This framework supports affine quantization in its various configuration formats and has recently added support for minifloat formats. In

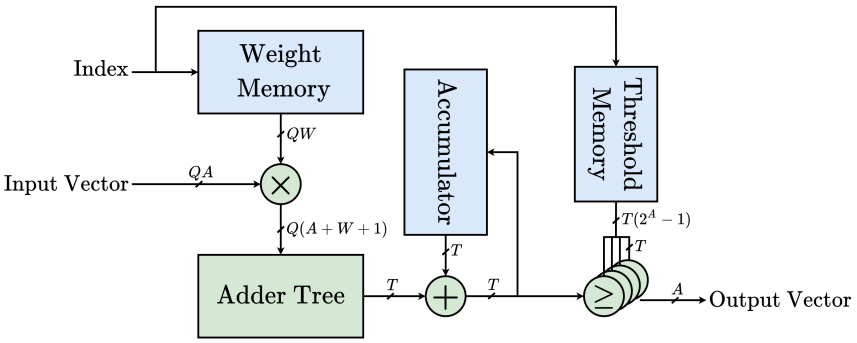



Figure 6.3: The FINN framework uses a single PE for dot product computation. A PE performs $Q$ parallel multiplications between the $W$-bit weights and the $A$-bit activations corresponding to the SIMD value. The weights are stored in on-chip memory to avoid memory bandwidth bottlenecks. It then reduces them in an adder tree for their subsequent accumulation towards the currently computed dot product. Finally, threshold comparisons derive the output values from the accumulation results. This figure is adapted from [Blo+18, Fig. 6].

particular, Brevitas includes a function to export quantized models in the Quantized Open Neural Network Exchange (QONNX) [Pap+22] intermediate representation, the entry format for FINN.

**Quantized Open Neural Network Exchange [Pap+22]** is an extension to the Open Neural Network Exchange (ONNX) standard set of operators to provide a high-level representation of DNN models that can be targeted by Deep Learning frameworks while minimizing reliance on implementation-specific details. This format introduces integer clipping to support sub-8-bit quantization in ONNX and corresponding quantized definitions of operators and data types. Its design and development aim to enable interoperability between different ML frameworks to streamline the path between research and production-ready models.

**FINN** is a framework for deploying dataflow DNN accelerators on AMD-Xilinx FPGA devices. The framework was initially designed to port BNNs [Umu+17] to FPGAs before being extended to larger bit-width integer formats [Blo+18] and Long-Short Term Memory Neural Networks [Ryb+18]. Various transformations are applied to the intermediate representation in QONNX to ensure the closest representation to the final hardware accelerator. The model is synthesized using an HLS library with predefined functions for

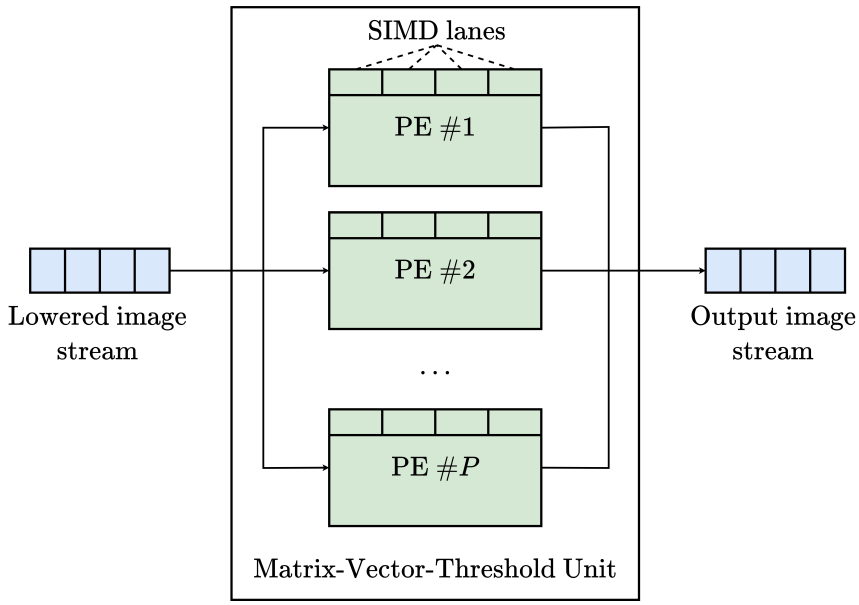



Figure 6.4: Matrix-vector-threshold unit (Matrix-Vector-Threshold Unit (MVU)) used for fully connected and convolutional layers in FINN. The module processes the data stream generated by the Sliding Window Unit (SWU), which reduces a convolution operation to a multiplication between filter and image matrices. The columns of the filter matrix are split between SIMD lanes, and each row is mapped to a different PE. This figure is adapted from [Blo+18, Fig. 7b].

the different nodes in the intermediate representation. FINN exploits the concurrency potential in a given quantized DNN to generate a solution scaled to utilize the committed resources optimally by tuning the previously introduced concurrency parameters. All of them allow to accelerate the computation of the respective layer whose throughput grows proportionally. Finally, the network is deployed on an FPGA with a PYNQ distribution, a Linux-based system, mainly providing a development platform based on Python. The first two published articles [Umu+17; Blo+18] explain in detail how FINN works. We will briefly present the architectural elements of an accelerator generated by FINN for a better understanding.

The basic operation used by dense and convolutional layers is a dot product, which consists of elementwise multiplication and the accumulation of products. The elementary PE used for dot product computation in FINN is shown in Figure 6.3. The processing begins with SIMD products of $A$-bit inputs and $W$-bit weights. The products are

Table 6.2: Comparison of prediction accuracy for the Thin U-Net 32 model and our proposed lightweight alternative on the Airbus Ship dataset in single precision and 5-bit integer affine quantization.

| Architecture | Bit-width | | Accuracy (%) |
|---|---|---|---|
| | Weight | Activation | |
| Thin U-Net 32 | 32 | 32 | 72.8 |
| Thin U-Net 32 Q5 | 5 | 5 | 72.2 |
| Lightweight Thin U-Net 32 | 32 | 32 | 72.0 |
| Lightweight Thin U-Net 32 Q5 | 5 | 5 | 71.4 |

summed in parallel in an adder tree to be sequentially accumulated towards the computed dot product. The final result is produced by comparing the accumulation results with threshold memory values, representing the activation function fused with the BN. The following degree of concurrency is obtained by using $P$ processing elements to form the MVU, illustrated in Figure 6.4, which computes the $P$ output channels in parallel. To lower operations to a matrix-matrix product, an image context of the filter size must be extracted from the layer input. This is carried out by the SWU. It generates the same vectors as those given as inputs, but with interleaved channels to simplify memory accesses and to avoid the need for transposition between layers. This exhibits significantly lower latency than full image buffers and reduces buffer size requirements. Only as many consecutive rows as the height of the convolutional kernel must be kept available.

Table 6.3: Resource usage for both architectures targeting a Zynq UltraScale+ MPSoC ZCU102 board. Input images have a size of $128 \times 128$. The values inside parentheses indicate the usage percentage of available resources on the target board.

| Resources | Thin U-Net 32 Q5 | Lightweight Thin U-Net 32 Q5 |
|---|---|---|
| LUT | 117,319 (42.8%) | 87,655 (32.0%) |
| as logic | 100,219 | 75,980 |
| as memory | 1,996 | 2,076 |
| Flip-Flop (FF) | 159,066 (29.0%) | 114,986 (21.0%) |
| Block RAM (BRAM) | 461.5 (50.6%) | 289.5 (31.7%) |
| $F_{\text{max}}$ (MHz) | 212.9 | 222.2 |

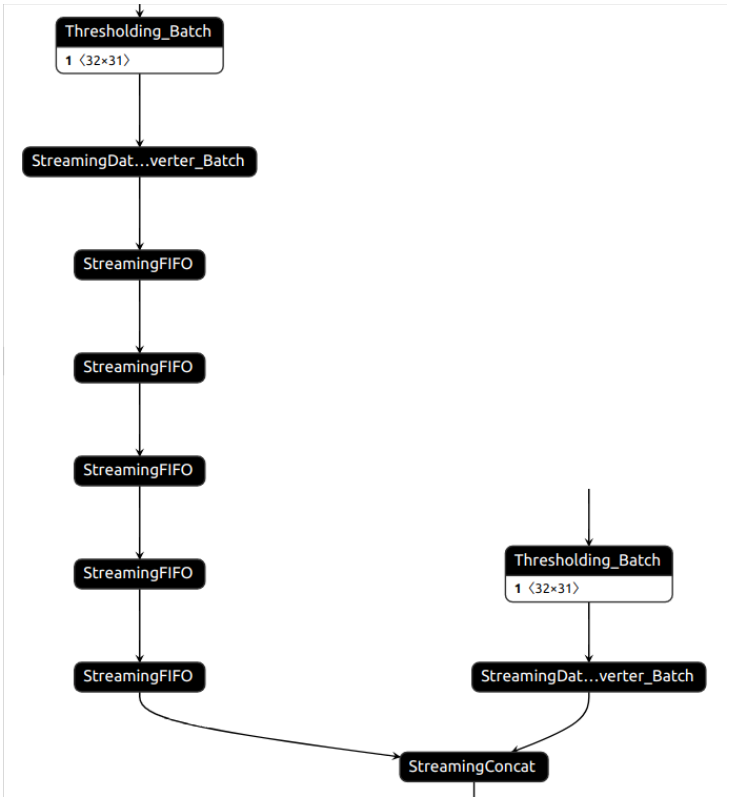



Figure 6.5: Model elements for the second skip connection in Thin U-Net 32 before the Register Transfer Language (RTL) model synthesis. All these elements of a skip connection are saved in Lightweight Thin U-Net 32

### 6.3.1 FPGA Experiments

In our experiments, we use Brevitas 0.9.1 and FINN 0.9. We train the models under the same experimental settings described in the previous section. We use the QAT optimizations available in Brevitas for our integer model using symmetric affine quantization. All the model layers are quantized to the same precision, except for the first and last layers, which are quantized to 8 bits. Table 6.2 presents the results of both the Thin U-Net and the Lightweight Thin U-Net with removed skip connections and their quantized versions. Our experiments show that quantizing the model to 5-bit integers slightly affects accuracy. The overall loss of precision due to the removal of skip connections and quantization is lower than 1.5% compared to the original baseline Thin U-Net and less than 0.8% for the quantized version. Once the model is trained, we use the export function available in Brevitas to generate models in QONNX format.

We target the same development board for both models, the Zynq UltraScale+ ZCU102, which contains a Zynq UltraScale+ MPSoC XCZU9EG chip. We choose not to use DSPs, which are sub-optimal for formats with small bit-widths. The accelerator is designed for $128 \times 128$ input images. Resource usage results are given in Table 6.3. Even though the comparison between the two accelerators is made difficult by the "black box" behavior of FINN and RTL synthesis optimizations, the results still show a trend towards reducing the memory footprint, with 37.3% and 27.7% less BRAMs and FF, respectively, and 4% more LUTs in the proposed lightweight model. This gain is mainly due to the considerable reduction in large FIFOs used to guarantee timings in the dataflow architectures, as illustrated in Figure 6.5 for the second skip connection elements preceding the RTL model synthesis generated by FINN. There is also a reduction of LUTs usage for the logic of 24.2%. Again, the FIFO reduction partly explains this gain. Nevertheless, removing skip connections also suppresses the concatenation process and the quantization operators required to ensure the inputs are on the same grid, which includes a reduction in memory and logic LUTs.

## 6.4 Conclusion

In this chapter, we have investigated the impact of removing skip connections from a U-shape segmentation model on prediction performance for ship detection, together with hardware synthesis results. Our experimental analysis revealed that removing starting skip connections of a U-shaped model has little impact on accuracy ($< 1\%$) while saving 21 Mb of memory for a single-precision model with input images of size $128 \times 128$, i.e. 94% of skip connections memory usage. Quantizing the model to 5 bits reduces the memory footprint significantly while limiting the loss of accuracy by less than 2% compared to the original single-precision model. Using the open-source FINN framework, we also ported the model on an FPGA. Our results show a clear gain in memory consumption, with 37.3% and 27.7% less BRAMs and FFs respectively.

Although this model works well for the Airbus ship dataset, it is probably unsuitable for lower-resolution images, as the feature maps would be more quickly affected by the loss of information, and the impacted skip connection stages would be earlier in the network. The opposite direction is also true, i.e., with more detail, it would be possible

to remove more skip connections. This is all the more true given that embedded image sensors on recent satellites have increased resolution. The authors of [Wan+22] suggest integrating skip connections in the design space exploration by justifying a different impact depending on the dataset used. Experiments on other ship detection datasets [Liu+17b; GPG18; Che+20; Zha+21b] must be carried out to determine and confirm the appropriate configuration of this lightweight model. Nevertheless, the memory footprint saved by removing the first skip connections is significant, with minimal accuracy degradation.

# Conclusion

Earth Observation (EO) provides an effective way of exploring the physical, chemical, and biological information related to the Earth. This information collected by EO satellites is widely used in various research fields, especially in relation to the environment, where the measurements made by EO satellites are indispensable. Moreover, these new space applications related to EO produce a huge volume of data extracted from various image and radar sensors. Transmitting all this data is possible through communication between satellites and ground stations. However, EO systems are limited by these downlink communications due to hardware and on-board power constraints or ground-station operation costs, for example. Thus, there is a need to reduce the amount of data transmitted through the downlink. While data compression is widely used for size reduction, transmitting only relevant data through on-board processing has only recently started gaining interest.

AI, and in particular DL, is starting to be successfully applied in space applications. However, the inference computation of many models is still mainly performed on ground platforms due to their memory footprint and computational intensity. This document presents our research work on CNNs compression using quantization methods. In particular, we focused on the problem of ship detection on satellite images and the feasibility of deploying semantic segmentation models. To address this problem, we proposed a mixed precision configuration exploration method. We also studied minifloats as a candidate arithmetic format for inference. Finally, we investigated the architecture of semantic segmentation U-shaped models and proposed a lightweight version.

Finding appropriate bit-widths can be time-consuming when dealing with a new dataset or model. In this context, we set out to learn bit-widths for the Airbus Ship dataset. This dataset is very large, so training the model takes a significant amount of time. When we reviewed existing methods, we found that most of them required many epochs to obtain a suitable mixed precision configuration. Chapter 4 introduces Adaptive Quantization Aware Training (AdaQAT), a new optimization-based method for mixed-precision quantization uniform quantization of both weights and activations. Our gradient-based approach uses relaxed fractional bit-widths updated using a gradient de-

scent rule. Compared to previous approaches that are generally intended to be used in a fine-tuning setting, in early tests, our method seems to be more flexible, being capable of operating in both fine-tuning and training from scratch scenarios, producing results that are on par with state-of-the-art mixed-precision quantization approaches on CIFAR10 with a ResNet20 network. It also performs well in mixed-precision fine-tuning of ResNet18 and MobileNet-V2 on ImageNet.

Floating-point formats are widely used during the training stage of deep neural networks. However, very little research has been conducted into their use for low-precision inference acceleration. Chapter 5 investigates using minifloat formats for DNN inference. We analyze the features and characteristics of minifloat formats, such as zero encoding, subnormal support, and scaling. This analysis has lead us to introduce a minifloat encoding scheme optimized for inference and its corresponding quantization procedure. We have proposed a QAT algorithm for learning compressed low-precision floating-point DNN models. In addition, we learn the exponent biases of each layer for both weights and activations. Our experiments on the Airbus Ship datasets show promising results, with low-precision floating-point models being competitive with single-precision baselines and affine quantization-based alternatives. To show the potential impact of using minifloat data formats, we have also suggested an implementation of a minifloat-enabled multiplier that can be used as a basis for a full DNN inference accelerator.

The U-shaped model forms the backbone for many semantic segmentation models. To deploy this type of model using an FPGA-based dataflow accelerator, the impact of long skip connections must be considered. Chapter 6 investigates the impact of removing skip connections from a U-shape segmentation model on prediction performance for ship detection, together with hardware synthesis results. Long skip connections induce a significant memory cost, especially the starting ones (they correspond to the largest intermediate features.) Skip connections are also a major barrier to the deployment of dataflow accelerators due to their hardware implications. Our experimental analysis on the Airbus Ship dataset revealed that removing starting skip connections of a U-shaped model has little impact on accuracy ($< 1\%$) while saving 94% of skip connections memory usage. Quantizing the model to 5 bits further reduces the memory footprint significantly while limiting the loss of accuracy by less than 2% compared to the original single-precision model. Using the open-source FINN framework, we also ported the model on an FPGA. Our results show a clear gain in memory consumption, with 37.3% and 27.7% less BRAMs and FFs respectively, at the cost of a 4% increase in LUT memory usage.

# Research Perspectives

DNN quantization is a very active research topic in both academia and industry. The growing interest in deploying DNNs on embedded systems is leading research towards new compression techniques.

## Expand the granularity of the AdaQAT exploration space

In the same way that it is possible to adjust the granularity of CNN quantization parameters, such as the scaling factor (Section 2.3.2), it is also possible to adapt the granularity of another parameter: the bit-width. AdaQAT offers the possibility of learning the bit-width at a granularity down to layer-wise configuration. With the quantization of convolution layers that can be adapted to the individual kernel level, we consider extending AdaQAT to finer levels of mixed-precision quantization granularity, such as channel-wise. However, the search space complexity increases exponentially with the number of kernels, so the time required to learn precision with our method would be extremely long as we evaluate each bit-width separately. To overcome this problem, we suggest experiments involving the simultaneous learning of multiple bit-widths. Initial experiments show the potential for accelerating convergence when designing a channel-wise quantization extension.

## AdaQAT hyperparameter setup

Although AdaQAT is relatively powerful in finding a competitive mixed precision configuration, properly setting the hyper-parameters $\lambda_{\mathbf{w}}$ and $\lambda_{\mathbf{a}}$ can be challenging. This is all the more difficult as mixed precision results are widely affected by variations in the loss function. By modifying these hyper-parameters, we can see significant changes in the final mixed precision results. This statement is not only valid for the batch size but also applies to any change in other parameters that may affect the loss function, such as the number of epochs, the learning rate scheduler, etc. This issue is due to the finite difference approximation used to calculate the gradient of the bit-widths. A future contribution would be to work on a method for avoiding this problem, such as using a reference value for a given batch size [Goy17].

### Modelling a more realistic hardware cost function

In AdaQAT, we have chosen to normalize the hardware cost function to free ourselves from the variable component related to the model's architecture. In the future, we intend to explore finer hardware complexity and energy consumption metrics tailored for a specific target architecture (*e.g.* FPGAs) in the $\mathcal{L}_{\text{Hard}}$ term. This is essential to assess changes in the behavior of our method when it is subjected to finer variations in the material cost function.

### Extend AdaQAT to other quantization formats

The exploration of mixed precision for uniform integer formats is widely studied in the literature. With the recent interest in new formats, particularly minifloats, as discussed in Chapter 5, it would be interesting to extend the exploration to this type of data format. Finding a suitable configuration for floating-point formats is challenging due to a larger search space with two bit-widths to optimize: the mantissa and the exponent.

### Design and deploy a full minifloat model on FPGA

Our initial results on minifloat quantization suggest that this format might be a potential candidate for inference acceleration. Future work will involve designing an accelerator for the Thin U-Net 32 model based on minifloats to better assess the feasibility of using low-precision floating-point data in an on-board spatial processing context.

### Explore Vision Transformer models

Vision Transformers, derived from the well-known Transformer models used in language processing, are becoming increasingly popular for image processing. The architecture of vision transformers differs from CNNs in that they do not include a convolutional layer, which is central to CNNs. Vision Transformers break down an input image into a series of patches, serialize each patch into a vector, and map it to a smaller dimension with a single matrix multiplication. These vector embeddings are then processed by a Transformer encoder as token embeddings. The introduction of Vision Transformers offers new opportunities and challenges for the compression and integration of this new type of model in accelerators. Experimenting with AdaQAT on Vision Transformer models would enable the method to be evaluated on other architectures that are very different

from those already studied. In addition, minifloats are widely studied in the literature for Transformer quantization, making our minifloat format a potential candidate.

### Alternative ship detection datasets

Image segmentation is essential to many space applications, such as cloud detection or area classification. Ship detection is another highly studied segmentation task due to its complexity, both for detecting small ships and ships in the harbor. This makes it a good starting task for evaluating methods in challenging conditions. The resolution of the input data is particularly important for the successful detection of small ships. Although our lightweight model works well for the Airbus ship dataset, the resolution influences the model's behavior. Further evaluation of our U-shaped model with suppressed skip connections on other ship detection datasets would strengthen its usefulness and applicability.

## List of Publications

This thesis has led to two publications in international conferences. These publications are as follows:

- Cédric Gernigon, Silviu-Ioan Filip, Olivier Sentieys, Clément Coggiola, and Mickaël Bruno, "Low-Precision Floating-Point for Efficient On-Board Deep Neural Network Processing", *in*: *2023 European Data Handling and Data Processing Conference (EDHPC)*, 2023, pp. 1–8

- Cédric Gernigon, Silviu-Ioan Filip, Olivier Sentieys, Clément Coggiola, and Mickaël Bruno, "AdaQAT: Adaptive Bit-Width Quantization-Aware Training", *in*: *2024 IEEE 6th International Conference on AI Circuits and Systems (AICAS)*, 2024, pp. 442–446

Parts of this research have also been presented on various occasions:

- 1$^{st}$ mixed-precision computing workshop in Paris (2022)

- 13$^{th}$ Rencontres Arithmétiques du GdR Informatique Mathématique (RAIM) in Nantes (2022)

- 22$^{\text{th}}$ Journées Jeunes Chercheurs CNES (JC2) in Toulouse (2022) with a poster
- Intelligence Artificielle Embarquée et Connectée (IA-EC) in Rennes (2023)
- 5$^{\text{th}}$ FPTalks workshop (2024)

# Bibliography


[AAA21] Abdolmaged Alkhulaifi, Fahad Alsahli, and Irfan Ahmad, "Knowledge distillation in deep learning and its applications", *in*: *PeerJ Computer Science* 7 (2021), e474.

[Aba+16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al., "Tensorflow: Large-scale machine learning on heterogeneous distributed systems", *in*: *arXiv preprint arXiv:1603.04467* (2016).

[Abd+17] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Serot, Cedric Bourrasset, and François Berry, "Tactics to directly map CNN graphs on embedded FPGAs", *in*: *IEEE Embedded Systems Letters* 9.*4* (2017), pp. 113–116.

[Ach+23] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al., "Gpt-4 technical report", *in*: *arXiv preprint arXiv:2303.08774* (2023).

[Agg+23] Shivam Aggarwal, Alessandro Pappalardo, Hans Jakob Damsgaard, Giuseppe Franco, Thomas B Preußer, Michaela Blott, and Tulika Mitra, "Post-training quantization with low-precision minifloats and integers on FPGAs", *in*: *arXiv preprint arXiv:2311.12359* (2023).

[Ahn+19] Sungsoo Ahn, Shell Xu Hu, Andreas Damianou, Neil D Lawrence, and Zhenwen Dai, "Variational information distillation for knowledge transfer", *in*: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 9163–9171.

[AHS85] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski, "A learning algorithm for Boltzmann machines", *in*: *Cognitive science* 9.*1* (1985), pp. 147–169.

[AIZ19] Babajide O Ayinde, Tamer Inanc, and Jacek M Zurada, "Redundant feature pruning for accelerated inference in deep neural networks", *in*: *Neural Networks* 118 (2019), pp. 148–158.

[AL20a] Roya Afshar and Shuai Lu, "Classification and recognition of space debris and its pose estimation based on deep learning of CNNs", *in*: *HCI International 2020-Posters: 22nd International Conference, HCII 2020, Copenhagen, Denmark, July 19–24, 2020, Proceedings, Part I 22*, Springer, 2020, pp. 605–613.

[AL20b] Zeyuan Allen-Zhu and Yuanzhi Li, "Towards understanding ensemble, knowledge distillation and self-distillation in deep learning", *in*: *arXiv preprint arXiv:2012.09816* (2020).

[Ale+17] Hande Alemdar, Vincent Leroy, Adrien Prost-Boucle, and Frédéric Pétrot, "Ternary neural networks for resource-efficient AI applications", *in*: *2017 international joint conference on neural networks (IJCNN)*, IEEE, 2017, pp. 2547–2554.

[ALL16] Nicolas Audebert, Bertrand Le Saux, and Sébastien Lefèvre, "Semantic segmentation of earth observation data using multimodal and multi-scale deep networks", *in*: *Asian conference on computer vision*, Springer, 2016, pp. 180–196.

[Alm+21] Gabriel Henrique de Almeida Pereira, Andre Minoro Fusioka, Bogdan Tomoyuki Nassu, and Rodrigo Minetto, "Active fire detection in Landsat-8 imagery: A large-scale dataset and a deep-learning study", *in*: *ISPRS Journal of Photogrammetry and Remote Sensing* 178 (2021), pp. 171–186.

[Alo+18a] Md Zahangir Alom, Mahmudul Hasan, Chris Yakopcic, Tarek M Taha, and Vijayan K Asari, "Recurrent residual convolutional neural network based on u-net (r2u-net) for medical image segmentation", *in*: *arXiv preprint arXiv:1802.06955* (2018).

[Alo+18b] Md Zahangir Alom, Tarek M Taha, Christopher Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Brian C Van Esesn, Abdul A S Awwal, and Vijayan K Asari, "The history began from alexnet: A comprehensive survey on deep learning approaches", *in*: *arXiv preprint arXiv:1803.01164* (2018).

[And+16] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini, "YodaNN: An ultra-low power convolutional neural network accelerator based on binary weights", *in*: *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, IEEE, 2016, pp. 236–241.

[Ans+22] Mohammed Yusuf Ansari, Yin Yang, Shidin Balakrishnan, Julien Abinahed, Abdulla Al-Ansari, Mohamed Warfa, Omran Almokdad, Ali Barah, Ahmed Omer, Ajay Vikram Singh, et al., "A lightweight neural network with multi-scale feature enhancement for liver CT segmentation", *in*: *Scientific reports* 12.*1* (2022), p. 14153.

[Ass+21] Imad Al Assir, Mohamad El Iskandarani, Hadi Rayan Al Sandid, and Mazen AR Saghir, "Arrow: A RISC-V vector accelerator for machine learning inference", *in*: *arXiv preprint arXiv:2107.07169* (2021).

[Bah+19] Gaétan Bahl, Lionel Daniel, Matthieu Moretti, and Florent Lafarge, "Low-power neural networks for semantic segmentation of satellite images", *in*: *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, IEEE, 2019.

[Bai73] Alexander Bain, *Mind and body: The theories of their relation*, vol. 4, D. Appleton, 1873.

[BB23] Christopher M Bishop and Hugh Bishop, *Deep learning: Foundations and concepts*, Springer Nature, 2023.

[BCN06] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil, "Model compression", *in*: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006, pp. 535–541.

[BFS24] Sami Ben Ali, Silviu-Ioan Filip, and Olivier Sentieys, "A Stochastic Rounding-Enabled Low-Precision Floating-Point MAC for DNN Training", *in*: *27th IEEE/ACM Design, Automation and Test in Europe (DATE)*, Valencia, Spain, 2024, pp. 1–6.

[Bha+20] Yash Bhalgat, Jinwon Lee, Markus Nagel, Tijmen Blankevoort, and Nojun Kwak, "LSQ+: Improving Low-Bit Quantization Through Learnable Offsets and Better Initialization", *in*: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2020, pp. 696–697.

[BJ20] Nazanin Beheshti and Lennart Johnsson, "Squeeze u-net: A memory and energy efficient image segmentation network", *in*: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, 2020, pp. 364–365.

[BLC13] Yoshua Bengio, Nicholas Léonard, and Aaron Courville, *Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation*, Aug. 2013.

[Blo+18] Michaela Blott, Thomas B Preußer, Nicholas J Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers, "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks", *in*: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11.*3* (2018), pp. 1–23.

[BNS19] Ron Banner, Yury Nahshan, and Daniel Soudry, "Post training 4-bit quantization of convolutional networks for rapid-deployment", *in*: *Advances in Neural Information Processing Systems* 32 (2019).

[Boo+21] Yoonho Boo, Sungho Shin, Jungwook Choi, and Wonyong Sung, "Stochastic precision ensemble: self-knowledge distillation for quantized deep neural networks", *in*: *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, 8, 2021, pp. 6794–6802.

[BT19] Adrian Bulat and Georgios Tzimiropoulos, "Xnor-net++: Improved binary neural networks", *in*: *arXiv preprint arXiv:1909.13863* (2019).

[CA20] Stefano Cherubin and Giovanni Agosta, "Tools for reduced precision computation: a survey", *in*: *ACM Computing Surveys (CSUR)* 53.*2* (2020), pp. 1–35.

[Cai+19] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han, "Once-for-all: Train one network and specialize it for efficient deployment", *in*: *arXiv preprint arXiv:1908.09791* (2019).

[Cai+20] Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W Mahoney, and Kurt Keutzer, "Zeroq: A novel zero shot quantization framework", *in*: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 13169–13178.

[Cav+15] Lukas Cavigelli, David Gschwend, Christoph Mayer, Samuel Willi, Beat Muheim, and Luca Benini, "Origami: A convolutional network accelerator", *in*: *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, 2015, pp. 199–204.

[CBD14] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David, "Training deep neural networks with low precision multiplications", *in*: *arXiv preprint arXiv:1412.7024* (2014).

[CBD15] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David, "BinaryConnect: Training Deep Neural Networks with binary weights during propagations", *in*: *Advances in Neural Information Processing Systems 28 – NIPS 2015*, vol. 2, NIPS'15, MIT Press, 2015, pp. 3123–3131.

[CDP22] Maxime Christ, Florent De Dinechin, and Frédéric Pétrot, "Low-precision logarithmic arithmetic for neural network accelerators", *in*: *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, IEEE, 2022, pp. 72–79.

[CES16] Yu-Hsin Chen, Joel Emer, and Vivienne Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks", *in*: *ACM SIGARCH computer architecture news* 44.*3* (2016), pp. 367–379.

[Cha+20] Sung-En Chang, Yanyu Li, Mengshu Sun, Weiwen Jiang, Runbin Shi, Xue Lin, and Yanzhi Wang, "MSP: an FPGA-specific mixed-scheme, multi-precision deep neural network quantization framework", *in*: *arXiv preprint arXiv:2009.07460* (2020).

[Che+14] 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*: *ACM SIGARCH Computer Architecture News* 42.*1* (2014), pp. 269–284.

[Che+15] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen, "Compressing neural networks with the hashing trick", *in*: *International conference on machine learning*, PMLR, 2015, pp. 2285–2294.

[Che+18a] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation", *in*: *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 801–818.

[Che+18b] Yukang Chen, Gaofeng Meng, Qian Zhang, Xinbang Zhang, Liangchen Song, Shiming Xiang, and Chunhong Pan, "Joint neural architecture search and quantization", *in*: *arXiv preprint arXiv:1811.09426* (2018).

[Che+19] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices", *in*: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.*2* (2019), pp. 292–308.

[Che+20] Kaiyan Chen, Ming Wu, Jiaming Liu, and Chuang Zhang, "FGSD: A dataset for fine-grained ship detection in high resolution satellite images", *in*: *arXiv preprint arXiv:2003.06832* (2020).

[Chi+20] Po-Sheng Chiu, Jia-Wei Chang, Ming-Che Lee, Ching-Hui Chen, and Da-Sheng Lee, "Enabling intelligent environment by the design of emotionally aware virtual assistant: A case of smart campus", *in*: *IEEE Access* 8 (2020), pp. 62032–62041.

[Cho+10] Young-Kyu Choi, Kisun You, Jungwook Choi, and Wonyong Sung, "A real-time FPGA-based 20 000-word speech recognizer with optimized DRAM access", *in*: *IEEE Transactions on Circuits and Systems I: Regular Papers* 57.*8* (2010), pp. 2119–2131.

[Cho+18] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan, "PACT: Parameterized Clipping Activation for Quantized Neural Networks", *in*: *arXiv preprint arXiv:1805.06085* (2018).

[Cho+19] Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev, "Low-bit quantization of neural networks for efficient inference", *in*: *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, IEEE, 2019, pp. 3009–3018.

[Cho17] François Chollet, "Xception: Deep learning with depthwise separable convolutions", *in*: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1251–1258.

[Cou+16] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio, "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or −1", *in*: *arXiv preprint arXiv:1602.02830* (2016).

[CTN18] Jingyong Cai, Masashi Takemoto, and Hironori Nakajo, "A deep look into logarithmic quantization of model parameters in neural networks", *in*: *Proceedings of the 10th International Conference on Advances in Information Technology*, 2018, pp. 1–8.

[CV20] Zhaowei Cai and Nuno Vasconcelos, "Rethinking differentiable search for mixed-precision neural networks", *in*: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 2349–2358.

[CWC21] Weihan Chen, Peisong Wang, and Jian Cheng, "Towards mixed-precision quantization of neural networks via constrained optimization", *in*: *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 5350–5359.

[CX14] Jason Cong and Bingjun Xiao, "Minimizing computation in convolutional neural networks", *in*: *International conference on artificial neural networks*, Springer, 2014, pp. 281–290.

[CZH18] Han Cai, Ligeng Zhu, and Song Han, "Proxylessnas: Direct neural architecture search on target task and hardware", *in*: *arXiv preprint arXiv:1812.00332* (2018).

[Dar+18] Sajad Darabi, Mouloud Belbahri, Matthieu Courbariaux, and Vahid Partovi Nia, "Bnn+: Improved binary network training", *in*: (2018).

[De +08] Florent De Dinechin, Bogdan Pasca, Octavian Cret, and Radu Tudoran, "An FPGA-specific approach to floating-point accumulation and sum-of-products", *in*: *2008 International Conference on Field-Programmable Technology*, IEEE, 2008, pp. 33–40.

[Dea+12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al., "Large scale distributed deep networks", *in*: *Advances in neural information processing systems* 25 (2012).

[Den+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei, "Imagenet: A large-scale hierarchical image database", *in*: *2009 IEEE conference on computer vision and pattern recognition*, Ieee, 2009, pp. 248–255.

[Den+13] Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando De Freitas, "Predicting parameters in deep learning", *in*: *Advances in neural information processing systems* 26 (2013).

[Den12] Li Deng, "The mnist database of handwritten digit images for machine learning research", *in*: *IEEE Signal Processing Magazine* 29.*6* (2012), pp. 141–142.

[DF12] Charles Dubout and François Fleuret, "Exact acceleration of linear object detectors", *in*: *Computer Vision–ECCV 2012: 12th European Conference on Computer Vision, Florence, Italy, October 7-13, 2012, Proceedings, Part III 12*, Springer, 2012, pp. 301–311.

[Din+19] Ruizhou Ding, Ting-Wu Chin, Zeye Liu, and Diana Marculescu, "Regularizing activation distribution for training binarized deep networks", *in*: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 11408–11417.

[DM74] Jack B Dennis and David P Misunas, "A preliminary architecture for a basic data-flow processor", *in*: *Proceedings of the 2nd annual symposium on Computer architecture*, 1974, pp. 126–132.

[Don+19] Zhen Dong, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer, "HAWQ: Hessian Aware Quantization of Neural Networks with Mixed-Precision", *in*: *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 293–302.

[Don+20] Zhen Dong, Zhewei Yao, Daiyaan Arfeen, Amir Gholami, Michael W Mahoney, and Kurt Keutzer, "Hawq-v2: Hessian aware trace-weighted quantization of neural networks", *in*: *Advances in neural information processing systems* 33 (2020), pp. 18518–18529.

[Dro+16] Michal Drozdzal, Eugene Vorontsov, Gabriel Chartrand, Samuel Kadoury, and Chris Pal, "The importance of skip connections in biomedical image segmentation", *in*: *International workshop on deep learning in medical image analysis, international workshop on large-scale annotation of biomedical data and expert label synthesis*, Springer, 2016, pp. 179–187.

[Dup+20] Etienne Dupuis, David Novo, Ian O'Connor, and Alberto Bosio, "On the automatic exploration of weight sharing for deep neural network compression", *in*: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2020, pp. 1319–1322.

[Dup+22] Etienne Dupuis, Silviu-Ioan Filip, Olivier Sentieys, David Novo, Ian O'Connor, and Alberto Bosio, "Approximate Computing Techniques - From Component- to Application-Level", *in*: Springer, 2022, chap. Approximations in Deep Learning.

[DV16] Vincent Dumoulin and Francesco Visin, "A guide to convolution arithmetic for deep learning", *in*: *arXiv preprint arXiv:1603.07285* (2016).

[EAA19] Taha Emara, Hossam E Abd El Munim, and Hazem M Abbas, "Liteseg: A novel lightweight convnet for semantic segmentation", *in*: *2019 digital image computing: Techniques and applications (DICTA)*, IEEE, 2019, pp. 1–7.

[Elt+19] Ahmed Elthakeb, Prannoy Pilligundla, FatemehSadat Mireshghallah, Amir Yazdanbakhsh, Sicuan Gao, and Hadi Esmaeilzadeh, "Releq: an automatic reinforcement learning approach for deep quantization of neural networks", *in*: *NeurIPS ML for Systems workshop, 2018*, 2019.

[ES16] Ronen Eldan and Ohad Shamir, "The power of depth for feedforward neural networks", *in*: *Conference on learning theory*, PMLR, 2016, pp. 907–940.

[Ess+19] Steven K Esser, Jeffrey L McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S Modha, "Learned Step Size Quantization", *in*: *arXiv preprint arXiv:1902.08153* (2019).

[FAG19] Alexander Finkelstein, Uri Almog, and Mark Grobman, "Fighting quantization bias with bias", *in*: *arXiv preprint arXiv:1906.03193* (2019).

[Fan+19] Hongxiang Fan, Gang Wang, Martin Ferianc, Xinyu Niu, and Wayne Luk, "Static block floating-point quantization for convolutional neural networks on FPGA", *in*: *2019 International Conference on Field-Programmable Technology (ICFPT)*, IEEE, 2019, pp. 28–35.

[Far+09] Clément Farabet, Cyril Poulet, Jefferson Y Han, and Yann LeCun, "Cnp: An fpga-based processor for convolutional networks", *in*: *2009 International Conference on Field Programmable Logic and Applications*, IEEE, 2009, pp. 32–37.

[Fas+21] Nael Fasfous, Manoj Rohit Vemparala, Alexander Frickenstein, Emanuele Valpreda, Driton Salihu, Nguyen Anh Vu Doan, Christian Unger, Naveen Shankar Nagaraja, Maurizio Martina, and Walter Stechele, "Hw-flowq: A multi-abstraction level hw-cnn co-design quantization methodology", *in*: *ACM Transactions on Embedded Computing Systems (TECS)* 20.*5s* (2021), pp. 1–25.

[FC18] Jonathan Frankle and Michael Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks", *in*: *arXiv preprint arXiv:1803.03635* (2018).

[Fuk80] Kunihiko Fukushima, "A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position", *in*: *Biol, Cybern* 36 (1980), pp. 193–202.

[Fur+20] Gianluca Furano, Gabriele Meoni, Aubrey Dunne, David Moloney, Veronique Ferlet-Cavrois, Antonis Tavoularis, Jonathan Byrne, Léonie Buckley, Mihalis Psarakis, Kay-Obbe Voss, and Luca Fanucci, "Towards the Use of Artificial Intelligence on the Edge in Space Systems: Challenges and Opportunities", *in*: *IEEE Aerospace and Electronic Systems Magazine* 35.*12* (2020), pp. 44–56.

[GB10] Xavier Glorot and Yoshua Bengio, "Understanding the difficulty of training deep feedforward neural networks", *in*: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.

[Ger+23] Cédric Gernigon, Silviu-Ioan Filip, Olivier Sentieys, Clément Coggiola, and Mickaël Bruno, "Low-Precision Floating-Point for Efficient On-Board Deep Neural Network Processing", *in*: *2023 European Data Handling and Data Processing Conference (EDHPC)*, 2023, pp. 1–8.

[Ger+24] Cédric Gernigon, Silviu-Ioan Filip, Olivier Sentieys, Clément Coggiola, and Mickaël Bruno, "AdaQAT: Adaptive Bit-Width Quantization-Aware Training", *in*: *2024 IEEE 6th International Conference on AI Circuits and Systems (AICAS)*, 2024, pp. 442–446.

[Gho+22] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer, "A survey of quantization methods for efficient neural network inference", *in*: *Low-Power Computer Vision*, Chapman and Hall/CRC, 2022, pp. 291–326.

[Gir+14] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation", *in*: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[Giu+21] Gianluca Giuffrida, Luca Fanucci, Gabriele Meoni, Matej Batič, Léonie Buckley, Aubrey Dunne, Chris Van Dijk, Marco Esposito, John Hefele, Nathan Vercruyssen, et al., "The Φ-Sat-1 mission: The first on-board deep neural network demonstrator for satellite earth observation", *in*: *IEEE Transactions on Geoscience and Remote Sensing* 60 (2021), pp. 1–14.

[Gon+14] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev, "Compressing deep convolutional networks using vector quantization", *in*: *arXiv preprint arXiv:1412.6115* (2014).

[Gon+19a] Chengyue Gong, Zixuan Jiang, Dilin Wang, Yibo Lin, Qiang Liu, and David Z Pan, "Mixed precision neural architecture search for energy efficient deep learning", *in*: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, 2019, pp. 1–7.

[Gon+19b] Ruihao Gong, Xianglong Liu, Shenghu Jiang, Tianxiang Li, Peng Hu, Jiazhen Lin, Fengwei Yu, and Junjie Yan, "Differentiable soft quantization: Bridging full-precision and low-bit neural networks", *in*: *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 4852–4861.

[Gou+21] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao, "Knowledge distillation: A survey", *in*: *International Journal of Computer Vision* 129 (2021), pp. 1789–1819.

[Goy17] P Goyal, "Accurate, large minibatch SG D: training imagenet in 1 hour", *in*: *arXiv preprint arXiv:1706.02677* (2017).

[GPG18] Antonio-Javier Gallego, Antonio Pertusa, and Pablo Gil, "Automatic ship classification from optical aerial images with convolutional neural networks", *in*: *Remote Sensing* 10.*4* (2018), p. 511.

[Gre+15] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Rezende, and Daan Wierstra, "Draw: A recurrent neural network for image generation", *in*: *International conference on machine learning*, PMLR, 2015, pp. 1462–1471.

[Gup+15] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan, "Deep learning with limited numerical precision", *in*: *International conference on machine learning*, PMLR, 2015, pp. 1737–1746.

[Hag+19] Andrei Hagiescu, Martin Langhammer, Bogdan Pasca, Philip Colangelo, Jason Thong, and Niayesh Ilkhani, "Bfloat MLP training accelerator for FPGAs", *in*: *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, IEEE, 2019, pp. 1–5.

[Ham90] Dan Hammerstrom, "A VLSI architecture for high-performance, low-cost, on-chip learning", *in*: *1990 IJCNN International Joint Conference on Neural Networks*, IEEE, 1990, pp. 537–544.

[Han+15] Song Han, Jeff Pool, John Tran, and William Dally, "Learning both weights and connections for efficient neural network", *in*: *Advances in neural information processing systems* 28 (2015).

[Har+20] Matan Haroush, Itay Hubara, Elad Hoffer, and Daniel Soudry, "The knowledge within: Methods for data-free model compression", *in*: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 8494–8502.

[HCH21] Cheng-Wei Huang, Tim-Wei Chen, and Juinn-Dar Huang, "All-You-Can-Fit 8-Bit Flexible Floating-Point Format for Accurate and Memory-Efficient Inference of Deep Neural Networks", *in*: *arXiv preprint arXiv:2104.07329* (2021).

[He+15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification", *in*: *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[He+16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, "Deep residual learning for image recognition", *in*: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[Heo+19] Byeongho Heo, Minsik Lee, Sangdoo Yun, and Jin Young Choi, "Knowledge transfer via distillation of activation boundaries formed by hidden neurons", *in*: *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, 01, 2019, pp. 3779–3787.

[HF91] Markus Hoehfeld and Scott E Fahlman, *Learning with limited numerical precision using the cascade-correlation algorithm*, Citeseer, 1991.

[HF92] Markus Höhfeld and Scott E Fahlman, "Probabilistic rounding in neural network learning with limited precision", *in*: *Neurocomputing* 4.*6* (1992), pp. 291–299.

[HG16] Dan Hendrycks and Kevin Gimpel, "Gaussian error linear units (gelus)", *in*: *arXiv preprint arXiv:1606.08415* (2016).

[HH93] J.L. Holt and J.-N. Hwang, "Finite precision error analysis of neural network hardware implementations", *in*: *IEEE Transactions on Computers* 42.*3* (Mar. 1993), Conference Name: IEEE Transactions on Computers, pp. 281–290.

[HJN20] Hai Victor Habi, Roy H Jennings, and Arnon Netzer, "Hmq: Hardware friendly mixed precision quantization block for cnns", *in*: *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXVI 16*, Springer, 2020, pp. 448–463.

[HK70] Arthur E Hoerl and Robert W Kennard, "Ridge regression: Biased estimation for nonorthogonal problems", *in*: *Technometrics* 12.*1* (1970), pp. 55–67.

[HMD15] Song Han, Huizi Mao, and William J Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding", *in*: *arXiv preprint arXiv:1510.00149* (2015).

[Hor14] Mark Horowitz, "1.1 computing's energy problem (and what we can do about it)", *in*: *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)*, IEEE, 2014, pp. 10–14.

[How+17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications", *in*: *arXiv preprint arXiv:1704.04861* (2017).

[How+19] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al., "Searching for mobilenetv3", *in*: *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 1314–1324.

[Hua+17] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger, "Densely connected convolutional networks", *in*: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[Hua+20] Huimin Huang, Lanfen Lin, Ruofeng Tong, Hongjie Hu, Qiaowei Zhang, Yutaro Iwamoto, Xianhua Han, Yen-Wei Chen, and Jian Wu, "Unet 3+: A full-scale connected unet for medical image segmentation", *in*: *ICASSP 2020-2020 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, IEEE, 2020, pp. 1055–1059.

[Hua+22] Xijie Huang, Zhiqiang Shen, Shichao Li, Zechun Liu, Hu Xianghong, Jeffry Wicaksana, Eric Xing, and Kwang-Ting Cheng, "SDQ: Stochastic Differentiable Quantization with Mixed Precision", *in*: *International Conference on Machine Learning*, PMLR, 2022, pp. 9295–9309.

[Hub+16] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio, "Binarized neural networks", *in*: *Advances in neural information processing systems* 29 (2016).

[Hub+17] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio, "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations", *in*: *The Journal of Machine Learning Research* 18.*1* (2017), pp. 6869–6898.

[Hub+20] Itay Hubara, Yury Nahshan, Yair Hanani, Ron Banner, and Daniel Soudry, "Improving post training neural quantization: Layer-wise calibration and integer programming", *in*: *arXiv preprint arXiv:2006.10518* (2020).

[HVD15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean, "Distilling the knowledge in a neural network", *in*: *arXiv preprint arXiv:1503.02531* (2015).

[HW17] Zehao Huang and Naiyan Wang, "Like what you like: Knowledge distill via neuron selectivity transfer", *in*: *arXiv preprint arXiv:1707.01219* (2017).

[HYK16] Lu Hou, Quanming Yao, and James T Kwok, "Loss-aware binarization of deep networks", *in*: *arXiv preprint arXiv:1611.01600* (2016).

[HZS17] Yihui He, Xiangyu Zhang, and Jian Sun, "Channel pruning for accelerating very deep neural networks", *in*: *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 1389–1397.

[Ian+16] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size", *in*: *arXiv preprint arXiv:1602.07360* (2016).

[IEE85] IEEE, "IEEE Standard for Binary Floating-Point Arithmetic", *in*: *ANSI/IEEE Std 754-1985* (1985), pp. 1–20.

[IEEE-754] W Kahan, "754-2008–IEEE Standard for Floating-Point Arithmetic", *in*: *IEEE: Los Alamitos, CA, USA* (2008).

[Ioa+17] Yani Ioannou, Duncan Robertson, Roberto Cipolla, and Antonio Criminisi, "Deep roots: Improving cnn efficiency with hierarchical filter groups", *in*: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1231–1240.

[IS15] Sergey Ioffe and Christian Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", *in*: *International conference on machine learning*, pmlr, 2015, pp. 448–456.

[Ja79] Hartigan Ja, "A k-means clustering algorithm", *in*: *JR Stat. Soc. Ser. C-Appl. Stat.* 28 (1979), pp. 100–108.

[Jai+20] Sambhav Jain, Albert Gural, Michael Wu, and Chris Dick, "Trained Quantization Thresholds for Accurate and Efficient Fixed-Point Inference of Deep Neural Networks", *in*: *Proceedings of Machine Learning and Systems* 2 (2020), pp. 112–128.

[Jep+19] Jacob Høxbroe Jeppesen, Rune Hylsberg Jacobsen, Fadil Inceoglu, and Thomas Skjødeberg Toftegaard, "A cloud detection algorithm for satellite imagery based on deep learning", *in*: *Remote sensing of environment* 229 (2019), pp. 247–259.

[Jér19] Soubirane Jérôme, "Shaping the future of earth observation with pléiades neo", *in*: *2019 9th International Conference on Recent Advances in Space Technologies (RAST)*, IEEE, 2019, pp. 399–401.

[JGP16] Eric Jang, Shixiang Gu, and Ben Poole, "Categorical reparameterization with gumbel-softmax", *in*: *arXiv preprint arXiv:1611.01144* (2016).

[Ji19] Zhuoran Ji, "Hg-caffe: Mobile and embedded neural network gpu (opencl) inference engine with fp16 supporting", *in*: *arXiv preprint arXiv:1901.00858* (2019).

[Joh18] Jeff Johnson, "Rethinking floating point for deep learning", *in*: *arXiv preprint arXiv:1811.01721* (2018).

[Jos+24] Vijay Joshi et al., "An Efficient FPGA Implementation of a Simple Lossless Algorithm (SLA) for On-board Satellite Hyperspectral Data Compression", *in*: *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2024, pp. 1–5.

[Jud+16] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos, "Stripes: Bit-serial deep neural network computing", *in*: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2016, pp. 1–12.

[Jum+21] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al., "Highly accurate protein structure prediction with AlphaFold", *in*: *Nature* 596.*7873* (2021), pp. 583–589.

[JV22] Zuzana Jelčicová and Marian Verhelst, "Delta keyword transformer: Bringing transformers to the edge through dynamically pruned multi-head self-attention", *in*: *arXiv preprint arXiv:2204.03479* (2022).

[JWL18] Shunping Ji, Shiqing Wei, and Meng Lu, "Fully convolutional networks for multisource building extraction from an open aerial and satellite imagery data set", *in*: *IEEE Transactions on geoscience and remote sensing* 57.*1* (2018), pp. 574–586.

[JYL19] Qing Jin, Linjie Yang, and Zhenyu Liao, "Towards efficient training for neural network quantization", *in*: *arXiv preprint arXiv:1912.10207* (2019).

[Kal+19] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, et al., "A study of BFLOAT16 for deep learning training", *in*: *arXiv preprint arXiv:1905.12322* (2019).

[Kan+22] Yoojin Kang, Eunna Jang, Jungho Im, and Chungeun Kwon, "A deep learning model using geostationary satellite data for forest fire detection with reduced detection latency", *in*: *GIScience & Remote Sensing* 59.*1* (2022), pp. 2019–2035.

[KB14] Diederik P Kingma and Jimmy Ba, "Adam: A method for stochastic optimization", *in*: *arXiv preprint arXiv:1412.6980* (2014).

[KH09] A. Krizhevsky and G. Hinton, *Learning Multiple Layers of Features from Tiny Images*, tech. rep., University of Toronto, 2009.

[Kha+22] Salman Khan, Muzammal Naseer, Munawar Hayat, Syed Waqas Zamir, Fahad Shahbaz Khan, and Mubarak Shah, "Transformers in vision: A survey", *in*: *ACM computing surveys (CSUR)* 54.*10s* (2022), pp. 1–41.

[Kim+19] Jangho Kim, Yash Bhalgat, Jinwon Lee, Chirag Patel, and Nojun Kwak, "Qkd: Quantization-aware knowledge distillation", *in*: *arXiv preprint arXiv:1911.12491* (2019).

[Kim+20] Nahsung Kim, Dongyeob Shin, Wonseok Choi, Geonho Kim, and Jongsun Park, "Exploiting retraining-based mixed-precision quantization for low-cost DNN accelerator design", *in*: *IEEE Transactions on Neural Networks and Learning Systems* 32.*7* (2020), pp. 2925–2938.

[KK21] Sagar Karki and Siddhivinayak Kulkarni, "Ship detection and segmentation using unet", *in*: *2021 International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)*, IEEE, 2021, pp. 1–7.

[Kös+17] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Scott Gray, Stewart Hall, Luke Hornof, et al., "Flexpoint: An adaptive numerical format for efficient training of deep neural networks", *in*: *Advances in neural information processing systems* 30 (2017).

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton, "Imagenet classification with deep convolutional neural networks", *in*: *Advances in neural information processing systems* 25 (2012).

[Kun+22] Souvik Kundu, Shikai Wang, Qirui Sun, Peter A Beerel, and Massoud Pedram, "Bmpq: bit-gradient sensitivity-driven mixed-precision quantization of dnns from scratch", *in*: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2022, pp. 588–591.

[Kuz+22] Andrey Kuzmin, Mart Van Baalen, Yuwei Ren, Markus Nagel, Jorn Peters, and Tijmen Blankevoort, "Fp8 quantization: The power of the exponent", *in*: *Advances in Neural Information Processing Systems* 35 (2022), pp. 14651–14662.

[Lan+20] Hamed F Langroudi, Vedant Karia, John L Gustafson, and Dhireesha Kudithipudi, "Adaptive posit: Parameter aware numerical format for deep learning inference on the edge", *in*: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2020, pp. 726–727.

[LAT18] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr, "Snip: Single-shot network pruning based on connection sensitivity", *in*: *arXiv preprint arXiv:1810.02340* (2018).

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton, "Deep learning", *in*: *nature* 521.*7553* (2015), pp. 436–444.

[LDS89] Yann LeCun, John Denker, and Sara Solla, "Optimal brain damage", *in*: *Advances in neural information processing systems* 2 (1989).

[LeC+89] Yann LeCun, Bernhard Boser, John Denker, Donnie Henderson, Richard Howard, Wayne Hubbard, and Lawrence Jackel, "Handwritten digit recognition with a back-propagation network", *in*: *Advances in neural information processing systems* 2 (1989).

[LeC+98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner, "Gradient-based learning applied to document recognition", *in*: *Proceedings of the IEEE* 86.*11* (1998), pp. 2278–2324.

[LG16] Andrew Lavin and Scott Gray, "Fast algorithms for convolutional neural networks", *in*: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 4013–4021.

[LH16] Ilya Loshchilov and Frank Hutter, "Sgdr: Stochastic gradient descent with warm restarts", *in*: *arXiv preprint arXiv:1608.03983* (2016).

[LH17] Ilya Loshchilov and Frank Hutter, "Decoupled weight decay regularization", *in*: *arXiv preprint arXiv:1711.05101* (2017).

[LHC19] Zhenhao Li, Wei Hu, and Shuang Chen, "Design and implementation of CNN custom processor based on RISC-V architecture", *in*: *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, IEEE, 2019, pp. 1945–1950.

[Li+16a] Fengfu Li, Bin Liu, Xiaoxing Wang, Bo Zhang, and Junchi Yan, "Ternary weight networks", *in*: *arXiv preprint arXiv:1605.04711* (2016).

[Li+16b] Lin Li, Tiziana Fanni, Timo Viitanen, Renjie Xie, Francesca Palumbo, Luigi Raffo, Heikki Huttunen, Jarmo Takala, and Shuvra S Bhattacharyya, "Low power design methodology for signal processing systems using lightweight dataflow techniques", *in*: *2016 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, IEEE, 2016, pp. 82–89.

[Li+17] Yuncheng Li, Jianchao Yang, Yale Song, Liangliang Cao, Jiebo Luo, and Li-Jia Li, "Learning from noisy labels with distillation", *in*: *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 1910–1918.

[Li+18a] Ruirui Li, Wenjie Liu, Lei Yang, Shihao Sun, Wei Hu, Fan Zhang, and Wei Li, "DeepUNet: A deep fully convolutional network for pixel-level sea-land segmentation", *in*: *IEEE journal of selected topics in applied earth observations and remote sensing* 11.*11* (2018), pp. 3954–3962.

[Li+18b] Xiaomeng Li, Hao Chen, Xiaojuan Qi, Qi Dou, Chi-Wing Fu, and Pheng-Ann Heng, "H-DenseUNet: hybrid densely connected UNet for liver and tumor segmentation from CT volumes", *in*: *IEEE transactions on medical imaging* 37.*12* (2018), pp. 2663–2674.

[Li+20] Yuhang Li, Wei Wang, Haoli Bai, Ruihao Gong, Xin Dong, and Fengwei Yu, "Efficient bitwidth search for practical mixed precision neural network", *in*: *arXiv preprint arXiv:2003.07577* (2020).

[Lia+18] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei, "FP-BNN: Binarized neural network on FPGA", *in*: *Neurocomputing* 275 (2018), pp. 1072–1086.

[Lia+19] Xiaocong Lian, Zhenyu Liu, Zhourui Song, Jiwu Dai, Wei Zhou, and Xiangyang Ji, "High-performance FPGA-based CNN accelerator with block-floating-point arithmetic", *in*: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.*8* (2019), pp. 1874–1885.

[Lil+15] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra, "Continuous control with deep reinforcement learning", *in*: *arXiv preprint arXiv:1509.02971* (2015).

[Lin+15] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio, "Neural networks with few multiplications", *in*: *arXiv preprint arXiv:1510.03009* (2015).

[Liu+17a] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang, "Learning efficient convolutional networks through network slimming", *in*: *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2736–2744.

[Liu+17b] Zikun Liu, Liu Yuan, Lubin Weng, and Yiping Yang, "A high resolution optical satellite image dataset for ship recognition and some new baselines", *in*: *International conference on pattern recognition applications and methods*, vol. 2, SciTePress, 2017, pp. 324–331.

[Liu+18] Zechun Liu, Baoyuan Wu, Wenhan Luo, Xin Yang, Wei Liu, and Kwang-Ting Cheng, "Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm", *in*:

*Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 722–737.

[Liu+21] Zhenhua Liu, Xinfeng Zhang, Shanshe Wang, Siwei Ma, and Wen Gao, "Evolutionary quantization of neural networks with mixed-precision", *in*: *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2021, pp. 2785–2789.

[LL16] Vadim Lebedev and Victor Lempitsky, "Fast convnets using group-wise brain damage", *in*: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2554–2564.

[LLC23] Shih-Yang Liu, Zechun Liu, and Kwang-Ting Cheng, "Oscillation-free quantization for low-bit vision transformers", *in*: *International Conference on Machine Learning*, PMLR, 2023, pp. 21813–21824.

[LM87] Edward A Lee and David G Messerschmitt, "Synchronous data flow", *in*: *Proceedings of the IEEE* 75.*9* (1987), pp. 1235–1245.

[Lou+19a] Qian Lou, Feng Guo, Lantao Liu, Minje Kim, and Lei Jiang, "Autoq: Automated kernel-wise neural network quantization", *in*: *arXiv preprint arXiv:1902.05690* (2019).

[Lou+19b] Marcia Sahaya Louis, Zahra Azad, Leila Delshadtehrani, Suyog Gupta, Pete Warden, Vijay Janapa Reddi, and Ajay Joshi, "Towards deep learning using tensorflow lite on risc-v", *in*: *Third Workshop on Computer Architecture Research with RISC-V (CARRV)*, vol. 1, 2019, p. 6.

[LS23] Yang Lyu and Donglin Su, "Application of Deep Learning in the Search and a Certain Celestial Body", *in*: *Procedia Computer Science* 228 (2023), pp. 366–372.

[LSD15] Jonathan Long, Evan Shelhamer, and Trevor Darrell, "Fully convolutional networks for semantic segmentation", *in*: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440.

[LSY18] Hanxiao Liu, Karen Simonyan, and Yiming Yang, "Darts: Differentiable architecture search", *in*: *arXiv preprint arXiv:1806.09055* (2018).

[LSZ17] Haoning Lin, Zhenwei Shi, and Zhengxia Zou, "Fully convolutional network with task partitioning for inshore ship detection in optical remote sensing images", *in*: *IEEE Geoscience and Remote Sensing Letters* 14.*10* (2017), pp. 1665–1669.

[LTA18] Griffin Lacey, Graham W Taylor, and Shawki Areibi, "Stochastic layer-wise precision in deep neural networks", *in*: *arXiv preprint arXiv:1807.00942* (2018).

[LY15] Ya Le and Xuan Yang, "Tiny imagenet visual recognition challenge", *in*: *CS 231N* 7.*7* (2015), p. 3.

[LZP17] Xiaofan Lin, Cong Zhao, and Wei Pan, "Towards accurate binary convolutional neural network", *in*: *Advances in neural information processing systems* 30 (2017).

[Ma+20] Xiaolong Ma, Wei Niu, Tianyun Zhang, Sijia Liu, Sheng Lin, Hongjia Li, Wujie Wen, Xiang Chen, Jian Tang, Kaisheng Ma, et al., "An image enhancing pattern-based sparsity for real-time inference on mobile devices", *in*: *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XIII 16*, Springer, 2020, pp. 629–645.

[Ma+21] Yuexiao Ma, Taisong Jin, Xiawu Zheng, Yan Wang, Huixia Li, Yongjian Wu, Guannan Jiang, Wei Zhang, and Rongrong Ji, "OMPQ: Orthogonal Mixed Precision Quantization", *in*: *arXiv preprint arXiv:2109.07865* (2021).

[Man+21] Laura Manoliu, Benjamin Schoch, Markus Koller, Jens Wieczorek, Sabine Klinkner, and Ingmar Kallfass, "High-speed FPGA-based payload computer for an in-orbit verification of a 71–76 GHz satellite downlink", *in*: *2021 IEEE Space Hardware and Radio Conference (SHaRC)*, IEEE, 2021, pp. 21–24.

[Mao+20] Yuxing Mao, Yuqin Yang, Ziyuan Ma, Mingzhe Li, Hao Su, and Jun Zhang, "Efficient low-cost ship detection for SAR imagery based on simplified U-net", *in*: *IEEE Access* 8 (2020), pp. 69742–69753.

[Meh+18] Sachin Mehta, Mohammad Rastegari, Anat Caspi, Linda Shapiro, and Hannaneh Hajishirzi, "Espnet: Efficient spatial pyramid of dilated convolutions for semantic segmentation", *in*: *Proceedings of the european conference on computer vision (ECCV)*, 2018, pp. 552–568.

[Mel+19a] Naveen Mellempudi, Sudarshan Srinivasan, Dipankar Das, and Bharat Kaul, "Mixed precision training with 8-bit floating point", *in*: *arXiv preprint arXiv:1905.12334* (2019).

[Mel+19b] Eldad Meller, Alexander Finkelstein, Uri Almog, and Mark Grobman, "Same, same but different: Recovering neural network quantization error through weight factorization", *in*: *International Conference on Machine Learning*, PMLR, 2019, pp. 4486–4495.

[Met+21] Paul Metzgen, Shaoxia Fang, Satyaprakash Pareek, Bing Tian, Yi Shan, Elliott Delaye, and Ashish Sirasao, *Higher Performance Neural Networks with Small Floating Point*, tech. rep. WP530, AMD-Xilinx, 2021.

[MG12] Franck Mamalet and Christophe Garcia, "Simplifying convnets for fast learning", *in*: *International Conference on Artificial Neural Networks*, Springer, 2012, pp. 58–65.

[MHL13] Michael Mathieu, Mikael Henaff, and Yann LeCun, "Fast training of convolutional networks through ffts", *in*: *arXiv preprint arXiv:1312.5851* (2013).

[MHN+13] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al., "Rectifier nonlinearities improve neural network acoustic models", *in*: *Proc. icml*, vol. 30, 1, Atlanta, GA, 2013, p. 3.

[Mic+17] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al., "Mixed precision training", *in*: *arXiv preprint arXiv:1710.03740* (2017).

[Mic+22] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, et al., "FP8 formats for deep learning", *in*: *arXiv preprint arXiv:2209.05433* (2022).

[Mis+17] Asit Mishra, Eriko Nurvitadhi, Jeffrey J Cook, and Debbie Marr, "WRPN: Wide reduced-precision networks", *in*: *arXiv preprint arXiv:1709.01134* (2017).

[Mol+16] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz, "Pruning convolutional neural networks for resource efficient inference", *in*: *arXiv preprint arXiv:1611.06440* (2016).

[Mol+19] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz, "Importance estimation for neural network pruning", *in*: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 11264–11272.

[MOM12] Grégoire Montavon, Geneviève Orr, and Klaus-Robert Müller, *Neural networks: tricks of the trade*, vol. 7700, springer, 2012.

[Mos+17] Duncan JM Moss, Eriko Nurvitadhi, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip HW Leong, "High performance binary neural networks on the Xeon+ FPGA™ platform", *in*: *2017 27Th International conference on field programmable logic and applications (FPL)*, IEEE, 2017, pp. 1–4.

[Moz17] Paul Mozur, "Google's AlphaGo defeats Chinese go master in win for AI", *in*: *International New York Times* (2017), NA–NA.

[MP43] Warren S McCulloch and Walter Pitts, "A logical calculus of the ideas immanent in nervous activity", *in*: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133.

[MS19] S. Mohajerani and P. Saeedi, "Cloud-Net: An End-To-End Cloud Detection Algorithm for Landsat 8 Imagery", *in*: *IGARSS 2019 - 2019 IEEE International Geoscience and Remote Sensing Symposium*, 2019, pp. 1029–1032.

[Nac+18] Ofir Nachum, Shixiang Shane Gu, Honglak Lee, and Sergey Levine, "Data-efficient hierarchical reinforcement learning", *in*: *Advances in neural information processing systems* 31 (2018).

[Nag+19] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling, "Data-free quantization through weight equalization and bias correction", *in*: *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 1325–1334.

[Nag+20] Markus Nagel, Rana Ali Amjad, Mart Van Baalen, Christos Louizos, and Tijmen Blankevoort, "Up or down? adaptive rounding for post-training quantization", *in*: *International Conference on Machine Learning*, PMLR, 2020, pp. 7197–7206.

[Nag+21] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart Van Baalen, and Tijmen Blankevoort, "A white paper on neural network quantization", *in*: *arXiv preprint arXiv:2106.08295* (2021).

[Nag+22] Markus Nagel, Marios Fournarakis, Yelysei Bondarenko, and Tijmen Blankevoort, "Overcoming oscillations in quantization-aware training", *in*: *International Conference on Machine Learning*, PMLR, 2022, pp. 16318–16330.

[Nah+21] Yury Nahshan, Brian Chmiel, Chaim Baskin, Evgenii Zheltonozhskii, Ron Banner, Alex M Bronstein, and Avi Mendelson, "Loss aware post-training quantization", *in*: *Machine Learning* 110.*11* (2021), pp. 3245–3262.

[Nes83] Yurii Nesterov, "A method of solving a convex programming problem with convergence rate O (1/k** 2)", *in*: *Doklady Akademii Nauk SSSR* 269.*3* (1983), p. 543.

[NH10] Vinod Nair and Geoffrey E Hinton, "Rectified linear units improve restricted boltzmann machines", *in*: *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[NHH15] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han, "Learning deconvolution network for semantic segmentation", *in*: *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1520–1528.

[Nie+18] Shanlan Nie, Zhiguo Jiang, Haopeng Zhang, Bowen Cai, and Yuan Yao, "Inshore Ship Detection Based on Mask R-CNN", *in*: *IGARSS 2018-2018 IEEE International Geoscience and Remote Sensing Symposium*, IEEE, 2018, pp. 693–696.

[Nik+20] Miloš Nikolić, Ghouthi Boukli Hacene, Ciaran Bannon, Alberto Delmas Lascorz, Matthieu Courbariaux, Yoshua Bengio, Vincent Gripon, and Andreas Moshovos, "BitPruning: Learning Bitlengths for Aggressive and Accurate Quantization", *in*: *arXiv preprint arXiv:2002.03090* (2020).

[Nin+20] Lin Ning, Guoyang Chen, Weifeng Zhang, and Xipeng Shen, "Simple augmentation goes a long way: Adrl for dnn quantization", *in*: *International Conference on Learning Representations*, 2020.

[Nin+21] Fernando Nino, Clement Coggiola, Denis Blumstein, Léa Lasson, and Stéphane Calmant, "Monitoring of inland water levels by satellite altimetry and deep learning", *in*: *IEEE Transactions on Geoscience and Remote Sensing* 60 (2021), pp. 1–14.

[Niu+20] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren, "Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning", *in*: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 907–922.

[Nou+22] Badreddine Noune, Philip Jones, Daniel Justus, Dominic Masters, and Carlo Luschi, "8-bit numerical formats for deep neural networks", *in*: *arXiv preprint arXiv:2206.02915* (2022).

[Okt+18] Ozan Oktay, Jo Schlemper, Loic Le Folgoc, Matthew Lee, Mattias Heinrich, Kazunari Misawa, Kensaku Mori, Steven McDonagh, Nils Y Hammerla, Bernhard Kainz, et al., "Attention u-net: Learning where to look for the pancreas", *in*: *arXiv preprint arXiv:1804.03999* (2018).

[Ovt+15] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung, "Accelerating deep convolutional neural networks using specialized hardware", *in*: *Microsoft Research Whitepaper* 2.*11* (2015), pp. 1–4.

[P3109] IEEE Working Group P3109, *Interim Report on 8-bit Binary Floating-point Formats*, tech. rep., IEEE, 2024.

[Pan+23] Nilesh Prasad Pandey, Markus Nagel, Mart van Baalen, Yin Huang, Chirag Patel, and Tijmen Blankevoort, "A practical mixed precision algorithm for post-training quantization", *in*: *arXiv preprint arXiv:2302.05397* (2023).

[Pap+22] Alessandro Pappalardo, Yaman Umuroglu, Michaela Blott, Jovan Mitrevski, Ben Hawks, Nhan Tran, Vladimir Loncar, Sioni Summers, Hendrik Borras, Jules Muhizi, et al., "Qonnx: Representing arbitrary-precision quantized neural networks", *in*: *arXiv preprint arXiv:2206.07527* (2022).

[Pap23] Alessandro Pappalardo, *Xilinx/brevitas*, 2023.

[Par+15] Seong-Wook Park, Junyoung Park, Kyeongryeol Bong, Dongjoo Shin, Jinmook Lee, Sungpill Choi, and Hoi-Jun Yoo, "An energy-efficient and scalable deep learning/inference processor with tetra-parallel MIMD architecture for big data applications", *in*: *IEEE transactions on biomedical circuits and systems* 9.*6* (2015), pp. 838–848.

[Pat+21] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean, "Carbon emissions and large neural network training", *in*: *arXiv preprint arXiv:2104.10350* (2021).

[Pik+22] Erion Vasilis Pikoulis, Christos Mavrokefalidis, Stavros Nousias, and Aris S Lalos, "A new clustering-based technique for the acceleration of deep convolutional networks", *in*: *Deep Learning Applications, Volume 3* (2022), pp. 123–150.

[Pol64] Boris T Polyak, "Some methods of speeding up the convergence of iteration methods", *in*: *Ussr computational mathematics and mathematical physics* 4.*5* (1964), pp. 1–17.

[PPA18] Antonio Polino, Razvan Pascanu, and Dan Alistarh, "Model compression via distillation and quantization", *in*: *arXiv preprint arXiv:1802.05668* (2018).

[Pri23] Simon JD Prince, *Understanding Deep Learning*, MIT press, 2023.

[Pro+17] Adrien Prost-Boucle, Alban Bourge, Frédéric Pétrot, Hande Alemdar, Nicholas Caldwell, and Vincent Leroy, "Scalable high-performance architecture for convolutional ternary neural networks on FPGA", *in*: *2017 27Th International conference on field programmable logic and applications (FPL)*, IEEE, 2017, pp. 1–7.

[Qin+20] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe, "Binary neural networks: A survey", *in*: *Pattern Recognition* 105 (2020), p. 107281.

[Ras+16] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks", *in*: *European conference on computer vision*, Springer, 2016, pp. 525–542.

[Raz+17] Mohammad Samragh Razlighi, Mohsen Imani, Farinaz Koushanfar, and Tajana Rosing, "Looknn: Neural network with no multiplication", *in*: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, IEEE, 2017, pp. 1775–1780.

[Ren+15] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks", *in*: *Advances in neural information processing systems* 28 (2015).

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox, "U-net: Convolutional networks for biomedical image segmentation", *in*: *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*, Springer, 2015, pp. 234–241.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams, "Learning representations by back-propagating errors", *in*: *nature* 323.*6088* (1986), pp. 533–536.

[Rom+14] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio, "Fitnets: Hints for thin deep nets", *in*: *arXiv preprint arXiv:1412.6550* (2014).

[Ros58] Frank Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.", *in*: *Psychological review* 65.*6* (1958), p. 386.

[RTR21] Gonçalo Raposo, Pedro Tomás, and Nuno Roma, "Positnn: Training deep neural networks with mixed low-precision posit", *in*: *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2021, pp. 7908–7912.

[Rus+20] Manuele Rusci, Marco Fariselli, Alessandro Capotondi, and Luca Benini, "Leveraging automated mixed-low-precision quantization for tiny edge microcontrollers", *in*: *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning: Second International Workshop, IoT Streams 2020, and First International Workshop, ITEM 2020, Co-located with ECML/PKDD 2020, Ghent, Belgium, September 14-18, 2020, Revised Selected Papers 2*, Springer, 2020, pp. 296–308.

[RW73] P Ready and P Wintz, "Information extraction, SNR improvement, and data compression in multispectral imagery", *in*: *IEEE Transactions on communications* 21.*10* (1973), pp. 1123–1131.

[Ryb+18] Vladimir Rybalkin, Alessandro Pappalardo, Muhammad Mohsin Ghaffar, Giulio Gambardella, Norbert Wehn, and Michaela Blott, "FINN-L: Library extensions and design trade-off analysis for variable precision LSTM networks on FPGAs", *in*: *2018 28th international conference on field programmable logic and applications (FPL)*, IEEE, 2018, pp. 89–897.

[RZL17] Prajit Ramachandran, Barret Zoph, and Quoc V Le, "Searching for activation functions", *in*: *arXiv preprint arXiv:1710.05941* (2017).

[San+09] Murugan Sankaradas, Venkata Jakkula, Srihari Cadambi, Srimat Chakradhar, Igor Durdanovic, Eric Cosatto, and Hans Peter Graf, "A massively parallel coprocessor for convolutional neural networks", *in*: *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, IEEE, 2009, pp. 53–60.

[San+18] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks", *in*: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[SBS20] Sungho Shin, Yoonho Boo, and Wonyong Sung, "Knowledge distillation for optimization of quantized deep neural networks", *in*: *2020 IEEE Workshop on Signal Processing Systems (SiPS)*, IEEE, 2020, pp. 1–6.

[Sch+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov, "Proximal policy optimization algorithms", *in*: *arXiv preprint arXiv:1707.0634* (2017).

[Set+18] Sean O Settle, Manasa Bollavaram, Paolo D'Alberto, Elliott Delaye, Oscar Fernandez, Nicholas Fraser, Aaron Ng, Ashish Sirasao, and Michael Wu, "Quantizing convolutional neural networks for low-power high-throughput inference engines", *in*: *arXiv preprint arXiv:1805.07941* (2018).

[Sha+18] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Joon Kyung Kim, Vikas Chandra, and Hadi Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural net-

work", *in*: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2018, pp. 764–775.

[Sha+19] Pourya Shamsolmoali, Masoumeh Zareapoor, Ruili Wang, Huiyu Zhou, and Jie Yang, "A novel deep structure U-Net for sea-land segmentation in remote sensing images", *in*: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 12.*9* (2019), pp. 3219–3232.

[Sha48] Claude Elwood Shannon, "A mathematical theory of communication", *in*: *The Bell system technical journal* 27.*3* (1948), pp. 379–423.

[She+10] C Shen, William Plishker, H Wu, and Shuvra S Bhattacharyya, "A lightweight dataflow approach for design and implementation of SDR systems", *in*: *Proceedings of the Wireless Innovation Conference and Product Exposition*, 2010, pp. 640–645.

[She+20] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer, "Q-bert: Hessian based ultra low precision quantization of bert", *in*: *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 05, 2020, pp. 8815–8821.

[She+23] Haihao Shen, Naveen Mellempudi, Xin He, Qun Gao, Chang Wang, and Mengni Wang, "Efficient Post-training Quantization with FP8 Formats", *in*: *arXiv preprint arXiv:2309.14592* (2023).

[She97] William Fleetwood Sheppard, "On the Calculation of the most Probable Values of Frequency-Constants, for Data arranged according to Equidistant Division of a Scale", *in*: *Proceedings of the London Mathematical Society* 1.*1* (1897), pp. 353–380.

[SM22] Olivier Sentieys and Daniel Menard, "Approximate Computing Techniques - From Component- to Application-Level", *in*: Springer, 2022, chap. Customizing Number Representation and Precision.

[Smi17] Leslie N Smith, "Cyclical learning rates for training neural networks", *in*: *2017 IEEE winter conference on applications of computer vision (WACV)*, IEEE, 2017, pp. 464–472.

[SNL18] Sanghyun Son, Seungjun Nah, and Kyoung Mu Lee, "Clustering convolutional kernels to compress deep neural networks", *in*: *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 216–232.

[Spa19] The Consultative Committee for Space Data Systems, *Low-Complexity Lossless and Near-Lossless Multispectral and Hyperspectral Image Compression*, Standard, Washington, DC, USA: National Aeronautics and Space Administration, 2019.

[SR21] Efstathia Soufleri and Kaushik Roy, "Network compression via mixed precision quantization using a multi-layer perceptron for the bit-width allocation", *in*: *IEEE Access* 9 (2021), pp. 135059–135068.

[Sri+10] Vinay Sriram, David Cox, Kuen Hung Tsoi, and Wayne Luk, "Towards an embedded biologically-inspired machine vision processor", *in*: *2010 International Conference on Field-Programmable Technology*, IEEE, 2010, pp. 273–278.

[Sri+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting", *in*: *The journal of machine learning research* 15.*1* (2014), pp. 1929–1958.

[Sud+17] Carole H Sudre, Wenqi Li, Tom Vercauteren, Sebastien Ourselin, and M Jorge Cardoso, "Generalised dice overlap as a deep learning loss function for highly unbalanced segmentations", *in*: *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support: Third International Workshop, DLMIA 2017, and 7th International Workshop, ML-CDS 2017, Held in Conjunction with MICCAI 2017, Québec City, QC, Canada, September 14, Proceedings 3*, Springer, 2017, pp. 240–248.

[Sun+19] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Viji Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan, "Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks", *in*: *Advances in neural information processing systems* 32 (2019).

[Sun+21] Qigong Sun, Licheng Jiao, Yan Ren, Xiufang Li, Fanhua Shang, and Fang Liu, "Effective and fast: A novel sequential single path search for mixed-precision quantization", *in*: *arXiv preprint arXiv:2103.02904* (2021).

[Sun+22a] Mengshu Sun, Zhengang Li, Alec Lu, Yanyu Li, Sung-En Chang, Xiaolong Ma, Xue Lin, and Zhenman Fang, "Film-qnn: Efficient fpga acceleration of deep neural networks with intra-layer, mixed-precision quantization", *in*: *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 134–145.

[Sun+22b] Zhenhong Sun, Ce Ge, Junyan Wang, Ming Lin, Hesen Chen, Hao Li, and Xiuyu Sun, "Entropy-driven mixed-precision quantization for deep network design", *in*: *Advances in Neural Information Processing Systems* 35 (2022), pp. 21508–21520.

[Swa+21] R Michael Swan, Deegan Atha, Henry A Leopold, Matthew Gildner, Stephanie Oij, Cindy Chiu, and Masahiro Ono, "Ai4mars: A dataset for terrain-aware autonomous driving on mars", *in*: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 1982–1991.

[SZ14] Karen Simonyan and Andrew Zisserman, "Very deep convolutional networks for large-scale image recognition", *in*: *arXiv preprint arXiv:1409.1556* (2014).

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

[SZZ20] Marzuraikah Mohd Stofa, Mohd Asyraf Zulkifley, and Siti Zulaikha Muhammad Zaki, "A deep learning approach to ship detection using satellite imagery", *in*: *IOP Conference Series: Earth and Environmental Science*, vol. 540, 1, IOP Publishing, 2020, p. 012049.

[Tam+20] Thierry Tambe, En-Yu Yang, Zishen Wan, Yuntian Deng, Vijay Janapa Reddi, Alexander Rush, David Brooks, and Gu-Yeon Wei, "Algorithm-hardware co-design of adaptive floating-point encodings for resilient deep learning inference", *in*: *2020 57th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2020, pp. 1–6.

[Tan+20] Hidenori Tanaka, Daniel Kunin, Daniel L Yamins, and Surya Ganguli, "Pruning neural networks without any data by iteratively conserving synaptic flow", *in*: *Advances in neural information processing systems* 33 (2020), pp. 6377–6389.

[Tan+22] Chen Tang, Kai Ouyang, Zhi Wang, Yifei Zhu, Wen Ji, Yaowei Wang, and Wenwu Zhu, "Mixed-precision neural network quantization via learned layer-wise importance", *in*: *European Conference on Computer Vision*, Springer, 2022, pp. 259–275.

[TGH08] Andrew J Tatem, Scott J Goetz, and Simon I Hay, "Fifty years of earth observation satellites: Views from above have lead to countless advances on the ground in both scientific knowledge and daily life", *in*: *American scientist* 96.*5* (2008), p. 390.

[Tib96] Robert Tibshirani, "Regression shrinkage and selection via the lasso", *in*: *Journal of the Royal Statistical Society Series B: Statistical Methodology* 58.*1* (1996), pp. 267–288.

[TM19] Frederick Tung and Greg Mori, "Similarity-Preserving Knowledge Distillation", *in*: *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 1365–1374.

[Tor+22] Yvan Tortorella, Luca Bertaccini, Davide Rossi, Luca Benini, and Francesco Conti, "RedMulE: A compact FP16 matrix-multiplication accelerator for adaptive deep learning on RISC-V-based ultra-low-power SoCs", *in*: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2022, pp. 1099–1102.

[Tse01] Paul Tseng, "Convergence of a block coordinate descent method for nondifferentiable minimization", *in*: *Journal of optimization theory and applications* 109 (2001), pp. 475–494.

[UA19] Irem Ulku and Erdem Akagunduz, "A Survey on Deep Learning-based Architectures for Semantic Segmentation on 2D images", *in*: *arXiv preprint arXiv:1912.10230* (2019).

[Uhl+19] Stefan Uhlich, Lukas Mauch, Kazuki Yoshiyama, Fabien Cardinaux, Javier Alonso Garcia, Stephen Tiedemann, Thomas Kemp, and Akira Nakamura, "Differentiable quantization of deep neural networks", *in*: *arXiv preprint arXiv:1905.11452* 2.*8* (2019).

[Umu+17] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers, "Finn: A framework for fast, scalable binarized neural network inference", *in*: *Proceedings of the 2017*

*ACM/SIGDA international symposium on field-programmable gate arrays*, 2017, pp. 65–74.

[UMW17] Karen Ullrich, Edward Meeds, and Max Welling, "Soft weight-sharing for neural network compression", *in*: *arXiv preprint arXiv:1702.04008* (2017).

[URS18] Yaman Umuroglu, Lahiru Rasnayake, and Magnus Själander, "Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing", *in*: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2018, pp. 307–3077.

[USB11] Ruth Urner, Shai Shalev-Shwartz, and Shai Ben-David, "Access to unlabeled data can speed up prediction time", *in*: *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 641–648.

[Van+20] Mart Van Baalen, Christos Louizos, Markus Nagel, Rana Ali Amjad, Ying Wang, Tijmen Blankevoort, and Max Welling, "Bayesian bits: Unifying quantization and pruning", *in*: *Advances in neural information processing systems* 33 (2020), pp. 5741–5752.

[Vas+21] Karina Vasquez, Yeshwanth Venkatesha, Abhiroop Bhattacharjee, Abhishek Moitra, and Priyadarshini Panda, "Activation density based mixed-precision quantization for energy efficient neural networks", *in*: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2021, pp. 1360–1365.

[VB16] Stylianos I Venieris and Christos-Savvas Bouganis, "fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs", *in*: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2016, pp. 40–47.

[VB17] Stylianos I Venieris and Christos-Savvas Bouganis, "Latency-driven design for FPGA-based convolutional neural networks", *in*: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2017, pp. 1–8.

[VD23] Antoine Vanderschueren and Christophe De Vleeschouwer, "Are Straight-Through gradients and Soft-Thresholding all you need for Sparse Training?", *in*: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2023, pp. 3808–3817.

[VXN20] Sagar Vaze, Weidi Xie, and Ana IL Namburete, "Low-Memory CNNs Enabling Real-Time Ultrasound Segmentation Towards Mobile Deployment", *in*: *IEEE Journal of Biomedical and Health Informatics* 24.*4* (2020), pp. 1059–1069.

[Wan+18] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers", *in*: *Advances in neural information processing systems* 31 (2018).

[Wan+19a] Huan Wang, Qiming Zhang, Yuehai Wang, Lu Yu, and Haoji Hu, "Structured pruning for efficient convnets via incremental regularization", *in*: *2019 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2019, pp. 1–8.

[Wan+19b] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han, "HAQ: Hardware-Aware Automated Quantization With Mixed Precision", *in*: *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019, pp. 8604–8612.

[Wan+20] Tianzhe Wang, Kuan Wang, Han Cai, Ji Lin, Zhijian Liu, Hanrui Wang, Yujun Lin, and Song Han, "Apq: Joint search for network architecture, pruning and quantization policy", *in*: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 2078–2087.

[Wan+21] Ziwei Wang, Han Xiao, Jiwen Lu, and Jie Zhou, "Generalizable mixed-precision quantization via attribution rank preservation", *in*: *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 5291–5300.

[Wan+22] Haonan Wang, Peng Cao, Jiaqi Wang, and Osmar R Zaiane, "Uctransnet: rethinking the skip connections in u-net from a channel-wise perspective with transformer", *in*: *Proceedings of the AAAI conference on artificial intelligence*, vol. 36, 3, 2022, pp. 2441–2449.

[Wan+24] Yingchun Wang, Song Guo, Jingcai Guo, Yuanhong Zhang, Weizhan Zhang, Qinghua Zheng, and Jie Zhang, "Data quality-aware mixed-precision quantization via hybrid reinforcement learning", *in*: *IEEE Transactions on Neural Networks and Learning Systems* (2024).

[Wu+16] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng, "Quantized convolutional neural networks for mobile devices", *in*: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 4820–4828.

[Wu+18a] Bichen Wu, Yanghan Wang, Peizhao Zhang, Yuandong Tian, Peter Vajda, and Kurt Keutzer, "Mixed Precision Quantization of ConvNets via Differentiable Neural Architecture Search", *in*: *arXiv preprint arXiv:1812.00090* (2018).

[Wu+18b] Junru Wu, Yue Wang, Zhenyu Wu, Zhangyang Wang, Ashok Veeraraghavan, and Yingyan Lin, "Deep k-means: Re-training and parameter sharing with harder cluster assignments for compressing deep convolutions", *in*: *International Conference on Machine Learning*, PMLR, 2018, pp. 5363–5372.

[Wu+20] Chen Wu, Mingyu Wang, Xiayu Li, Jicheng Lu, Kun Wang, and Lei He, "Phoenix: A low-precision floating-point quantization oriented architecture for convolutional neural networks", *in*: *arXiv preprint arXiv:2003.02628* (2020).

[XCS23] Yu Xue, Chen Chen, and Adam Słowik, "Neural architecture search based on a multi-objective evolutionary algorithm with probability stack", *in*: *IEEE Transactions on Evolutionary Computation* 27.*4* (2023), pp. 778–786.

[Xu+18] Xiaowei Xu, Yukun Ding, Sharon Xiaobo Hu, Michael Niemier, Jason Cong, Yu Hu, and Yiyu Shi, "Scaling for edge inference of deep neural networks", *in*: *Nature Electronics* 1.*4* (2018), pp. 216–222.

[Xu+22] Ming Xu, Liang Chen, Hao Shi, Zhu Yang, Jiahao Li, and Teng Long, "FPGA-based implementation of ship detection for satellite on-board processing", *in*: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 15 (2022), pp. 9733–9745.

[XY17] Yangyang Xu and Wotao Yin, "A globally convergent algorithm for nonconvex optimization based on block coordinate update", *in*: *Journal of Scientific Computing* 72.*2* (2017), pp. 700–734.

[Yan+20] Dingcheng Yang, Wenjian Yu, Ao Zhou, Haoyuan Mu, Gary Yao, and Xiaoyi Wang, "DP-Net: Dynamic programming guided deep neural network compression", *in*: *arXiv preprint arXiv:2003.09615* (2020).

[Yan+21] H Yang, L Duan, Y Chen, and H Li, "BSQ: Exploring Bit-Level Sparsity for Mixed-Precision Neural Network Quantization", *in*: *International Conference on Learning Representations*, 2021.

[Yao+21] Zhewei Yao, Zhen Dong, Zhangcheng Zheng, Amir Gholami, Jiali Yu, Eric Tan, Leyuan Wang, Qijing Huang, Yida Wang, Michael Mahoney, et al., "HAWQ-V3: Dyadic Neural Network Quantization", *in*: *International Conference on Machine Learning*, PMLR, 2021, pp. 11875–11886.

[Yao99] Xin Yao, "Evolving artificial neural networks", *in*: *Proceedings of the IEEE* 87.*9* (1999), pp. 1423–1447.

[YJ21] Linjie Yang and Qing Jin, "FracBits: Mixed Precision Quantization via Fractional Bit-Widths", *in*: *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021, pp. 10612–10620.

[Yu+20] Haibao Yu, Qi Han, Jianbo Li, Jianping Shi, Guangliang Cheng, and Bin Fan, "Search what you want: Barrier panelty nas for mixed precision quantization", *in*: *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part IX 16*, Springer, 2020, pp. 1–16.

[Yua+20] Yong Yuan, Chen Chen, Xiyuan Hu, and Silong Peng, "Evoq: Mixed precision quantization of dnns via sensitivity guided evolutionary search", *in*: *2020 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2020, pp. 1–8.

[ZBS22] Chiyuan Zhang, Samy Bengio, and Yoram Singer, "Are all layers created equal?", *in*: *Journal of Machine Learning Research* 23.*67* (2022), pp. 1–28.

[Zha+15] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks", *in*: *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, 2015, pp. 161–170.

[Zha+18a] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua, "LQ-Nets: Learned Quantization for Highly Accurate and Compact Deep Neural Networks", *in*: *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 365–382.

[Zha+18b] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices", *in*: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 6848–6856.

[Zha+19] Shaoming Zhang, Ruize Wu, Kunyuan Xu, Jianmei Wang, and Weiwei Sun, "R-CNN-Based Ship Detection from High Resolution Remote Sensing Imagery", *in*: *Remote Sensing* 11.*6* (2019), p. 631.

[Zha+21a] Zhaoyang Zhang, Wenqi Shao, Jinwei Gu, Xiaogang Wang, and Ping Luo, "Differentiable Dynamic Quantization with Mixed Precision and Adaptive Resolution", *in*: *International Conference on Machine Learning*, PMLR, 2021, pp. 12546–12556.

[Zha+21b] Zhengning Zhang, Lin Zhang, Yue Wang, Pengming Feng, and Ran He, "ShipRSImageNet: A large-scale fine-grained dataset for ship detection in high-resolution optical remote sensing images", *in*: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 14 (2021), pp. 8458–8472.

[Zha+23] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola, *Dive into Deep Learning*, Cambridge University Press, 2023.

[ZHK19] Hao Zhang, Jiongrui He, and Seok-Bum Ko, "Efficient posit multiply-accumulate unit generator for deep learning applications", *in*: *2019 IEEE international symposium on circuits and systems (ISCAS)*, IEEE, 2019, pp. 1–5.

[Zho+16] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients", *in*: *arXiv:1606.06160* (2016).

[Zho+18] Zongwei Zhou, Md Mahfuzur Rahman Siddiquee, Nima Tajbakhsh, and Jianming Liang, "Unet++: A nested u-net architecture for medical image segmentation", *in*: *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support: 4th International Workshop, DLMIA 2018, and 8th International Workshop, ML-CDS 2018, Held in Conjunction with MICCAI 2018, Granada, Spain, September 20, 2018, Proceedings 4*, Springer, 2018, pp. 3–11.

[Zhu+16] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally, "Trained ternary quantization", *in*: *arXiv preprint arXiv:1612.01064* (2016).

[ZYH21] Sijie Zhao, Tao Yue, and Xuemei Hu, "Distribution-aware adaptive multi-bit quantization. In 2021 IEEE", *in*: *CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021, pp. 9277–9286.

COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

Université de Rennes

**Titre :** Méthodes de Quantification des Réseaux de Neurones pour le Traitement Embarqué d'Images Satellites sur FPGA

**Mot clés :** Réseaux de Neurones Profonds, Précision Limitée, Quantification Pendant l'Entraînement, Optimisation Adaptative de la Largeur de Bits, Petit Flottant, Détection de Navires, Segmentation Sémantique, FPGA

**Résumé :** L'apprentissage profond commence à être employé avec succès dans diverses applications spatiales. Cependant, en raison de son empreinte mémoire et de son volume de calcul, la phase d'inférence de nombreux modèles est encore principalement réalisée sur des plateformes terrestres. Des recherches récentes se sont intéressées à la quantification des réseaux de neurones afin de réduire ce volume de calcul et de faciliter le traitement embarqué. Cette thèse étudie la quantification des réseaux de neurones et son impact sur la précision des prédictions des modèles destinés aux accélérateurs FPGA pour les applications spatiales embarquées. Nous introduisons une méthode d'optimisation pour la quantification de nombres entiers uniformes en précision mixte des poids et des activations. Sa caractéristique principale est l'utilisation de largeurs de bits fractionnaires relâchés mises à jour à l'aide d'une règle de descente de gradient, mais par ailleurs discrétisées pour toutes les opérations (lors les calculs progressif et rétrogressif). Nous présentons également une approche performante basée sur la quantification en virgule flottante de faible précision. Notre méthode adapte une approche d'entraînement de réseaux de neurones profonds quantifiés, principalement utilisée pour la quantification en nombres entiers/virgule fixe. Enfin, nous avons étudié l'impact des sauts de connexion longs des modèles de segmentation sémantique en forme de «U» sur la précision des prédictions et mis en évidence leur empreinte mémoire.

**Title:** Neural Network Quantization Methods for FPGA On-Board Processing of Satellite Images

**Keywords:** Deep Neural Networks, Reduced Precision, Quantization Aware Training, Adaptive Bit-Width Optimization, Minifloat, Ship Detection, Semantic Segmentation, FPGA

**Abstract:** Deep learning is starting to be successfully applied to various space applications. However, due to its memory footprint and computational intensity, inference of many models is still mainly performed on ground platforms. Recent research has focused on neural network quantization to mitigate this computational burden and facilitate on-board processing. This thesis investigates neural network quantization and its impact on model accuracy targeting FPGA accelerators for on-board space applications. We introduce an optimization-based method for mixed-precision uniform quantization of both weights and activations. Its defining characteristic is the use of relaxed fractional bit-widths that are updated using a gradient descent rule but otherwise discretized for all operations (in forward and backward pass computations). We also present an efficient approach based on low-precision floating-point quantization. Our method adapts a quantized deep neural network training approach predominantly used for integer/fixed-point-based quantization. Finally, we studied the impact of long skip connections in U-shaped semantic segmentation models on accuracy and highlighted their memory footprint.