Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Suport Eficient de Processos d'Inferència en Plataformes Heterogènies

Treball fi de grau

Grau en Enginyeria Informàtica

Autor: Alejandro Iznardo Ruiz

Tutors: Pedro Juan López Rodríguez
Carles Hernández Luz

Curs 2020-2021

Resum

La intel·ligència artificial i les seues variants estan evolucionant ràpidament i millorant molts aspectes de la vida humana, tant en aplicacions industrials com en usos quotidians. Per aquest motiu, molts investigadors busquen formes d'optimitzar els costosos processos que componen aquests algoritmes, com són l'entrenament i la inferència.

El motiu d'aquest treball és explorar les diferents alternatives de maquinari en la inferència, amb l'ús de programari que implementa optimitzacions de les xarxes neuronals per tenir un procés més eficient. S'estudiara com afecta cada alternativa al resultat final, fent èmfasi en les principals característiques de la inferència. A més, s'implementaran noves funcionalitats en una plataforma d'entrenament i inferència, amb l'objectiu d'estudiar com afecten diferents configuracions a aquests algoritmes.

Paraules clau: FPGA, GPU, Xilinx, Nvidia, Xarxes Neuronals, Inferència, Funcions d'activació

Resumen

La inteligencia artificial y sus variantes están evolucionando rápidamente y mejorando muchos aspectos de la vida humana, tanto en aplicaciones industriales como en usos cuotidianos. Por este motivo, muchos investigadores buscan maneras de optimizar los costosos procesos que forman estos algoritmos, como el entrenamiento y la inferencia.

El motivo de este trabajo es explorar las diferentes alternativas de hardware en la inferencia, con el uso de herramientas de software que implementan optimizaciones de las redes neuronales para obtener un proceso más eficiente. Se estudiará cómo afecta cada alternativa al resultado final, haciendo énfasis en las principales características de la inferencia. Además, se implementarán nuevas funcionalidades sobre una plataforma de entrenamiento e inferencia, con el objetivo de estudiar cómo afectan las diferentes configuraciones a estos algoritmos.

Palabras clave: FPGA, GPU, Xilinx, Nvidia, Redes Neuronales, Inferencia, Funciones de Activación

Abstract

Artificial intelligence and its derivatives are rapidly evolving and improving a lot of key areas in human life, both in industrial applications and in everyday uses. For this reason, many researchers are looking for ways to optimize the expensive processes that make up these algorithms, such as training and inference.

The reason for this work is to explore some of the different hardware alternatives in inference, and using software tools that implement neural network optimization in order to obtain a more efficient process. It will be studied how each alternative affects the final result, emphasizing the main features of the inference. In addition, new functionalities will be implemented on a training and inference platform, in order to study how some different configurations affect these algorithms.

Key words: FPGA, GPU, Xilinx, Nvidia, Neural Networks, Inference, Activation Functions

Índex

Índex v
Índex de figures vii
Índex de taules viii

1 Introducció 1
1.1 Motivació . . . 1
1.2 Objectius . . . 2
1.3 Context de treball . . . 3
1.4 Objectius de Desenvolupament Sostenible (ODS) . . . 4
1.5 Estructura de la memòria . . . 4
2 Xarxes Neuronals Artificials 7
2.1 Història . . . 7
2.2 Xarxes Neuronals . . . 8
2.2.1 Funcions d'activació . . . 9
2.2.2 Entrenament . . . 10
2.2.3 Capes principals . . . 11
2.2.4 Inferència . . . 14
2.2.5 Quantització en inferència . . . 14
3 Suport del procés d'inferència en FPGA 17
3.1 FPGA . . . 17
3.1.1 Arquitectura de la FPGA . . . 18
3.1.2 Targeta acceleradora Alveo U200 . . . 19
3.2 Vitis AI . . . 20
3.2.1 Instal·lació . . . 21
3.2.2 Procés de quantització . . . 22
3.2.3 Procés de compilació . . . 26
3.2.4 Desplegament a la FPGA . . . 28
4 Suport del procés d'inferència en GPU 31
4.1 GPU . . . 31
4.1.1 Arquitectura de la GPU . . . 32
4.1.2 Model d'execució CUDA . . . 33
4.2 TensorRT . . . 34
4.2.1 Instal·lació . . . 34
4.2.2 Procés d'optimització . . . 35
4.2.3 Desplegament en GPU . . . 38
5 Implementació de noves funcionalitats en HELENNA 41
5.1 Descripció de la plataforma . . . 41
5.2 Funcions d'activació . . . 43
5.2.1 Suport a noves funcions d'activació en HELENNA . . . 43
6 Avaluació dels resultats 49
6.1 Resultats amb Vitis AI . . . 50
6.2 Resultats amb TensorRT . . . 54

6.3 Impacte de les funcions d'activació . . . 57
6.4 FPGA vs. GPU . . . 59

7 Conclusions **63**
7.1 Relació amb els estudis cursats . . . 63
7.2 Treball futur . . . 64
7.2.1 Quantificar l'ús de memòria en les ferramentes . . . 64
7.2.2 Procés de quantització en HELENNA . . . 64

Bibliografia **65**

Apèndixs

A Funcions i codi **69**
A.1 Vitis AI i TensorRT . . . 69
A.1.1 Preprocessar imatges: `input_fn.py` . . . 69
A.1.2 Inferència amb Vitis AI, codi de Xilinx modificat: `inferenceFPGA.py` 70
A.1.3 Crear motor d'inferència de TensorRT: `buildEngine.py` . . . 73
A.1.4 Inferència de 1 image TensorRT: `inferenceGPU.py` . . . 75
A.2 Funcions d'activació . . . 77
A.2.1 ReLU . . . 77
A.2.2 Leaky ReLU . . . 77
A.2.3 Parametric ReLU . . . 78
A.2.4 ELU . . . 78
A.2.5 Swish . . . 79
A.2.6 GELU . . . 79

B Topologies de xarxes neuronals **81**
B.1 Resnet . . . 81
B.2 Inception . . . 81
B.3 VGG . . . 81

Índex de figures

1.1 Comparació de GPU NVIDIA amb CPU Intel, milions d'operacions de FLOPS, productes amb eixida des de 2004 a 2016. 2

2.1 Diagrama de funcionament de l'algoritme perceptró, amb 5 entrades i 5 neurones. 8
2.2 Representació de la funció sigmoide i la seua derivada. 10
2.3 Processos d'entrenament i inferència en aprenentatge profund. 11
2.4 Procés de convolució amb un filtre 3*x*3 i un *stride* d'1 sobre un tensor de dues dimensions 5*x*5. 12
2.5 Procés de *Max pooling* amb un filtre 2*x*2 i un *stride* de 2 sobre un tensor de dues dimensions 4*x*4. 13
2.6 Escalat dels pesos per a la representació en INT4. 15

3.1 Estructura d'una FPGA tradicional. 18
3.2 Concepte de *Fast Tracks* sobre una estructura de FPGA tradicional, on les rutes de connexió no cal que siguen amb els blocs adjacents. 20
3.3 Diferències entre màquines virtuals i tecnologia de contenidors, on cada màquina virtual té el seu propi sistema operatiu mentre que els contenidors es limiten a aïllar recursos sobre el sistema operatiu amfitrió. 21
3.4 Flux a seguir per quantitzar un model en precisió FP32 en el Vitis AI *quantizer*. 23
3.5 Execució de l'aplicació Netron sobre un model Resnet50 dissenyat amb TensorFlow. En l'esquerra, primeres capes de la xarxa neuronal. En la dreta, últimes capes del model. 25
3.6 Flux a seguir en models de xarxes neuronals en Vitis AI, on es quantitza i compila la xarxa per generar instruccions DPU. 26
3.7 Arquitectura del DPUCADX8G. 27

4.1 Arquitectura bàsica de la GPU i dels Multiprocessadors de *Streaming*. . . . 33
4.2 Flux de funcionament de l'arquitectura CUDA. 34
4.3 Mida en bits de les precisions FP32, FP16 i INT8. 36
4.4 Flux d'optimització de models entrenats en TensorRT, on a partir del model entrenat, es crea un motor d'inferència amb un *Builder*, i en temps d'execució processa els tensors d'entrada per a produir els tensors d'eixida. 37

5.1 A l'esquerra: funció ReLU i la seua derivada. A la dreta, funció Leaky ReLU i la seua derivada. 45
5.2 A l'esquerra: funció ELU i la seua derivada amb $\alpha = 0.1$. A la dreta, funció ELU i la seua derivada amb $\alpha = 0.5$. 47
5.3 A l'esquerra: funció Swish i la seua derivada. A la dreta, funció GELU i la seua derivada. 48

6.1 Concepte d'aprenentatge residual en les xarxes `Resnet` 49
6.2 Mòdul *inception* amb reduccions de dimensió. 50

6.3 Diferència de top-1 i top-5 en precisió FP32 i INT8, en 3 dels models utilitzats en CPU i FPGA. 51
6.4 Latència de 4 models en la inferència d'una imatge durant 5 execucions, en FPGA i CPU. 53
6.5 Rendiment de 3 models executats en la FPGA amb diferents dimensions de lot. 53
6.6 Diferència de top-1 i top-5 en precisió FP32 i INT8, en 3 dels models utilitzats en CPU i GPU. 55
6.7 Rendiment de 3 models executats en la GPU amb diferents dimensions de lot. 57
6.8 Evolució de la precisió en l'entrenament del model `VGG16` amb el dataset CIFAR 10 en les 6 funcions d'activació estudiades. 58
6.9 Evolució de la precisió en l'entrenament del model `AlexNet` amb el dataset ImageNet en 3 de les funcions d'activació estudiades. 59
6.10 Diferència de latència en la inferència dels 6 models amb Vitis AI i TensorRT. 60
6.11 Diferència de rendiment en la inferència de 5000 imatges en els 6 models, amb Vitis AI i TensorRT. 61

B.1 Xarxa `Resnet50` 82
B.2 Xarxa `InceptionV1` 83
B.3 Xarxa `VGG16` 84

Índex de taules

6.1 Diferència de la precisió top-1 i top-5 en inferència de 5000 imatges amb una CPU utilitzant TensorFlow amb precisió FP32 i una FPGA utilitzant Vitis AI amb precisió reduïda INT8, en 6 models de xarxes neuronals diferents. 52
6.2 Latència i *speedup* dels models en la inferència d'una imatge, utltizant la FPGA amb Vitis AI o la CPU amb TensorFlow. 52
6.3 Temps d'execució dels models en la inferència de 5000 imatges, utltizant la FPGA amb Vitis AI o la CPU amb TensorFlow. 54
6.4 Diferència de la precisió top-1 i top-5 en inferència de 5000 imatges amb una CPU utilitzant TensorFlow amb precisió FP32 i una GPU utilitzant TensorRT amb precisió reduïda INT8, en 6 models de xarxes neuronals diferents. 56
6.5 Latència i *speedup* dels models en la inferència d'una imatge, utltizant la GPU amb TensorRT o la CPU amb TensorFlow. La primera fila de *speedup* representa CPU vs GPU amb TensorRT utilitzant precisió de 32 bits i la segona fila de *speedup* representa CPU vs GPU amb TensorRT utilitzant precisió de 8 bits. 56
6.6 Temps d'execució de la inferència de 5000 imatges en els models, utltizant la CPU amb TensorFlow, o la GPU, amb TensorRT i precisió de 8 bits. 57
6.7 Precisió final en la inferència de les 6 funcions d'activació en el model `VGG16` 58
6.8 Precisió top-1 i top-5 dels sis models estudiats en la inferència de 5000 imatges, utilitzant Vitis AI i TensorRT. 59

CAPÍTOL 1
Introducció

En aquest primer capítol expliquem la situació actual de les xarxes neuronals i les plataformes heterogènies, així com els motius que han portat a fer aquest treball, els objectius que volem assolir, una explicació del context en què ha sigut elaborat el projecte i, una breu descripció dels capítols que constitueixen la memòria del treball realitzat.

1.1 Motivació

En els últims anys la intel·ligència artificial (IA) està prenent un pes molt important en la vida de qualsevol persona, ja estiga relacionada directament o indirectament amb la informàtica. Aquests algoritmes poden anar des de reconeixement del llenguatge natural, conducció autònoma o classificació d'imatges entre altres moltes aplicacions.

Amb l'evolució d'aquesta ciència, s'ha vist que els dispositius de maquinari tradicionals com la unitat central de processament (CPU), limiten molt l'evolució d'aquests algoritmes. Els principals processos, coneguts com a entrenament i inferència són molt costosos, sobretot l'entrenament. L'entrenament d'una xarxa neuronal profunda de classificació pot arribar a durar dies o fins i tot setmanes.

Així doncs, entren en joc les noves arquitectures de maquinari i la computació heterogènia. Les arquitectures diferents de la CPU, com la unitat de processament gràfic (GPU), ofereixen major còmput d'operacions de coma flotant per segon (FLOPs). En la Figura 1.1 s'aprecia la diferència de potència de còmput d'aquestes dues arquitectures.

És a dir, la computació heterogènia és una gran aliada de la IA i les xarxes neuronals. L'ús de diferents tipus de processadors i el paral·lelisme entre aquests, optimitzen els temps d'entrenament de les costoses xarxes neuronals. No obstant això, el procés d'inferència és igual d'important que l'entrenament d'una xarxa, ja que és l'etapa on el model s'utilitza en aplicacions reals com poden ser conducció autònoma o processos industrials.

La necessitat d'optimitzar aquest procés és un repte hui en dia. Els investigadors busquen formes de reduir la latència d'execució de la inferència de xarxes neuronals, sense augmentar el consum energètic o les prestacions significativament. Els fabricants de maquinari estan desenvolupant solucions de programari per a utilitzar en els seus dispositius, que milloren aquest procés, amb l'objectiu d'aconseguir una inferència eficient, precisa, poc ús de memòria, amb baixa latència i amb un rendiment elevat.

Actualment, la GPU té molta popularitat en el món de la IA i l'aprenentatge profund, però existeixen altres arquitectures com la matriu de portes programable in situ (FPGA, sigles de *Field-Programmable Gate Array*) que són grans alternatives a l'hora de dissenyar aplicacions d'inferència de xarxes neuronals [18]. L'exploració de les diferents arquitec-

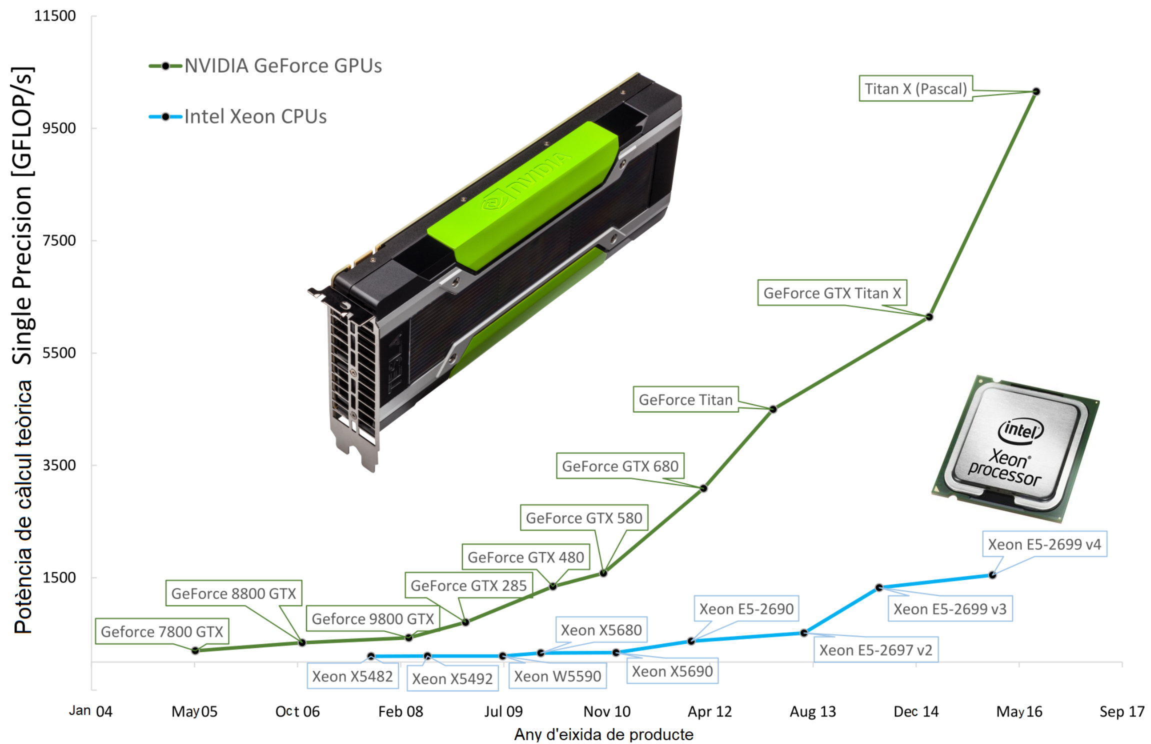

Figura 1.1: Comparació de GPU NVIDIA amb CPU Intel, milions d'operacions de FLOPS, productes amb eixida des de 2004 a 2016.

tures existents i del seu comportament amb aquests algoritmes, convida a la investigació i experimentació amb aquest maquinari.

1.2 Objectius

Aquest treball busca investigar sobre les diferents plataformes heterogènies òptimes per a la creació i execució de xarxes neuronals, així com experimentar amb les principals ferramentes dels fabricants més potents de la indústria del maquinari, ofereixen per optimitzar les xarxes neuronals artificials a les seues arquitectures hardware. No solament buscarem optimitzacions en l'àmbit de maquinari, també indagarem en possibles millores en modificar algunes capes de les xarxes neuronals artificials en diferents models.

Llavors, els objectius del treball són explorar les diferents opcions d'optimització de la inferència de les xarxes neuronals en plataformes heterogènies, així com desplegar diferents topologies de xarxes utilitzant ferramentes de suport i, per últim, implementar noves funcionalitats en una ferramenta d'entrenament i inferència de xarxes neuronals artificials.

Cal fer un esment específic que aquest projecte busca entendre les diferents virtuts i mancances de les solucions actuals, per tant, s'analitzaran les arquitectures utilitzades, fent èmfasi en les principals característiques que afecten directament al procés d'inferència de les xarxes neuronals artificials. A més, es busca aprofundir també en l'arquitectura de les xarxes neuronals artificials i experimentar amb modificacions que puguen optimitzar i millorar la utilització d'aquestes.

Això no obstant, la naturalesa d'aquest treball ve amb limitacions per part del maquinari necessari per a les ferramentes d'optimització, sols disposem d'una FPGA Xilinx Alveo U200 i una GPU NVIDIA GeForce GTX 1050. Amb aquestes dues arquitectures

heterogènies podrem aconseguir els objectius d'aquest projecte. Els objectius queden resumits en:

- Experimentar i realitzar inferència de models de xarxes neuronals en targetes FPGA amb la ferramenta de Xilinx, Vitis AI.
- Implementar i executar inferència de models de xarxes neuronals en targetes GPU amb la llibreria de NVIDIA, TensorRT.
- Donar suport a noves funcions de xarxes neuronals en la ferramenta d'entrenament i inferència de xarxes neuronals, HELENNA.
- Implementació i experimentació de les noves funcions implementades en diferents plataformes heterogènies.

1.3 Context de treball

Aquest treball de final de grau ha estat desenvolupat durant les pràctiques realitzades al Grup d'Arquitectures Paral·leles (GAP) del Departament d'Informàtica de Sistemes i Computadors (DISCA) [8]. El GAP és un grup d'investigació de la Universitat Politècnica de València, té una llarga història treballant amb xarxes d'interconnexió de computació paral·lela, així com en microarquitectures i memòria de computadors. Aquest grup fa més de trenta anys que està activa en el món de la informàtica i ha participat en nombrosos projectes d'investigació de la Comissió Europea (CE), projectes nacionals i projectes autonòmics. També ha col·laborat amb altres grups d'investigació de diferents universitats europees.

El GAP compta amb multitud de membres amb gran experiència en el sector acadèmic i de la investigació, grans projectes completats i en desenvolupament i té accés a grans equips de maquinari per a la investigació, des de nodes amb l'última tecnologia de targetes gràfiques fins a clústers de més de 30 nodes amb capacitats de còmput molt elevades.

En els últims anys, el GAP també s'ha endinsat en la investigació d'intel·ligència artificial i les seues derivades, com són l'aprenentatge automàtic, l'aprenentatge profund i les xarxes neuronals artificials. Entre altres coses que han realitzat en aquest camp tan actual, han desenvolupat una ferramenta capaç de definir, entrenar i fer inferència en models de xarxes neuronals, anomenada HELENNA *(HEterogeneous LEarning Neural Network Application)*.

Durant el curs hem realitzat les pràctiques d'empresa a aquest grup d'investigació, realitzant tasques relacionades amb la intel·ligència artificial, les plataformes heterogènies i els models de xarxes neuronals artificials. Hem estat 6 alumnes treballant per al GAP, on tots hem fet el treball de fi de grau, investigant diferents aspectes de la computació heterogènia i les xarxes neuronals; treballant amb les diferents ferramentes que han desenvolupat en l'organització, especialment en HELENNA i, utilitzant el maquinari al qual té accés el grup, on se'ns ha facilitat l'ús de computadors amb molta potència de càlcul com per exemple un node amb un processador de 12 nuclis, 64 GB de ram i una targeta acceleradora FPGA Alveo U200 o un clúster amb 6 Jetson Nano i 3 Jetson Xavier.

Consegüentment, les pràctiques d'empresa i elaboració del treball de final de grau amb el GAP han estat molt fructíferes i s'ha contribuït a crear un ambient de treball molt agradable, amb reunions setmanals on compartíem el treball realitzat durant la setmana i l'evolució dels objectius establerts per cadascú.

1.4 Objectius de Desenvolupament Sostenible (ODS)

Les Nacions Unides (ONU) han establert uns objectius per transformar el món i aconseguir un futur sostenible per a tots. Aquests 17 objectius van des d'erradicar la pobresa, passant garantir educació de qualitat per a tothom i fins a reduir les desigualtats entre països. Van ser aprovats el 2015 per tots els estats membres de l'ONU i l'objectiu és haver complit aquests objectius per al 2030.

Des de la Universitat Politècnica de València es promou que aquests objectius es compleixen als seus campus i, en aquest treball s'ha col·laborat per complir els que estaven dins de les nostres possibilitats:

- ODS 9: Fomentar la innovació. Durant el transcurs de la investigació i el treball s'ha treballat amb una de les tecnologies més innovadores dels últims anys com és la intel·ligència artificial i l'aprenentatge automàtic i s'han buscat formes d'innovar i fomentar l'ús de tècniques que milloren aquestes ciències.

- ODS 12: garantir producció i consum sostenibles. A partir de l'ús de plataformes heterogènies es busca optimitzar el consum d'electricitat alt que tenen les xarxes neuronals. La gran quantitat de càlculs que realitzen aquests models amb milions de paràmetres, són optimitzables quan s'utilitza un maquinari adequat per a l'execució d'aquestes. Així podrem garantir un consum energètic sostenible.

1.5 Estructura de la memòria

Aquest treball de final de grau es compon de 7 capítols:

- **Introducció**: En aquest apartat expliquem la motivació requerida per fer aquest treball, els objectius establerts i el context en què s'ha fet aquest treball. També s'hi han inclòs els ODS que ha intentat complir aquest treball.

- **Xarxes Neuronals Artificials**: Comentarem la història recent de les xarxes neuronals, així com el funcionament i característiques bàsiques dels seus processos.

- **Suport del procés d'inferència en FPGA**: Estudiarem les característiques principals de l'arquitectura FPGA i utilitzarem un programari per a desplegar processos d'inferència sobre aquesta plataforma. Es detallarà el procés seguit per optimitzar les xarxes neuronals amb la ferramenta.

- **Suport del procés d'inferència en GPU**: Veurem l'arquitectura de la GPU i el seu model d'execució i utilitzarem una llibreria que dóna suport al procés d'inferència en targetes GPU. Explicarem el procés a seguir per aconseguir inferir en xarxes neuronals.

- **Implementació de noves funcionalitats en HELENNA**: Descriurem el treball realitzat sobre aquesta plataforma d'entrenament i inferència, i detallarem les implementacions afegides i com afecten el rendiment de les xarxes neuronals.

- **Avaluació dels resultats**: Analitzarem els resultats obtinguts en l'execució d'inferència i entrenament en models de xarxes neuronals. Compararem les alternatives i maquinaris utilitzats i veurem la influència de cada funcionalitat en aquests algoritmes.

- **Conclusions**: Acabarem exposant l'aconseguit durant el projecte. A més, explicarem la relació del treball amb els estudis cursats i s'exposaran millores a realitzar en el treball futur.

CAPÍTOL 2

Xarxes Neuronals Artificials

A continuació introduirem diversos conceptes de les xarxes neuronals artificials, així com de l'aprenentatge automàtic i l'aprenentatge profund. Es detallara l'estructura de les xarxes neuronals artificials, fent èmfasi en les xarxes neuronals convolucionals. Observarem el funcionament de les diferents capes i les funcions d'activació en aquests models. També es comentarà la història i la situació actual de la intel·ligència artificial.

2.1 Història

El britànic Alan Turing va ser un dels precursors de la IA, quan en 1950 va proposar l'ara conegut com el test de Turing [30]. En ell, es planteja un mètode per determinar si una màquina o computadora és capaç de pensar, és a dir, és intel·ligent. L'experiment consistia en una interacció entre una màquina i un interlocutor humà on, l'interlocutor no sap si està interaccionant amb una màquina o una persona. La màquina respon a preguntes escrites que realitza la persona i, si les contestacions que realitza fan creure a l'interlocutor que és una persona, se li considerava intel·ligent i capaç de pensar. Amb el naixement d'aquest concepte, comença el que molts consideren l'inici de la intel·ligència artificial.

Per altra banda, les xarxes neuronals ja havien aparegut el 1943, quan un lògic i un neurocientífic nord-americans estableixen el primer model matemàtic d'una xarxa neuronal [25]. Amb aquest model defineixen el comportament de les funcions cerebrals i la importància de les neurones en aquests models, intentant replicar el funcionament d'un cervell humà.

Però no seria fins a 1957 quan s'obtindria una aproximació a una xarxa neuronal artificial amb la creació del model perceptró, creat pel psicòleg nord-americà Frank Rosenblatt [22]. Rosenblatt va definir un model que suma els valors d'entrada d'acord amb uns pesos sinàptics i introdueix el sumatori en una funció d'activació que dóna el resultat. En la Figura 2.1 es pot veure el funcionament d'aquest algoritme. El perceptró regula els seus pesos en funció del resultat obtingut i ho fa en cada iteració. Amb aquest algoritme naix la xarxa neuronal més bàsica que existeix.

Una volta existia un model de xarxa neuronal vàlid, la IA va fer un gran salt amb l'algoritme de *backpropagation* o retropropagació. Va ser descobert en 1974, però es va popularitzar en 1986 amb la publicació d'un famós article que va canviar la IA [6]. Consisteix en un algoritme capaç de calcular l'error que té l'eixida d'una xarxa neuronal i propagar-lo per totes les capes, començant per l'última, fent les correccions més importants en les neurones que més pes hagen tingut en el resultat final. D'aquesta manera, les xarxes neuronals són capaces d'aprendre per si mateixa, ja que solament amb les dades

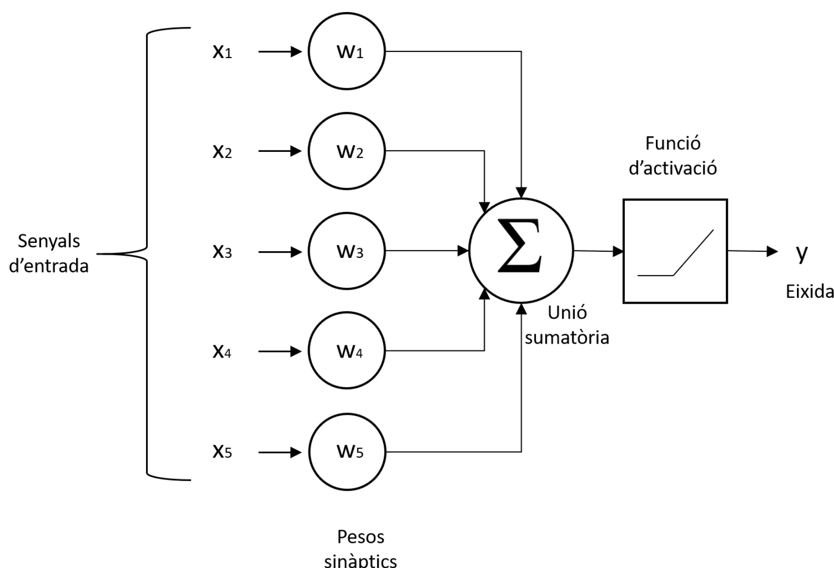

Figura 2.1: Diagrama de funcionament de l'algoritme perceptró, amb 5 entrades i 5 neurones.

d'entrada els paràmetres de la xarxa es van regulant utilitzant la retropropagació. Conseqüentment, apareixen xarxes neuronals complexes, com el perceptró multicapa, que consten de múltiples capes ocultes que van ajustant els seus pesos sinàptics amb l'algoritme de retropropagació.

A continuació, la intel·ligència artificial i les xarxes neuronals artificials van fer grans passos per aproximar-se on estem actualment. Apareixen els termes aprenentatge automàtic (*machine learning*) i aprenentatge profund (*deep learning*), per referir-nos a la creació de sistemes que aprenen automàticament a partir de conjunts de dades de gran volum. Però la popularització de l'aprenentatge profund va vindre en 2008, quan el grup d'aprenentatge automàtic de Stanford, va començar a incentivar l'ús de la GPU en l'entrenament de les xarxes neuronals profundes, ja que accelerava aquest procés significativament respecte a les GPU [20].

La investigació amb xarxes neuronals va rebre un gran impuls amb la incorporació de les unitats de processament gràfic per a l'entrenament, però encara hi havia mancances com l'obtenció de dades per a l'aprenentatge de la xarxa. Va ser aleshores quan, en 2009, una professora de Stanford va publicar la base de dades ImageNet, constituïda per més de 14 milions d'imatges etiquetades per a la investigació en aprenentatge automàtic i profund [24].

Tot això i fins a l'actualitat, la intel·ligència artificial i l'aprenentatge profund han anat prenent un pes molt important en el món de la informàtica. Ferramentes de desenvolupament com Tensorflow, Keras o Caffe, faciliten la creació de nous models de xarxes neuronals i promouen la investigació en aquest camp, on queda molt per recórrer [13].

2.2 Xarxes Neuronals

Les xarxes neuronals artificials són un algoritme de l'aprenentatge automàtic i de l'aprenentatge profund. Són una representació abstracta del cervell humà, on s'intenta emular el comportament d'aquest. Les xarxes neuronals estan formades per una gran quantitat de neurones artificials, que col·laboren entre elles per produir un resultat.

Una neurona artificial realitza una part del còmput de la xarxa neuronal. Té com a entrades les eixides d'altres neurones, associades a un pes. Els pesos de les neurones

indiquen la importància que té cada neurona respecte a la qual està connectada. Per exemple, si tenim unes neurones A i B connectades a una neurona C, on la connexió $A - B$ té major pes que la $A - C$, tenim que la neurona A té més importància en C que B. Les neurones artificials també contenen una funció d'activació i un terme biaix o *bias*, que actua com a valor llindar. La funció d'activació s'encarrega de processar l'eixida de la neurona i normalitzar el valor mitjançant una funció matemàtica. Per exemple, una funció d'activació binària s'activarà si el resultat de la neurona és major que 0 i no s'activarà en cas contrari[17].

El funcionament de les neurones artificials per produir una eixida consisteix a sumar les entrades rebudes i el biaix, que són les neurones computades pels seus pesos, i la suma es processa en una funció d'activació, que regula el valor d'eixida i es propaga a les següents neurones. El còmput d'una neurona el podem veure en l'equació 2.1, on el resultat serà processat per una funció d'activació.

$$z = b + \sum_i x_i w_i \tag{2.1}$$

Les neurones estan organitzades en diferents capes. Les neurones d'una capa es connecten a les de la capa següent i estan connectades a les de la capa anterior. La primera capa es coneix com la capa d'entrada, que rep les dades externes que ha de processar el model. L'última capa és la capa d'eixida i s'encarrega de calcular el resultat final. Entre aquestes estan les capes ocultes, que fan operacions matemàtiques en les dades per processar-les i decidir el resultat final.

L'objectiu de la xarxa neuronal és aprendre a realitzar certa funció a partir d'un conjunt de dades etiquetat, on sabem el resultat que ha de donar cada dada. Per açò, s'ha de seguir un procés d'entrenament, on s'aprendran els valors dels paràmetres de la xarxa (pesos i biaixos). En l'aprenentatge profund aquest entrenament s'aconsegueix en els processos de *forwardpropagation* i *backpropagation*.

En la *forwardpropagation* o propagació cap endavant, les dades avancen des de la capa d'entrada fins a la capa d'eixida recorrent tota la xarxa neuronal. Cada capa rep les dades, les processa junt amb la funció d'activació i les passa a la capa següent. Quan totes les neurones hagen realitzat els seus càlculs, s'arribarà a la capa final amb un resultat. Aquest resultat s'utilitzarà per a mesurar el nivell d'exactitud respecte al resultat esperat, mitjançant una funció de cost. L'objectiu és reduir el cost, ja que, a major cost, significa que estem més lluny del resultat esperat. El resultat de la funció de cost s'ha de propagar cap enrere, perquè totes les neurones de la xarxa processen aquesta informació i ajusten els seus paràmetres. Açò s'aconsegueix amb la retropropagació o *backpropagation*, on des de l'última capa, totes les neurones reben aquesta informació. Cada neurona es modifica depenent de la contribució que han fet al resultat final [5].

Cada capa emmagatzema tots els seus pesos en matrius o vectors de gran longitud, però en l'àmbit de les xarxes neuronals i l'aprenentatge profund es coneixen com a tensors.

2.2.1. Funcions d'activació

Les funcions d'activació s'utilitzen per propagar l'eixida de la neurona. L'eixida és rebuda per les neurones de la següent capa i així fins a la capa d'eixida. La linealitat en la propagació de les dades en les xarxes neuronals afecta molt els resultats de cada neurona. Com el resultat de 2.1 pot tenir qualsevol valor, s'ha d'establir alguna forma de normalitzar el resultat del còmput dels pesos. Les funcions d'activació fan aquesta funció, amb l'objectiu d'eliminar la linealitat. A més, influeixen en l'algoritme de retropropagació,

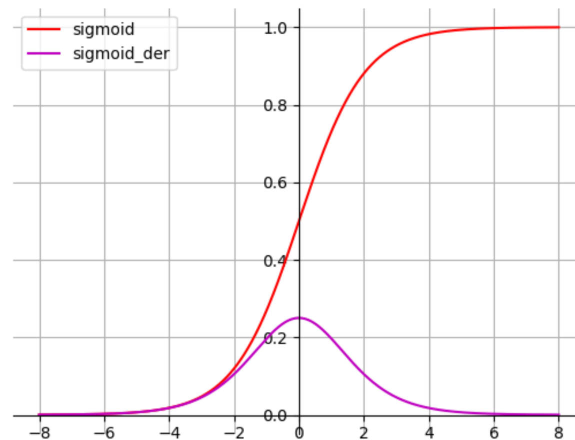

Figura 2.2: Representació de la funció sigmoide i la seua derivada.

ja que la derivada d'aquestes funcions s'utilitza per a ajustar els pesos de les neurones artificials [6].

Un exemple de funció d'activació és la funció sigmoide. Permet transformar valors atípics o molt elevats en dades vàlides per al model. Aquesta funció resulta en eixides entre 0 i 1, però sense un canvi directe de valors, sinó una transició suau en el canvi d'un valor a un altre. Les equacions de 2.2 i 2.3 són la funció i la seua derivada i la representació la podem veure en la figura 2.2.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

$$f'(x) = \frac{dy}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} \tag{2.3}$$

Un altre exemple de les funcions més utilitzades en les xarxes neuronals és la funció *SoftMax*. Aquesta funció rep com argument un vector i el normalitza a una distribució de probabilitat de la grandària del vector rebut, amb probabilitats proporcionals a les exponencials dels nombres d'entrada. Per aquest motiu, és utilitzada com a capa final dels models classificadors. En l'equació 2.4 podem veure la seua representació.

$$f_i(x) = \frac{e^{x_i}}{\sum e^{x_i}} \tag{2.4}$$

Existeixen moltes més funcions d'activació en el món de l'aprenentatge profund i les xarxes neuronals. Aquestes tenen un impacte directe en les prestacions i en la precisió de les xarxes neuronals, per tant, cal tenir-les molt en compte a l'hora de dissenyar models.

2.2.2. Entrenament

L'entrenament és el procés de crear l'algoritme d'aprenentatge profund perquè realitze una funció concreta, a partir d'un conjunt de dades etiquetat. La inferència és ficar aquest model a funcionar, provant el seu ús en dades sense etiquetar. En la figura 2.3 podem veure aquests dos processos il·lustrats.

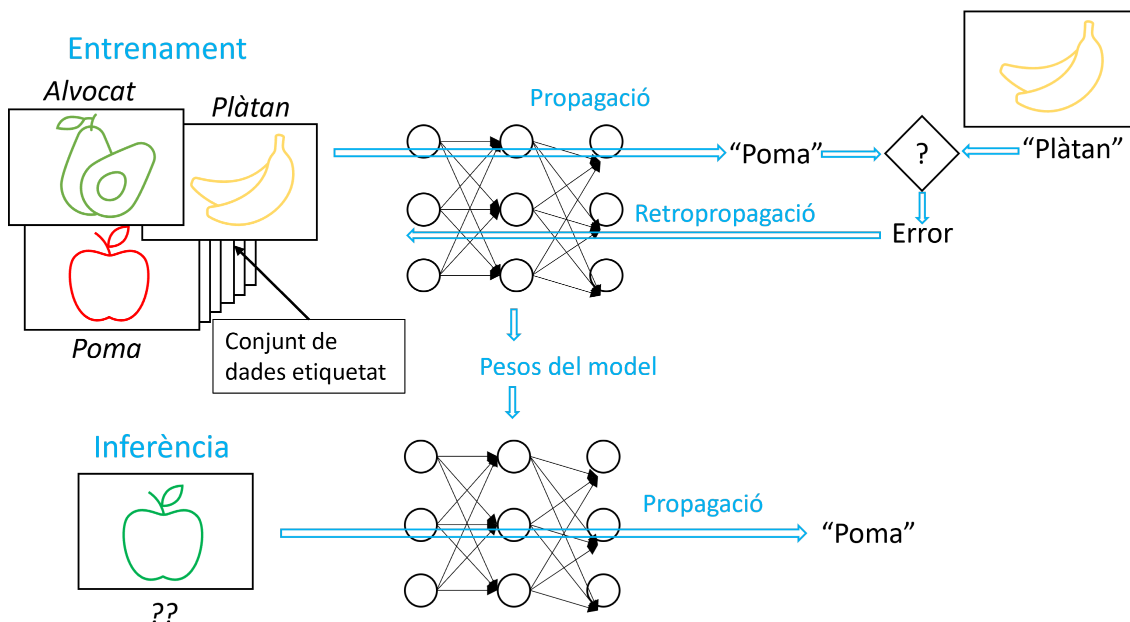

Figura 2.3: Processos d'entrenament i inferència en aprenentatge profund.

Les xarxes neuronals artificials aprenen a partir de processar una gran quantitat de dades etiquetades correctament, conegut com a conjunt de dades d'entrenament. En aquest procés, la xarxa fa prediccions amb el conjunt de dades i, en cas de fallar en aquestes, va corregint els seus paràmetres fins que produeix un resultat desitjat o arriba al seu límit.

En aquesta etapa podem diferenciar diversos termes que defineixen aquest procés. Una mostra és una unitat del conjunt de dades, per exemple, si volem entrenar una xarxa classificadora d'imatges, una mostra és una imatge. Aquestes mostres s'agrupen en lots o *batch*, nombre de mostres menor o igual al total del conjunt de dades que alimenten al model perquè faça les prediccions. Quan totes les mostres han fet el procés de propagació cap endavant, es compara el resultat de cada una amb el resultat esperat i comença el procés de retropropagació, on es corregiran els paràmetres de la xarxa neuronal. Amb els paràmetres de la xarxa neuronal ajustats, es repetirà aquest procés amb el següent lot de mostres, fins que no queden més mostres en el conjunt de dades. Quan s'acaben les mostres, haurà finalitzat una època. El nombre d'èpoques determina el nombre de vegades que un model recorrerà tot el conjunt de dades, on cada vegada es processaran en un ordre distint.

Un major nombre d'èpoques ens assegura que el resultat final del model serà més precís, però també implica un cost major tant temporal, energètic i computacional. El procés d'entrenament és molt costós, on es depén molt del maquinari i les prestacions d'aquest. Per exemple, la base de dades ImageNet té més de 14 milions d'imatges etiquetades, que poden ser utilitzades per a l'entrenament de models d'aprenentatge profund [24].

2.2.3. Capes principals

Les capes que formen les neurones artificials fan diferents funcionalitats per al processament de les dades i l'entrenament. A continuació es detallen algunes de les més utilitzades i que formen part dels models utilitzats en aquest projecte.

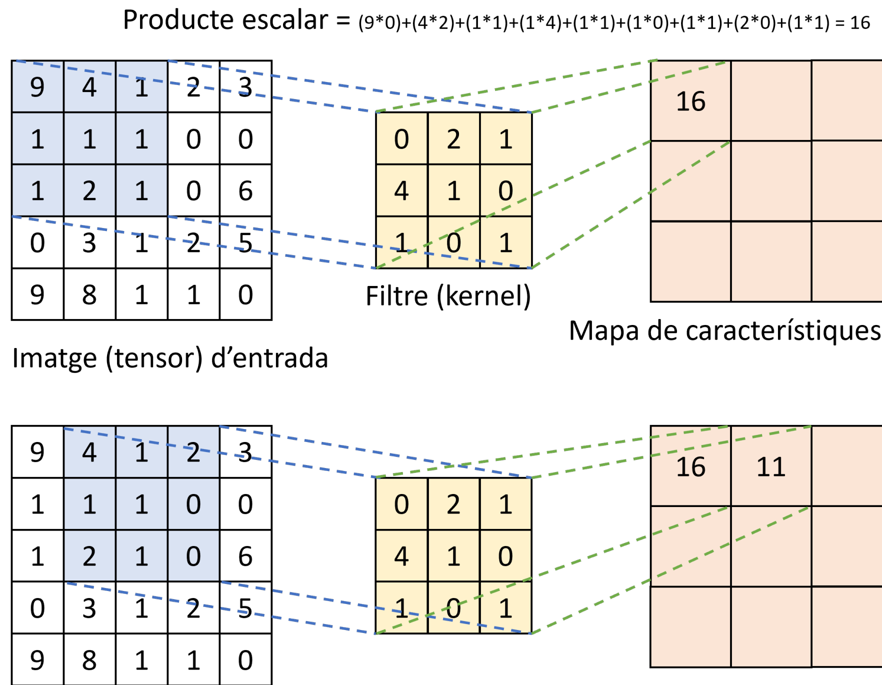

Figura 2.4: Procés de convolució amb un filtre $3x3$ i un *stride* d'1 sobre un tensor de dues dimensions $5x5$.

Capa Convolucional

La capa convolucional és la principal en les xarxes neuronals convolucionals, molt utilitzades en classificació d'imatges, detecció d'objectes, anàlisis d'imatges mèdiques i més. Aquestes xarxes neuronals extrauen les característiques de les dades d'entrada, redueixen les dimensions de les dades fins que aconsegueixen tenir neurones simples sobre les quals realitzar la predicció del model. L'entrada d'aquestes capes és un tensor amb la forma: $(nombred'entrades)x(alturadel'entrada)x(ampladadel'entrada)x(canalsdel'entrada)$. Per exemple, si la primera mostra que passem a la xarxa és una imatge de 224 píxels d'amplada i alçada i els 3 canals de color (RGB), el tensor d'entrada serà $1x224x224x3$.

L'objectiu de la capa convolucional és extraure les característiques principals del tensor al mateix temps que es redueix la seua mida. Açò s'aconsegueix amb l'aplicació d'un filtre més menut que les dades d'entrada. Aquest filtre o *kernel* té unes dimensions NxN i uns valors que anomenarem pesos, i va recorrent el tensor i fent l'operació de producte escalar per obtenir un tensor conegut com a mapa de característiques. El filtre recorre el tensor desplaçant-se un cert nombre de posicions anomenat *stride*.

D'una forma breu, tenim una dada d'entrada com pot ser una imatge representada en píxels, i tenim un filtre, amb uns pesos que s'aplica sistemàticament a les dades d'entrada per a crear un mapa de característiques En la figura 2.4 podem veure el funcionament de les convolucions en un tensor de dues dimensions.

Capa Maxpooling

Les capes de *pooling* o d'agrupació soles anar després de les capes convolució en les xarxes neuronals convolucionals per a poder processar millor les eixides d'aquestes capes. La idea principal és reduir la dimensió del tensor d'entrada amb una tècnica coneguda com a *down sampling*, on a partir de la dada original, es crea una dada amb menys resolució, però on es conserven les característiques principals. La idea de *pooling* és pareguda

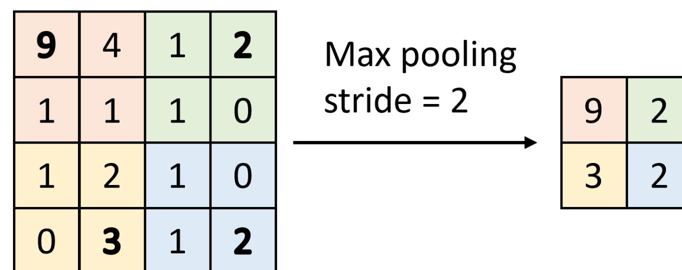

Figura 2.5: Procés de *Max pooling* amb un filtre $2x2$ i un *stride* de 2 sobre un tensor de dues dimensions $4x4$.

la convolució, on un filtre NxN recorre tota l'entrada, però la diferència és que aquest filtre no té cap paràmetre o pesos. Aquest filtre fa una operació com la mitjana aritmètica o obtenir el màxim, on l'eixida computada passa a formar part del tensor redimensionat. En la figura 2.5 podem veure el funcionament de la capa *Max pooling* en un tensor $4x4$, amb un *stride* de 2, on es redimensiona a un tensor $2x2$.

Existeixen diversos tipus de *pooling*, on destaquen dos, el *Max pooling* i el *Average pooling*. En el *Max pooling*, el filtre selecciona com a eixida el major dels valors al que s'està aplicant, mentre que en el *Average pooling* es realitza una mitjana aritmètica dels valors seleccionats per produir una aproximació més precisa. En les xarxes neuronals convolucionals predomina l'ús de *Max pooling*.

Altres capes importants

Existeixen moltes més capes en el camp de l'aprenentatge profund, que fan moltes funcionalitats distintes. Una de les més bàsiques i utilitzades és la *Fully Connected*. Aquesta capa té totes les seues entrades connectades a la següent capa, on no es fa cap operació addicional sobre les dades que recorren la xarxa. En els primers sistemes perceptró, totes les capes són *fully connected*, ja que les neurones estan connectades a totes les neurones de la capa següent. Aquesta capa té un cost computacional molt alt i se sol utilitzar molt en xarxes neuronals classificadores, per situar-la entre les últimes capes i classificar les dades entre les classes requerides.

Una altra capa important és la *batch normalization*. Sovint, les xarxes neuronals amb gran quantitat de capes ocultes poden presentar problemes d'inestabilitat. El procés de retropropagació actualitza els pesos de cada capa, des del final fins al principi i, suposant que la resta de paràmetres de la xarxa són fixes. Açò pot derivar en què alguns pesos estan sempre actualitzant-se a un valor que no para de canviar. La solució ve amb una tècnica coneguda com a *batch normalization*, que estandarditza els valors d'entrada de cada lot d'entrenament, amb operacions com escalar o centrar els valors. Açò millora el procés d'aprenentatge del model, ja que redueix el nombre d'èpoques requerides per ser entrenat [14].

L'última capa que introduirem és la *dropout*. Les xarxes neuronals amb molts paràmetres, entrenades amb conjunts de dades relativament menuts poden tenir el problema de *overfitting* o sobre ajustament. Aquest problema ocorre quan els paràmetres de la xarxa neuronal s'ajusten excessivament a les dades d'entrada, on quedarà entrenat per a unes característiques molt específiques i s'empitjora la predicció de noves dades. La funció *dropout* evita el problema del sobre ajustament ignorant algunes neurones aleatòriament, durant el procés d'entrenament. Aquest mètode de regularització evita que les neurones s'ajusten excessivament a les dades d'entrenament [27].

2.2.4. Inferència

El procés d'inferència d'un model de xarxa neuronal consisteix a utilitzar la xarxa neuronal entrenada, a fer la funció que ha aprés com classificació d'imatges, detecció d'objectes, traducció a temps real o qualsevol de la gran quantitat d'aplicacions que té la IA. En la inferència, es processen les dades per a ïnferirün resultat, que serà l'eixida del model com per exemple un resultat numèric. Les dades utilitzades són diferents de les dades amb què s'ha entrenat el model, on podem apreciar la validesa del model i les seues utilitats.

Hi ha 5 factors crítics que s'utilitzen per mesurar la inferència:

- Rendiment. El volum de sortida en un període donat, habitualment mesurat en mostres/segon. És la capacitat que té el programari en processar una quantitat de dades gran en poc de temps, important per a l'escalat i costos en els centres de dades. Per exemple, en classificació d'imatges, el nombre d'imatges processades per segon és una mesura del rendiment.

- Eficiència. És la quantitat de rendiment entregat per unitat de potència o l'energia utilitzada per a l'execució de la xarxa neuronal. Els costos d'electricitat de les xarxes neuronals profundes són un factor molt important en les aplicacions reals dels models, per tant els desenvolupadors busquen formes de reduir el consum energètic d'aquests algoritmes.

- Latència. Temps requerit per a executar una inferència. Una baixa latència és fonamental en aplicacions que necessiten aquesta característica, com és la conducció autònoma amb IA, on la velocitat de resposta pot comprometre la seguretat dels usuaris.

- Precisió. Percentatge d'encert de la resposta correcta en el resultat de la inferència. En operacions com la classificació d'imatges o aplicacions industrials (rebuig de productes defectuosos), es requereix una precisió elevada, ja que uns resultats incorrectes comprometen la fiabilitat del model.

- Ús de la memòria. Els algoritmes d'aprenentatge profund requereixen memòria tant de l'amfitrió com del dispositiu on es despleguen les xarxes. Un model molt complex requerira molta memòria i veurà limitades les seues aplicacions en els recursos de maquinari que els usuaris disposen. És particularment important en sistemes de conducció autònoms amb càmeres, que solen disposar d'una memòria limitada.

Si un model no és capaç d'oferir uns resultats acceptables en aquests cinc aspectes, segurament no és factible utilitzar-lo en aplicacions reals. La recerca de millores al procés d'inferència que optimitzen aquest procés és un dels grans reptes avui en dia.

2.2.5. Quantització en inferència

Les solucions d'inferència actuals requereixen ajustar-se a cada aplicació, però una inferència perfecta seria eficient, precisa, amb baixa latència, molt de rendiment i poc ús de memòria. En els últims anys estan sortint tècniques d'optimització que milloren algun o diversos dels aspectes principals, sense afectar significativament als altres, com és la quantització.

La quantització és una tècnica que consisteix a reduir la dimensió dels pesos i de les activacions en les xarxes neuronals per optimitzar característiques de la inferència, com la

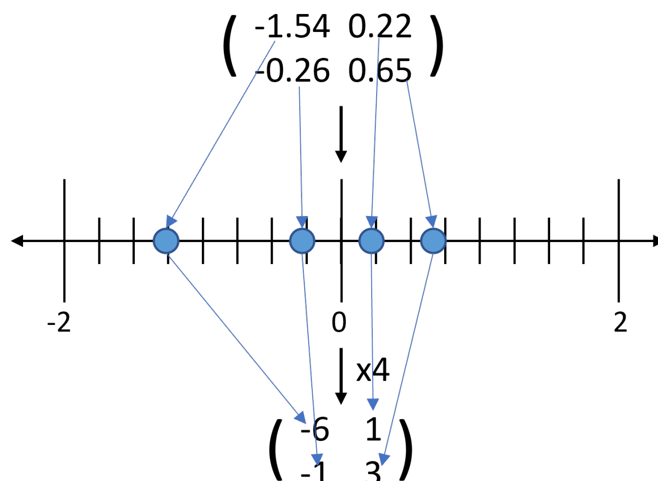

Figura 2.6: Escalat dels pesos per a la representació en INT4.

latència, l'eficiència i l'ús de memòria. Els pesos i les activacions en les xarxes neuronals solen estar representades en coma flotant de simple precisió (FP32), on cada paràmetre ocupa 32 bits de memòria. La quantització busca representar aquests paràmetres en tipus enters que ocupen poc espai, sense coma, com són la representació d'enters de 8 bits (INT8), 4 bits (INT4) o inclús 1 bit (INT1)[15] [31].

El funcionament general d'aquesta tècnica el podem veure amb un exemple, on podrem visualitzar el concepte de quantització. Suposem una xarxa neuronal amb dues neurones i una sola capa, on tindrem un tensor 2x2 representant els pesos, un altre 2x1 representant les activacions dels pesos. El resultat de la xarxa neuronal serà un vector 2x1. Aquests valors en maquinari estarien representats en FP32. Per exemple tenim els següents valors:

$$\begin{pmatrix} -1.54 & 0.22 \\ -0.26 & 0.65 \end{pmatrix} * \begin{pmatrix} 0.35 \\ -0.51 \end{pmatrix} = \begin{pmatrix} -0.651 \\ -0.423 \end{pmatrix}$$

I suposem que volem quantitzar la xarxa neuronal per poder representar els valors amb el format INT4, que soles permet representar valors de -8 a 7 inclosos. Per aconseguir açò cal escalar els valors reals a una aproximació en enters. La idea és situar els valors en un rang de $[-X, X)$, dividit en 16 parts, ja que en INT4 sols es poden representar 16 nombres. Després caldria escalar aquest rang al rang INT4, que és $[-8, 8)$. Per exemple, situarem els pesos en un rang de $[-2, 2)$ i els escalarem a la representació INT4, equivalent a multiplicar aquests valors per 4. En la figura 2.6 podem veure la idea principal d'escalar els valors.

A continuació cal fer el mateix amb les activacions. Un detall important és que l'escalat no cal fer-lo sempre en el mateix rang. En les activacions del nostre exemple podem veure com es poden situar en un rang de $[-1, 1)$, que és equivalent a multiplicar els valors per 8. En l'exemple els valors d'activació $[0.35, -0.51]$ passaran a ser $[3, -4]$ Amb totes les parts quantitzades podem calcular el resultat de la nostra xarxa neuronal quantitzada.

$$\begin{pmatrix} -6 & 1 \\ 1 & 3 \end{pmatrix} * \begin{pmatrix} 3 \\ -4 \end{pmatrix} = \begin{pmatrix} -22 \\ -15 \end{pmatrix}$$

Per obtenir el resultat definitiu, caldrà "desquantitzar"el resultat. El mateix procés que s'ha fet es pot fer a la inversa, on tornarem a la representació FP32 mitjançant una multiplicació.

$$\begin{pmatrix} -22 \\ -15 \end{pmatrix} * \frac{1}{4 * 8} = \begin{pmatrix} -0.688 \\ -0.469 \end{pmatrix}$$

Com podem observar, en aquest exemple els resultats de la quantització són molt similars als primers resultats, calculats amb els paràmetres en FP32. Els resultats enters de l'exemple estan fora del rang de representació d'INT4, així que aquest exemple no podria implementar-se en maquinari. En la majoria de casos de quantització s'utilitza precisió de 8 bits entera, on es poden representar valors en el rang $[-128, 128)$.

Inevitablement, la quantització comporta una pèrdua de precisió. Existeixen tècniques com escalar cada fila (neurona) a un rang diferent, ja que cada fila actua independentment en el resultat, després sols caldrà "desquantitzarçada posició del vector resultat en l'escala que s'haja aplicat.

Avui en dia, companyies com NVIDIA o Xilinx volen explotar el potencial d'aquesta tècnica, i han desenvolupat programari per a poder fer inferència amb xarxes neuronals quantitzades, com TensorRT o Vitis AI [32] [29].

CAPÍTOL 3
Suport del procés d'inferència en FPGA

En aquest capítol explicarem el treball realitzat amb la ferramenta Vitis AI de Xilinx, capaç de donar suport al procés d'inferència de models de xarxes neuronals artificials. També aprofundirem en l'estructura del maquinari utilitzat i la sinergia amb el programari utilitzat.

3.1 FPGA

Les FPGA són dispositius lògics programables, un tipus de circuit integrat capaç d'implementar qualsevol circuit digital. Les FPGA naixen en 1984, quan la companyia estatunidenca Xilinx comercialitza la primera FPGA. Altres empreses com Altera o Lattice no tardarien a entrar a competir amb el seu propi maquinari. Amb els anys, els principals competidors, Xilinx i Altera, van anar imposant-se en el mercat d'aquests dispositius fins a arribar a la situació actual, on Xilinx predomina sobre Intel, empresa que va comprar al fabricant Altera per poder competir en Xilinx en els circuits integrats.

Els circuits integrats lògics es poden dividir en dues categories: els circuits integrats personalitzats, maquinari dissenyat per funcions concretes com poden ser els circuits integrats d'aplicació específica (ASIC) i els dispositius lògics programables, com són les matrius de portes programable in situ (FPGA). Els ASIC ofereixen un major rendiment i més eficiència energètica, en els últims anys empreses com Google estan dissenyant ASIC per aplicacions específiques, com la intel·ligència artificial, amb resultats molt competitius [1]. Per altra part, les FPGA permeten ser reprogramades i complir diferents funcions, a més de tindre costs d'adquisició i de desenvolupament menors. Aquestes característiques fan que les FPGA siguen un dels dispositius més atractius per a accelerar el procés d'inferència de les xarxes neuronals.

Des del seu naixement fins a l'actualitat, la versatilitat d'aquest maquinari l'ha fet aconseguir ocupar un espai notable en molts àmbits de la tecnologia, com són la conducció autònoma [2] o inclús en missions a l'espai exterior [3]. Actualment, amb l'auge d'aplicacions de IA, els fabricants ofereixen dispositius i ferramentes que poden adaptar-se a aplicacions que utilitzen IA i xarxes neuronals artificials.

[1] Cloud TPU: ASIC de Google per a aplicacions d'intel·ligència artificial https://cloud.google.com/tpu

[2] Aplicacions de conducció autònoma amb FPGA d'Intel https://www.intel.es/content/www/es/es/automotive/products/programmable/applications.html

[3] Ús de FPGA reprogramables en aplicacions espacials per l'Agència Espacial Europea (ESA) https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Microelectronics/The_use_of_reprogrammable_FPGAs_in_space

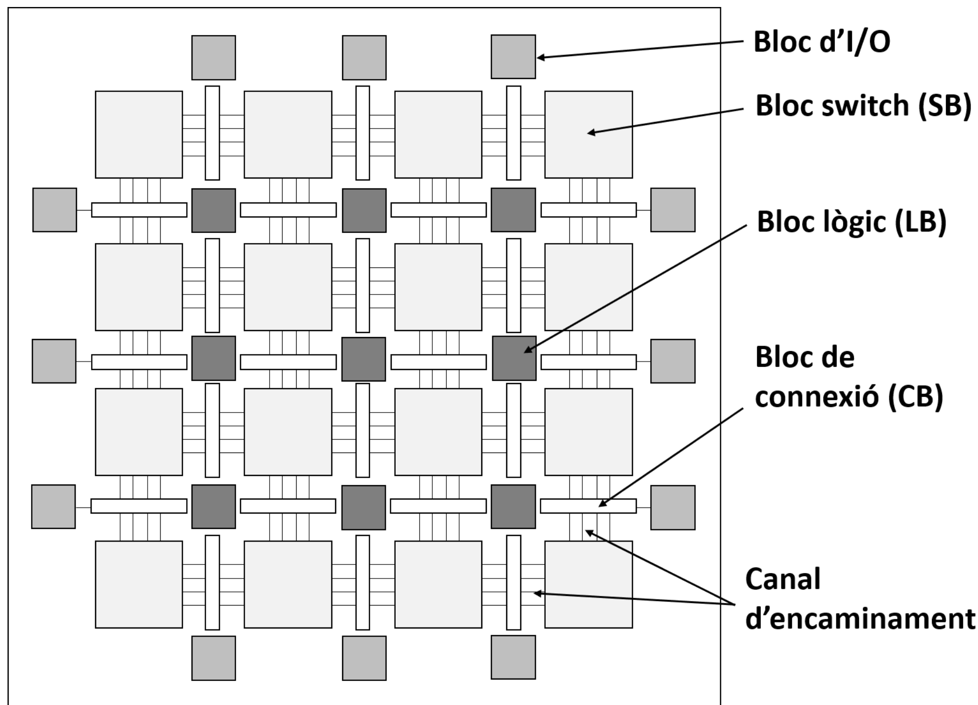

Figura 3.1: Estructura d'una FPGA tradicional.

3.1.1. Arquitectura de la FPGA

Les FPGA bàsiques es caracteritzen per tenir una estructura dividida en tres parts [2]. La primera part està composta pels elements lògics que formen els circuits lògics, que expressen una funció lògica. La segona conforma els elements d'entrada i eixida (I/O), encarregats de la comunicació amb els senyals de l'exterior i que donen una interfície externa. I la tercera part són els elements de cablejat o d'encaminament, que connecten els elements de la part lògica i la part d'entrada i eixida entre ells. Les FPGA més comercials també contenen circuits que realitzen funcions específiques com processadors, memòries o multiplicadors, entre les quals destaquen el processador de senyals digitals (DSP), les memòries incrustades que incrementen les capacitats de càlcul de la targeta, com les memòries RAM i els *phase-locked loop* (PPL) o el *delay-locked loop* (DLL), que proporcionen un sistema de rellotge al dispositiu. En la Figura 3.1 podem veure l'estructura bàsica d'una FPGA i els seus elements, on les funcions de cada part es detallen a continuació:

- Element lògic: són elements que implementen funcions lògiques dins de la targeta. Destaquen les taules *lookup* (LUT), variacions de les taules de valors de veritat on es poden programar per donar els valors que necessitem segons les entrades rebudes i els multiplexors (MUX), que en la majoria de casos fan la funció de seleccionadors d'una de les entrades per a propagar-la. Els elements lògics s'utilitzen per a implementar qualsevol circuit lògic, com per exemple el *flip-flop*, capaç d'emmagatzemar dos estats i controlar el resultat amb un rellotge, fent el seu funcionament síncron.

- Element d'entrada/eixida (I/O): són blocs que connecten els pins d'I/O amb els elements d'encaminament interns. Solen afectar el funcionament dels circuits mitjançant els senyals que reben de l'exterior. Les targetes comercials solen suportar estàndards d'entrada i eixida com PCI, PCIe, SSTL o altres sistemes de comunicació punt a punt.

- Element d'encaminament: estan formats per canals de cablejat, blocs de connexió (CB) i *switch* blocs (SB), que connecten els elements lògics amb altres elements lògics

i amb els elements d'entrada i eixida. Cada bloc *switch* és programable i es pot formar qualsevol ruta de connexió entre els elements de l'estructura de la FPGA.

- Altres elements: On destaquen els elements de memòria que, els fabricants com Xilinx, introdueixen en l'estructura bàsica com a blocs de memòria, que poden connectar-se i formar xips de fins a 500 Mb en alguns casos [4]

L'arquitectura de la Figura 3.1 és de les més bàsiques de les targetes FPGA. En aquest projecte hem treballat amb la targeta Alveo U200, construïdes a partir de l'arquitectura UltraScale+ i que funcionen amb l'arquitectura Alveo de Xilinx.

3.1.2. Targeta acceleradora Alveo U200

La targeta Alveo U200 és una FPGA dissenyada pel comercial Xilinx llançada en 2018. Construïda a partir de l'arquitectura de 16 nm UltraScale, aquesta targeta és capaç de ser reconfigurada per a gran quantitat de casos d'ús, com algoritmes de cerca en bases de dades, transcodificació de vídeo o inferència de xarxes neuronals artificials.

L'arquitectura UltraScale, de la qual estan dissenyades les targetes Alveo, naix en 2014, amb l'objectiu d'incrementar les capacitats de processament i les comunicacions d'amplada de banda [23]. Per a aconseguir açò, es fa un redisseny dels blocs lògics amb la inclusió dels *Fast Tracks*. Un gran problema de l'arquitectura clàssica de les FPGA és que als elements lògics s'incrementen en quantitat molt més ràpid que les interconnexions que els comuniquen. Les connexions són entre elements adjacents, però els *Fast Tracks* s'encarreguen d'enviar les dades entre elements lògicament connectats, com podem veure en la Figura 3.2. A l'hora de configurar la targeta existeixen moltes més formes de connectar els recursos lògics d'una forma òptima gràcies a la implementació d'aquestes connexions. També ajuda a reduir els colls d'ampolla en les comunicacions. Altres optimitzacions d'aquesta arquitectura són la inclusió de controladors de memòria DDR3 i DDR4, un sistema de rellotge similar a les ASIC, amb una gran quantitat de buffers de rellotge per transmetre els senyals de rellotge, a més de reconfiguració i gestió d'energia optimitzades.

La targeta Alveo està basada en aquesta arquitectura, i la aracterística reconfigurable de les FPGA a més de la seua gran quantitat d'elements, fan que aquesta targeta es puga adaptar a una gran varietat de càrregues de treball [1]. De les característiques de la targeta destaquen:

- 1.182.000 tables LUT: Amb aquestes dimensions de *lookup tables* hi ha una gran capacitat d'implementar gran quantitat de funcions lògiques.
- 2.364.000 registres: els registres en FPGA serveixen per a emmagatzemar bits, entre altres coses serveixen per a mantenir l'estat durant iteracions d'un bucle, sincronització d'entrada i eixida o comunicació amb la host. Tenir gran quantitat de registres resulta altament útil.
- 6.840 DSP *slices*: els DSP *slices* poden implementar funcions de processador de senyals, com per exemple multiplicació, multiacumulació, sumes de 4 entrades.
- 960 UltraRAMs: les ultraRAM són blocs de memòria disponibles en les arquitectures UltraScale, on cada una és una memòria RAM de 288Kb que permeten ser combinades per formar RAM més grans, formar registres RAM i crear vector de

[4] Especificacions de l'UltraRAM de Xilinx, blocs de memòria amb altes capacitats. https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf (visitada 27/06/2021)

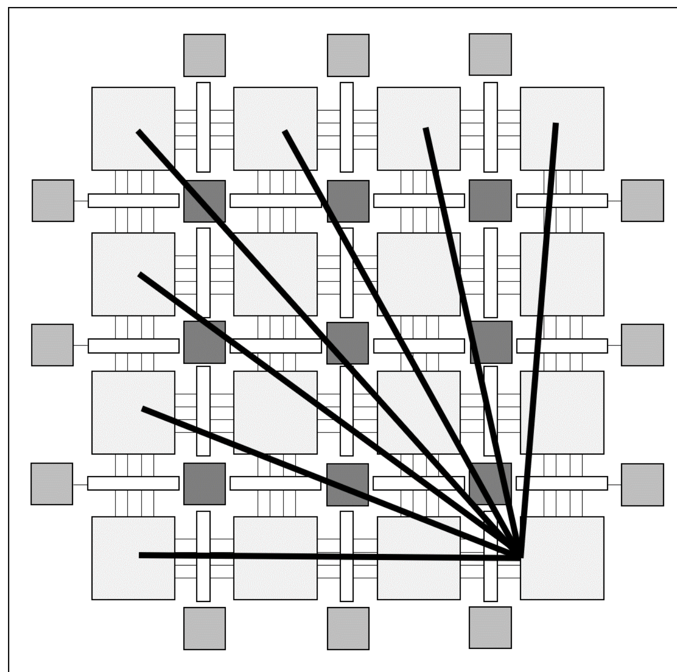

Figura 3.2: Concepte de *Fast Tracks* sobre una estructura de FPGA tradicional, on les rutes de connexió no cal que siguen amb els blocs adjacents.

memòria *on-chip* de grans dimensions, per no dependre tant de components de memòria externs.[5]

- Memòria DDR de 64 GB: Aquesta memòria RAM de grans dimensions permet executar aplicacions d'una càrrega elevada.

- Velocitat màxima d'amplada de banda DDR de 77 GB/s: les comunicacions amb la memòria RAM han de ser ràpides per tenir bona latència i prestacions en les aplicacions executades en la FPGA.

Totes aquestes característiques donen grans prestacions a aquesta targeta, que és capaç d'executar aplicacions molt costoses amb la configuració adequada. Per a convidar a l'ús de la targeta en aplicacions d'aprenentatge profund, Xilinx va crear la ferramenta Vitis AI, que configura la targeta per a aprofitar tots els seus recursos i executar la inferència de xarxes neuronals profundes.

3.2 Vitis AI

Per fomentar l'ús d'aquests dispositius en IA, en desembre de 2019, Xilinx va llançar l'entorn de desenvolupament Vitis AI, un programari capaç de suportar el procés d'inferència de la IA en dispositius de maquinari de Xilinx com són les FPGA, facilitant la programabilitat de la targeta sense haver de tindre alts coneixements del maquinari [32]. Les seues llibreries permeten accelerar el procés d'inferència en el maquinari de la companyia, com són les FPGA.

El seu procés d'optimització es pot realitzar en xarxes neuronals dissenyades amb els principals *frameworks* de desenvolupament d'aprenentatge profund, com són Tensor-

[5]Característiques de les UltraRAM https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf

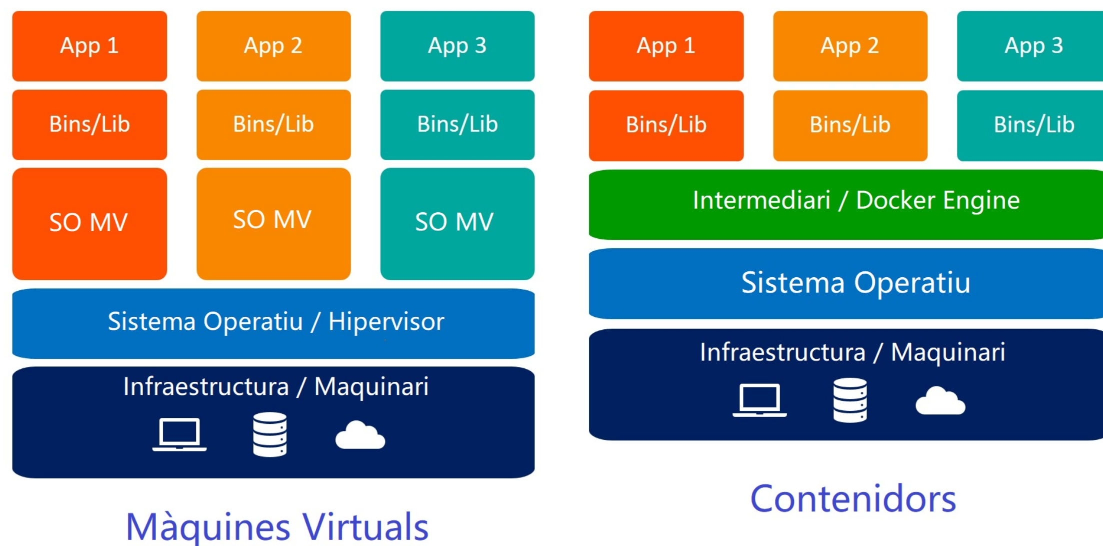

Figura 3.3: Diferències entre màquines virtuals i tecnologia de contenidors, on cada màquina virtual té el seu propi sistema operatiu mentre que els contenidors es limiten a aïllar recursos sobre el sistema operatiu amfitrió.

Flow, una plataforma de codi obert desenvolupada per Google o PyTorch, una altra plataforma de codi obert creada per Facebook. Permet als desenvolupadors de xarxes neuronals desplegar els seus models ràpidament en plataformes de Xilinx, com la FPGA Xilinx Alveo U200, que hem utilitzat en el desenvolupament d'aquest treball.

3.2.1. Instal·lació

Per poder utilitzar aquest programari cal seguir un procés d'instal·lació en el maquinari amfitrió que utilitza la targeta FPGA. Nosaltres disposem d'una computadora amb un processador de 12 nuclis, 64 GB de RAM i 1 TB d'emmagatzematge SSD, amb el sistema operatiu Linux Ubuntu 18.04. A aquesta computadora està connectada la targeta FPGA Alveo U200 via gigabit ethernet.

Aquest programari ve encapsulat utilitzant la ferramenta de contenidors software, Docker[6]. Docker és un projecte de codi obert especialitzat en l'automatització d'aplicacions fent ús de contenidors de programari. Els contenidors de programari permeten desplegar aplicacions aïllades de l'entorn físic i de la màquina on s'executen. Cada contenidor té el seu propi sistema d'arxius i els seus processos, amb llibreries, codi i configuracions que no afecten el servidor on estan funcionant. Els contenidors es creen a partir d'imatges, que són plantilles amb el programari i dependències que tindrà el contenidor quan s'execute.

El funcionament és similar al de les màquines virtuals, però amb la gran diferència que la màquina virtual desplega un sistema operatiu complet dins de l'amfitrió, mentre que els contenidors comparteixen la majoria de recursos amb el sistema l'amfitrió, utilitzant un intermediari que, en el nostre cas és *Docker engine*. Açò permet que els recursos de cada contenidor estiguen aïllats malgrat que no es desplegue un nou sistema operatiu sobre l'amfitrió. La tecnologia de contenidors és molt més ràpida d'utilitzar i permet als desenvolupadors crear aplicacions i desplegar-les més ràpidament, com és el cas de Vitis AI. En la Figura 3.3 podem veure la diferència entre aquestes dues tecnologies.

[6] Pàgina oficial de Docker `https://www.docker.com/` (url visitat 13/06/2021)

Tot açò ens permet tenir l'aplicació funcional en la nostra màquina sense haver de preocupar-nos per dependències de programari. Per poder executar els contenidors necessitem crear una imatge amb els recursos de la nostra aplicació. Vitis AI ens proporciona el codi necessari per a reconstruir les imatges i executar els contenidors. Amb les següents ordres podrem crear la imatge i executar un contenidor amb l'aplicació Vitis AI, situant-nos prèviament dins de la carpeta on hem descarregat els arxius de la ferramenta i amb Docker instal·lat i funcional en el nostre sistema.

```
$ cd ./docker
$ ./docker_build_cpu.sh
$ cd ..
$ ./docker_run.sh xilinx/vitis-ai-cpu:latest
```

Una volta tenim l'aplicació funcionant sobre un contenidor, sols necessitem models de xarxes neuronals artificials per optimitzar-los i realitzar el procés d'inferència sobre la targeta FPGA. En aquest projecte hem treballat amb models especificats amb la ferramenta TensorFlow, una ferramenta de codi obert que facilita el procés de creació de xarxes neuronals artificials i aprenentatge profund. En disposar d'un model entrenat cal seguir un flux de vida on el model serà quantitzat, compilat i desplegat en una targeta de Xilinx.

3.2.2. Procés de quantització

El procés d'inferència de les xarxes neuronals és molt costós i requereix connexions amb una gran amplada de banda per poder executar les aplicacions d'intel·ligència artificial amb una velocitat acceptable. A més, les FPGA solen ser dispositius amb una memòria limitada on models excessivament grans de xarxes neuronals poden causar problemes d'assignació i desbordament. Per poder aconseguir executar aquests models tan grans i obtenir poca latència d'execució existeix la tècnica de quantització dels pesos d'una xarxa neuronal artificial.

Aquesta tècnica consisteix a reduir la dimensió dels pesos i les activacions del model prèviament entrenat. Els pesos i les activacions estan en aritmètica de coma flotant de 32 bits (FP32), tipus de dades numèric on cada nombre ocupa 32 bits de memòria, és a dir, 4 bytes. El procés de quantització de Vitis AI utilitza la ferramenta Vitis AI *quantizer*. Aquest busca transformar els FP32 al tipus de dades INT8, format de dades per representar enters des de -128 fins a 127 que tan sols ocupa 8 bits, és a dir, 1 byte. Amb aquesta reducció de grandària aconseguim comprimir els models perquè s'executen paral·lelament amb menys consum d'energia en targetes FPGA, però, açò implica una pèrdua de precisió a l'hora de la inferència.

Altrament, el procés que segueix la ferramenta per quantitzar els pesos requereix diversos passos previs. Primerament, necessitem un model de xarxa neuronal entrenat amb un conjunt de dades, que utilitzarem per a obtenir la versió quantitzada. Segonament, necessitem una xicoteta porció de la part de validació del conjunt de dades utilitzat, amb 1% del total de les imatges de validació és suficient. Per acabar, necessitarem desenvolupar una funció encarregada del preprocessament de les dades, ja que depenent de cada model i conjunt de dades les imatges han de passar per un processat, on es facen operacions com redimensionar o retallar la mostra. El procés consisteix a fer passar al model per una segona fase d'entrenament amb el conjunt de dades seleccionat, perquè el *quantizer* calibre les activacions i reculla estadístiques necessàries per a la quantització dels pesos. Al final obtindrem un arxiu amb estadístiques de calibració i quantització que utilitzarem per a crear un model capaç de ser desplegat a la targeta FPGA.

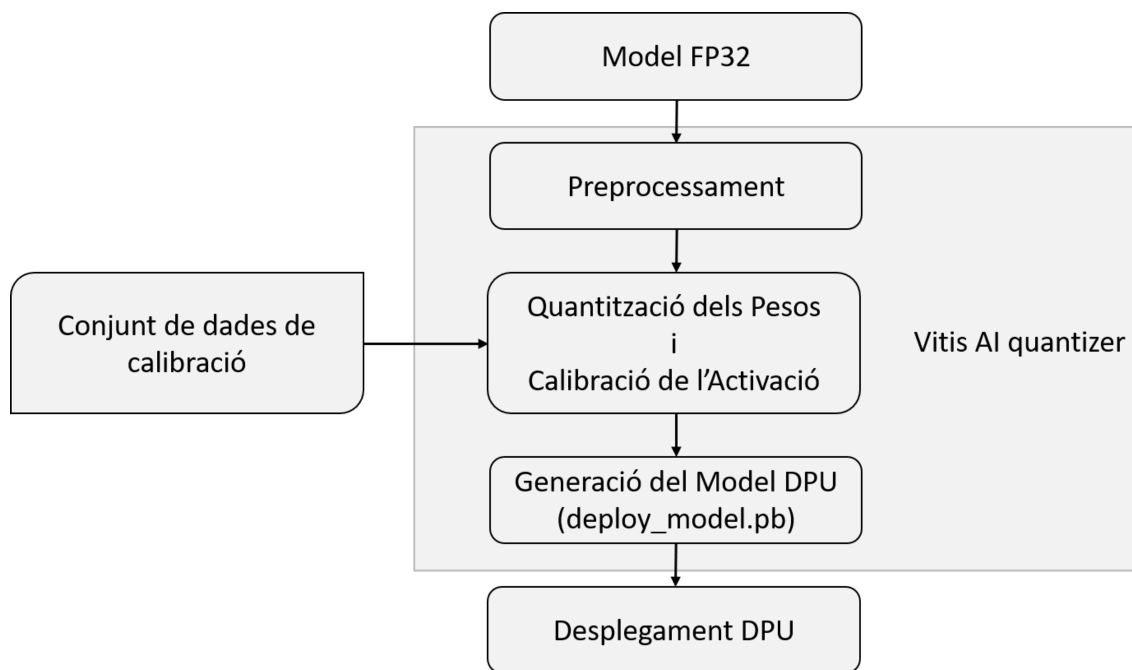

Figura 3.4: Flux a seguir per quantitzar un model en precisió FP32 en el Vitis AI *quantizer*.

Amb tots aquests elements podem passar a utilitzar el *quantizer* per obtenir el nostre model quantitzat. En el nostre cas, hem utilitzat la plataforma TensorFlow i, en l'exemple que tractarem, un model Resnet amb 50 capes de profunditat i entrenat amb el conjunt de dades ImageNet 2012 que consta de 1000 classes diferents [10] [24].

En aquest treball s'ha utilitzat la versió 1.2.1 de Vitis AI, que ofereix suport de quantització a models especificats en TensorFlow, Caffe o PyTorch. Cadascuna d'aquestes plataformes requereix unes dependències de programari diferents que poden causar conflictes si estan totes funcionals en el mateix entorn. Per exemple, podria ser que Caffe faça ús de llibreries Python amb diferents versions o incompatibles amb les que utilitza TensorFlow. Cal destacar que per a aïllar el programari necessari de Vitis AI del programari instal·lat en la màquina local hem utilitzat Docker, però ara, per aïllar les dependències de les diferents plataformes de desenvolupament de xarxes neuronals artificials, utilitzarem el gestor de paquets Conda[7].

Conda és un gestor de paquets i gestor d'entorns virtuals que permet crear entorns virtuals, que són ferramentes amb la capacitat de separar les dependències, els arxius i els paquets de diferents projectes, sense entrar en contacte amb altres entorns. Els contenidors de Vitis AI utilitzen diferents entorns virtuals per a les diferents plataformes de programari. Nosaltres utilitzarem l'entorn especificat per a TensorFlow[8], per activar-lo caldrà activar la següent ordre en el contenidor que està corrent l'aplicació.

```
$ conda activate vitis-ai-tensorflow
```

TensorFlow facilita la creació de models de xarxes neuronals, donant suport a l'entrenament i la inferència amb gran quantitat de ferramentes, llibreries i recursos. Aquesta ferramenta té alta sinergia amb Keras [9], una llibreria que ofereix un gran suport a funcions d'aprenentatge profund. Els models especificats en Keras-TensorFlow es poden

[7]Guia d'usuari de Conda https://conda.io/projects/conda/en/latest/user-guide/index.html (visitada 13/06/2021)

[8]Pàgina web de TensorFlow amb informació de la plataforma. https://www.tensorflow.org/ (visitada 26/06/2021

[9]Què és Keras i usos en aprenentatge profund https://keras.io/

emmagatzemar de moltes formes com a fitxers, però la més sencilla és guardar-los en un arxiu amb extensió *Hierarchical Data Format Version 5*[10] (HDF5). El HDF5 és un format que permet emmagatzemar i organitzar grans quantitats de dades. En aquest cas guardem l'arquitectura del model, la configuració d'entrenament, els pesos i altres paràmetres relacionats amb el model de xarxa neuronal. Quantitzar un model de TensorFlow en Vitis AI 1.2.1 requereix tenir aquesta informació en un *frozen graph* o model çongelat". Per poder convertir un model a çongelat", utilitzarem la ferramenta `tf2onnx`[11], que permet interactuar amb arxius que contenen especificacions de models i convertir-los a altres formats amb molta facilitat. Per exemple, per convertir el model emmagatzemat amb Keras-TensorFlow en HDF5, a *frozen graph*, executarem la següent ordre després d'instal·lar la ferramenta, on en `-keras` indicarem l'arxiu HDF5 i en `-output_frozen_graph` ficarem el nom de l'arxiu amb extensió .pb on anirà el nostre model çongelat":

```
$ python -m tf2onnx.convert --keras model.h5 \
                             --output_frozen_graph frozen_graph.pb
```

Els *protocol buffers* són estructures creades per Google, que permeten serialitzar i estructurar les dades «com XML però més reduït, més ràpid i més simple»[12]. Com les xarxes neuronals artificials estan estructurades per capes, és un gran avantatge serialitzar-les amb un mecanisme simple i ràpid, on cada capa té les seues característiques distribuïdes d'una forma amb fàcil interacció i que permet l'ús d'aplicacions com Netron[13]. Aquest útil permet visualitzar grafs i estructures de dades especificades amb aquest protocol. La Figura 3.5 és el resultat de l'ús de la ferramenta en el nostre model çongelat", on podem veure a l'esquerra les primeres capes de la xarxa neuronal i a la dreta, les últimes.

El procés de quantització en TensorFlow requereix un pas addicional, que és indicar quins són els nodes d'entrada i d'eixida del model de xarxa neuronal. Fent ús de la ferramenta Netron podem visualitzar millor l'estructura d'una xarxa neuronal artificial. Els nodes d'entrada són les capes d'entrada de la xarxa neuronal, en la Figura 3.5 podem veure un model de Resnet50, on el node d'entrada és una capa *input*, que s'utilitza com a punt d'entrada a la xarxa. Si ens fixem en els últims nodes de 3.5 la capa d'eixida del nostre model de Resnet és l'última *Reshape*, capa que canvia la forma del vector de pesos rebut. Hi haurà casos on solament visualitzant el graf de la xarxa neuronal no serà suficient per a determinar les capes d'entrada i d'eixida, aleshores utilitzarem l'ordre *inspect*, que és una funcionalitat del Vitis AI *quantizer*, capaç d'examinar el model *freeze* i determinar els nodes d'eixida i d'entrada. Tan sols hem d'indicar amb `-input_frozen_graph` la ruta fins al nostre *frozen model*. L'execució i resultat d'aquesta ordre en el model visualitzat en la 3.5 el podem veure a continuació.

```
$ vai_q_tensorflow inspect --input_frozen_graph resnet50.pb

Found 1 possible inputs: (name=input, type=float(1), shape=[?,224,224,3])
Found 1 possible outputs: (name=resnet50/predictions/Reshape_1, op=Reshape)
```

L'últim pas abans de poder quantitzar el model és seleccionar una porció del conjunt de dades utilitzat en l'entrenament i la validació del model. En el nostre cas hem utilitzat el model ImageNet 2012, amb 150.000 imatges etiquetades de 1000 classes diferents per a

[10] Què és i en què s'utilitza el HDF5 https://www.hdfgroup.org/solutions/hdf5/

[11] Repositori de la ferramenta `tf2onnx`, https://github.com/onnx/tensorflow-onnx (visitada 26/06/2021)

[12] Que és i com utilitzar els *protocol buffers* https://developers.google.com/protocol-buffers (visitada 13/06/2021)

[13] Repositori de l'aplicació Netron https://github.com/lutzroeder/netron (visitada 13/06/2021)

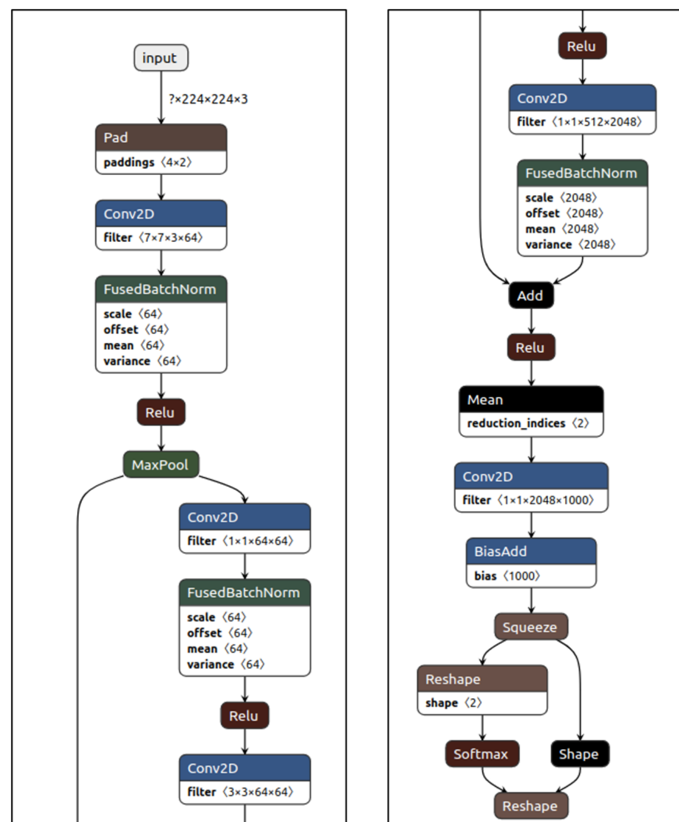

Figura 3.5: Execució de l'aplicació Netron sobre un model Resnet50 dissenyat amb TensorFlow. En l'esquerra, primeres capes de la xarxa neuronal. En la dreta, últimes capes del model.

l'entrenament i 50.000 imatges per al procés de validació. Hem seleccionat 1000 imatges per al procés de calibració i quantització. El preprocessament de les imatges en aquest model de xarxa neuronal requereix una sèrie d'operacions que s'han definit en un arxiu de Python, amb la funció *calib_input*, que s'encarrega de preprocessar cada imatge. A més, caldrà especificar un arxiu de text amb el nom de cada imatge en una línia, per a facilitar el preprocessat que realitza el programa.

Una volta ja tenim tots els paràmetres necessaris per a la quantització, podem procedir a quantitzar el nostre model amb l'execució de l'ordre:

```
vai_q_tensorflow quantize --input_frozen_graph resnet50.pb \
--input_nodes input --input_shapes ?,224,224,3 \
--output_nodes resnet50/predictions/Reshape_1 \
--input_fn input_fn.calib_input --calib_iter 300
```

On en `-input_frozen_graph` indicarem el resultat de la conversió `tf2onnx`, és a dir, el nostre model guardat en *protocol buffers*; en `-input_nodes` i `-output_nodes` la primera i última capa del nostre model, que hem pogut obtenir utilitzant el `inspect`; a més, cal indicar el `-input_shape` les dimensions de les imatges que es reben en la primera capa, per exemple, les imatges d'ImageNet tenen 224 píxels d'amplada i d'alçada, i 3 canals de color (RGB), per tant en aquest camp indicarem "?,224,224,3", on el primer paràmetre és la dimensió dels lots, desconegut per a nosaltres així que indicarem una interrogació. També hem d'indicar la funció encarregada de fer el preprocessament en `-input_fn`, en el nostre cas és `calib_input` dins de l'arxiu `input_fn.py` disponible en l'annex A.1.1; amb `-calib_iter` indicarem el nombre d'iteracions que volem que dure aquest procés, on en

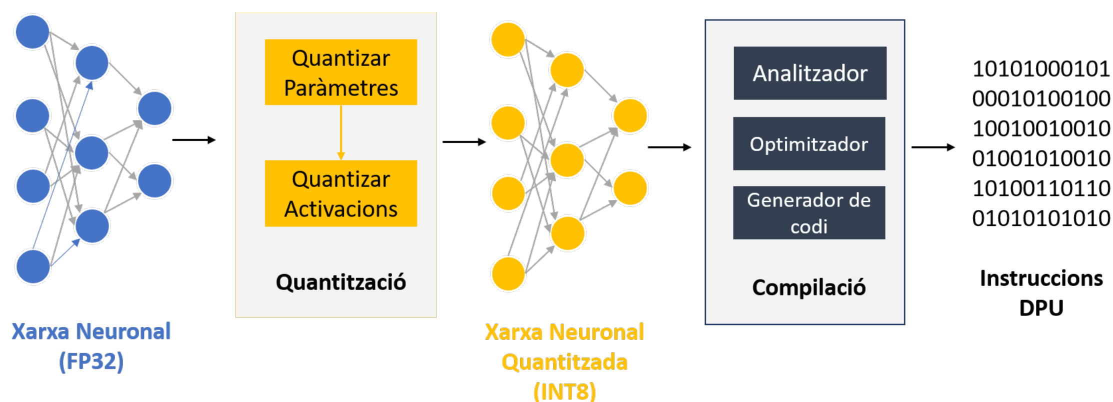

Figura 3.6: Flux a seguir en models de xarxes neuronals en Vitis AI, on es quantitza i compila la xarxa per generar instruccions DPU.

cada iteració es recorrerà tot el conjunt de dades en un procés similar a l'entrenament per traure les estadístiques necessàries per a la quantització.

L'execució d'aquesta ordre resulta en l'obtenció de dos models en *protocol buffers*, un dels quals ens servirà per a compilar i desplegar la xarxa neuronal en una FPGA, i amb l'altre podem comprovar la precisió que, teòricament, tindrà el resultat final d'aquest model la precisió reduïda INT8.

```
INFO: Deploy Model Generated.
******************** Quantization Summary ********************
INFO: Output:
quantize_eval_model: ./quantize_results/quantize_eval_model.pb
deploy_model: ./quantize_results/deploy_model.pb
```

3.2.3. Procés de compilació

El procés de compilació és el següent pas per aconseguir desplegar un model de xarxa neuronal artificial utilitzant la ferramenta Vitis AI. En aquest procés transformarem el resultat de la quantització realitzada prèviament en una seqüència d'instruccions *Deep-Learning Processor Unit* (DPU).

Una unitat de processador d'aprenentatge profund o DPU, és un motor programable optimitzat per a xarxes neuronals profundes. És un grup de nuclis parametritzables de propietat intel·lectual de Xilinx, especificats per un funcionament òptim en el seu maquinari. Estan dissenyats per accelerar les càrregues de computació d'algorismes d'aprenentatge profund, com les xarxes neuronals de classificació d'imatges en les quals estem treballant. La compilació amb DPU facilita la implementació eficient de xarxes neuronals artificials.

En el nostre cas, utilitzarem la DPUCADX8G, una DPU especialitzada en xarxes convolucionals, especificada per a les targetes FPGA Alveo U200 i U250 i per treballar amb xarxes amb els pesos quantitzats al format INT8. En la Figura 3.7 podem veure les característiques de la DPU, on destaquen:

- 96x16 Array Sistólic processador de senyals (DSP) operant a 700 MHz
- Model de programació basat en instruccions per a simplificar la representació de models de xarxes neuronals
- 9 MB en xip de memòria de Tensors formada per UltraRAMs

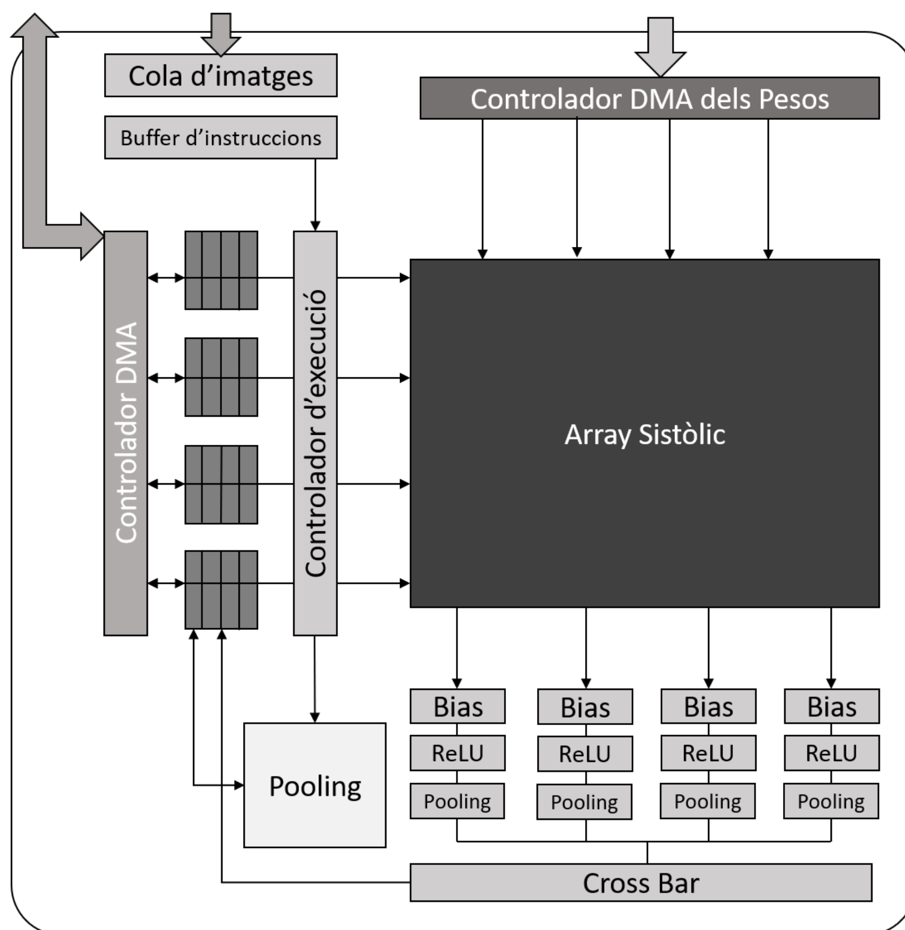

Figura 3.7: Arquitectura del DPUCADX8G.

- Memòria DDR externa per a les dades dels Tensors
- Un pipeline de Scale, activacions ReLU i operacions de Pooling per a obtenir la màxima eficiència
- Bloc independent d'execució d'operacions Pooling, per obtenir processament paral·lel amb les capes convolucionals

Tot açò i altres característiques que permeten als models compilats tenir un suport eficient en les targetes FPGA mencionades. L'entorn de Vitis AI proporciona les característiques necessàries per a compilar models amb aquesta DPU en un arxiu JSON.

Així doncs, partim del nostre model de TensorFlow que hem quantitzat prèviament, tenint com a resultat l'arxiu *deploy_model.pb*, el qual té les característiques necessàries per a desplegar el model final en la FPGA, com són la calibració de les activacions i els valors quantitzats dels pesos. El procés de compilació requereix dos passos, el primer pas serà extraure un arxiu de text amb tota la informació relativa de la quantització en les capes de la xarxa neuronal i el segon pas consisteix a utilitzar aquest arxiu per generar els fitxers resultats amb els quals podrem fer inferència en la FPGA. Amb la següent ordre podem extraure el resum de la quantització en un fitxer de text:

```
$ vai_c_tensorflow --frozen_pb quantize_results/deploy_model.pb \
--output_dir out  --net_name resnet50 \
--arch /opt/vitis_ai/compiler/arch/DPUCADX8G/ALVEO/arch.json \
-q
```

Hem d'indicar el model resultant de la quantització amb -frozen_pb, un directori existent on s'emmagatzemarà el resultat amb -output_dir, ficar un nom arbitrari a la xarxa amb -net_name, l'arxiu JSON que indica que la compilació ha de fer-se amb l'arquitectura de la DPUCADX8G amb -arch i, per últim, establir l'opció d'extraure la informació de la quantització amb -q. Els arxius amb les arquitectures DPU estan disponibles en la ubicació /opt/vitis-ai/compiler/arch del contenidor Docker de Vitis AI.

Seguidament, tenim ja disponible el resum de la quantització en un arxiu de text. L'última etapa de la compilació requereix utilitzar una altra volta el nostre model çongelat", la versió sense quantitzar de la xarxa neuronal. Proporcionant al compilador aquest model i la informació de la quantització, aquest s'encarrega de la generació dels fitxers finals. Amb la següent ordre aconseguirem els resultats:

```
$ vai_c_tensorflow --frozen_pb resnet_50.pb \
--output_dir model --net_name resnet50 \
--arch /opt/vitis_ai/compiler/arch/DPUCADX8G/ALVEO/arch.json \
--options "{'quant_cfgfile':'out/resnet50_fix.txt', \
'placeholdershape':{'input':[1,224,224,3]}}"
```

On podem veure que hem d'indicar el model *freeze* de la xarxa neuronal amb -frozen_pb, un directori on s'emmagatzemaran els resultats amb -output_dir, el nom de la xarxa amb -net_name i l'arxiu JSON amb l'estructura de la DPUCADX8G amb -arch. A més, amb l'opció -options indicarem la ubicació del fitxer de text amb les característiques de la quantització, indicant abans que és 'quant_cfgfile' i el 'placeholdershape', que són les dimensions que tenen les imatges en entrar a la xarxa neuronal pel node indicat, on indiquem el nom del node d'entrada, 'input' en el cas del model Resnet50 utilitzat, i un vector de 4 dimensions amb la dimensió del lot, l'alçada, l'amplada i el nombre de canals de color de la imatge.

Com a resultat final tenim la generació de 4 fitxers per a ser utilitzats en el desplegament en la FPGA. Els arxius són els següents:

- compiler.json: Arxiu JSON que conté instruccions de maquinari de baix nivell.
- weights.h5: Arxiu HDF que emmagatzema els pesos de la xarxa neuronal en format FP32.
- quantizer.json: Fitxer JSON amb factors d'escalat per a cada capa del model especificat.
- meta.json: Fitxer JSON amb rutes a les diferents llibreries necessàries, altres arxius, DPU utilitzat.

3.2.4. Desplegament a la FPGA

Per acabar, sols queda provar l'eficiència del nostre model optimitzat amb Vitis AI a la FPGA Alveo U200. Per a açò haurem de reconfigurar la FPGA amb un arxiu binari i utilitzar alguna de les API que ens proporciona Vitis AI per la inferència en FPGA.

Les FPGA són dispositius amb la característica principal de ser reprogramables, la qual cosa les permet implementar un disseny de maquinari que fa específicament el que l'usuari vol. Després del disseny del maquinari, la FPGA s'ha de programar utilitzant un arxiu binari, fent el que es coneix com a configuració de la FPGA. Una targeta FPGA pot ser reconfigurada constantment i, per configurar les targetes de Xilinx, Vitis AI ens

proporciona els binaris en un arxiu anomenat *xclbin*. Per poder realitzar la inferència en la targeta Alveo U200, abans caldrà executar la següent ordre en un contenidor que estiga funcionant l'aplicació Vitis AI:

```
source /workspace/alveo/overlaybins/setup.sh
```

El codi de *setup.sh* s'encarrega de reconfigurar la targeta i localitzar les llibreries necessàries per a executar inferència de xarxes neuronals.

A continuacó caldrà utilitzar les API de Vitis AI de C++ o Python. El funcionament és el mateix per a qualsevol API: utilitzar la classe *Runner* proporcionada per Vitis AI, obtenir els tensors d'entrada i eixida del model i alimentar els tensors d'entrada amb imatges preprocessades per a realitzar la inferència.

L'equivalent als vectors en aprenentatge profund són els tensors, són estructures de dades que ajuden a emmagatzemar les dades en diferents dimensions en les xarxes neuronals. Un tensor és un vector de N-dimensions de dades. Si alimentem la capa d'entrada de la xarxa neuronal amb un tensor que tinga les característiques de la imatge, en el tensor de la capa d'eixida tindrem el resultat de la inferència de la xarxa neuronal. A més, podem afegir diverses imatges pel tensor d'entrada perquè la FPGA processe més d'un resultat a l'hora. Açò es coneix com a grandària de *batch* o dimensió dels lots.

La classe *Runner* utilitza les llibreries amb propietat intel·lectual de Xilinx per poder realitzar inferència asíncrona d'imatges rebudes com a argument. A més, gestiona la comunicació entre FPGA i amfitrió i l'obtenció dels tensors de la xarxa neuronal artificial.

Per últim, hi ha capes del model que no s'executen en la FPGA, per motius d'eficiència o de falta suport en Vitis AI. És el cas de la capa *Softmax*, una funció d'activació que computa el resultat final en les xarxes neuronals classificadores. Aquesta operació per tant ha de ser executada en el sistema amfitrió (CPU), no en la FPGA. Tan sols caldrà implementar la funció en el codi que compute els resultats rebuts de la FPGA.

Vitis AI proporciona un executable de Python que realitza totes aquestes funcions descrites. Amb unes lleugeres modificacions hem desenvolupat el codi que realitza la inferència dels models optimitzats amb Vitis AI en la targeta FPGA Alveo U200. A l'annex A.1.2 està disponible l'executable modificat que executa inferència en una imatge a la FPGA.

CAPÍTOL 4

Suport del procés d'inferència en GPU

En aquest capítol aprofundirem en el treball realitzat amb el programari TensorRT de NVIDIA, especialitzat a optimitzar el procés d'inferència de models d'aprenentatge profund en GPU de NVIDIA. També ens endinsarem en l'arquitectura de la GPU utilitzada i les característiques que aprofita aquest programari.

4.1 GPU

La unitat de processament gràfic o GPU, és una de les tecnologies de computació més potents hui en dia. Va ser dissenyada per a la computació paral·lela de dades, per aquest motiu ofereix un gran rendiment en aplicacions gràfiques, renderització de vídeo, mineria de dades i, en els últims anys, s'ha aprofitat el seu potencial en el camp de la intel·ligència artificial.

La història d'aquests dispositius està molt lligada a la història dels videojocs, on les primeres màquines *arcade* utilitzaven circuits especialitzats en els sistemes gràfics. De fet, el terme GPU va ser utilitzat per primera volta en 1994 per l'empresa *Sony*, quan presentava la videoconsola *PlayStation*. Però no seria fins a 1999 quan una companyia anomenada NVIDIA, inventa el que avui en dia coneixem com a *Graphic Processor Unit* (GPU), la GeForce 256 GPU de NVIDIA. Aquesta targeta gràfica oferia grans millores respecte a les versions anteriors de GPU de NVIDIA, i per això la companyia la va batejar com a l'la primera 'GPU' del món", un dispositiu capaç de processar un mínim de 10 milions de polígons per segon. [1]

A partir d'ací l'evolució de les GPU va seguir potenciant els gràfics dels ordinadors i altres sistemes com les videoconsoles o superordinadors. Tot això i fins a la dècada de 2010, on els investigadors que treballaven amb la intel·ligència artificial i l'aprenentatge automàtic s'adonen del potencial que tenen les unitats de processament gràfic en el món de les xarxes neuronals artificials. L'arquitectura de les GPU està dissenyada per poder computar multiplicacions de vectors amb altes prestacions, operacions que abunden en els sistemes gràfics, els videojocs i en les xarxes neuronals. L'ús d'aquests dispositius en aprenentatge automàtic i aprenentatge profund va resultar altament fructífer, i companyies com NVIDIA i AMD comencen a endinsar-se en aquest món, potenciant l'etapa d'entrenament d'aquests algoritmes tan costosos.

[1] Linía temporal de la companyia NVIDIA https://www.nvidia.com/es-es/about-nvidia/corporate-timeline/

4.1.1. Arquitectura de la GPU

L'arquitectura de la GPU és òptima per als algoritmes de IA. Pel que fa a la unitat central de processament (CPU), sol estar optimitzada per a executar tasques amb la mínima latència possible, però amb la capacitat de canviar d'operació ràpidament. Una GPU està dissenyada per executar el màxim nombre de tasques en el menor temps possible, executant gran quantitat de tasques en paral·lel. La gran diferència entre ambdues és que la GPU té molts més nuclis per a processar les tasques, es prioritza un alt rendiment abans que una baixa latència.

Aquesta característica fa que la GPU siga millor que la CPU en tasques matemàtiques, en paral·lelització de dades i en operacions d'alt cost matemàtic. En la Figura 4.1 podem veure l'esquema general de l'arquitectura de la GPU on destaquen els següents elements:

- Multiprocessadors de *Streaming* (SM) o nuclis, que comparteixen una memòria global de GPU.
- Una memòria catxe L2. Aquesta memòria intermèdia s'encarrega de reduir els temps d'accés a la memòria principal de la GPU.
- Components d'Accés Directe a Memòria (DMA). S'encarreguen copiar dades de la GPU a la memòria principal del sistema, encara que també es poden utilitzar per a la comunicació amb altres dispositius.

Els nuclis o SM estan formats per diferents components:

- Un Banc de Registres. Una gran quantitat de registres que poden ser particionats i repartits pels fils d'execució.
- Planificadors *warp*. Poden canviar de context ràpidament entre els fils i planificar instruccions que estiguen preparades per a l'execució.
- Una memòria compartida entre els SM i una memòria local.
- Diferents sistemes de memòria catxe.
- Unitats de càrrega i emmagatzematge de dades (L/S).
- Unitats de textures. Associen la informació dimensional en els buffers de la memòria, són molt útils a l'hora de carregar gràfics, una de les principals labors de la GPU.
- Unitats de còmput. Hi ha molts tipus d'unitats de còmput, com per exemple les Unitats Aritmètiques (ALU), que implementen operacions aritmètiques i lògiques entre valors i també conegudes com a *Cores* en l'arquitectura NVIDIA. Algunes targetes també inclouen unitats aritmètiques especialitzades en *half precision* i *double precision* per a executar les operacions en aquests tipus més ràpids. També es dóna el cas en l'arquitectura NVIDIA, la inclusió d'Unitats de Funcions Especials (SFU), que s'encarreguen de computar ràpidament operacions especials Per exemple, en l'arquitectura *Volta* s'inclouen les unitats *Tensor* que acceleren el còmput de les multiplicacions de matrius, operació molt comuna en l'aprenentatge profund[2].

[2] Característiques dels *Tensor Cores* de NVIDIA. https://www.nvidia.com/es-es/data-center/tensor-cores/

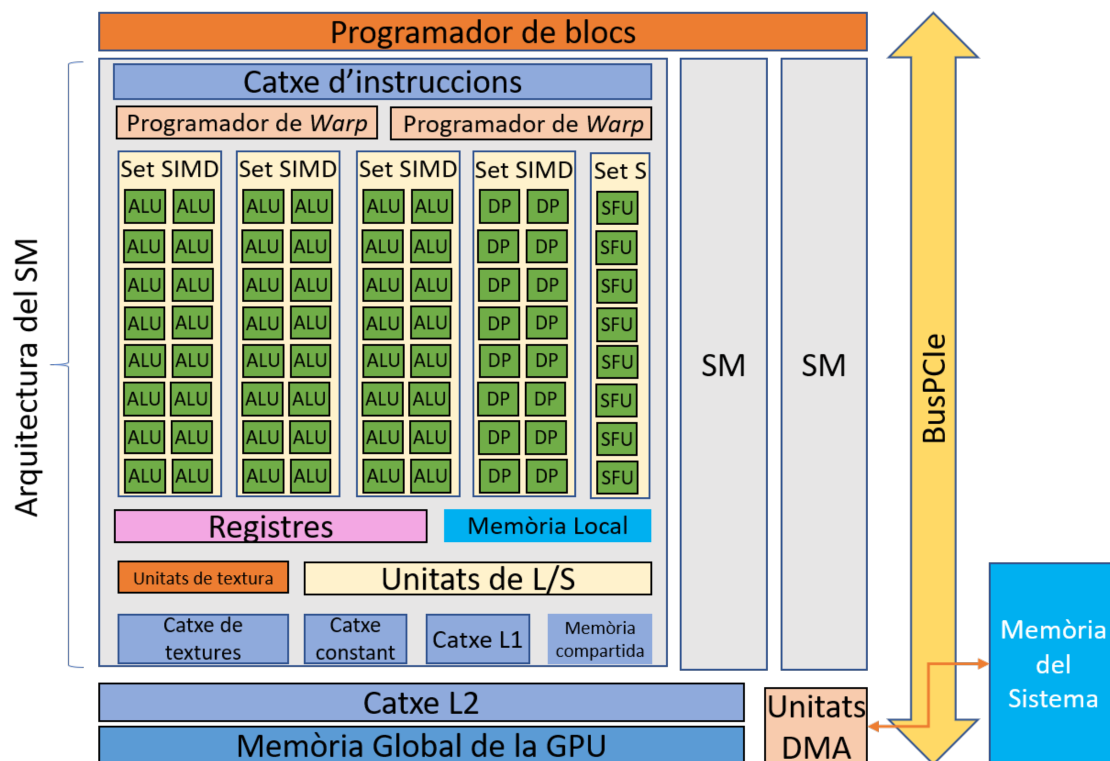

Figura 4.1: Arquitectura bàsica de la GPU i dels Multiprocessadors de *Streaming*.

Les unitats de còmput no operen independentment, sinó en grups que operen utilitzant la tècnica SIMD (una instrucció, múltiples dades), una tècnica utilitzada per al paral·lelisme de tasques. Cada grup és capaç d'executar la mateixa instrucció en diferents elements de dades al mateix temps. [3] [33]

En aquest treball s'ha utilitzat la GPU NVIDIA GeForce GTX 1050, una targeta dissenyada amb l'arquitectura *Pascal* i amb un rendiment fins a tres voltes major que les de la generació anterior, segons NVIDIA[3]. Compta amb 640 NVIDIA CUDA *Cores* o unitats de còmput, 2 GB de memòria RAM, amb una velocitat de memòria de 7 Gbps i un funcionament de 1455 MHz.

4.1.2. Model d'execució CUDA

Les arquitectures GPU utilitzen el model de processament de dades SIMT (una instrucció, múltiples fils). El treball es representa com una xarxa de tres dimensions de tasques o fils com les anomena CUDA. Tots els fils estan executant el mateix codi que s'anomena *kernel*, però cada fil té assignat un índex `(x, y, z)` dins de la xarxa.

La xarxa o graella es divideix en múltiples blocs de la mateixa mida. Els blocs s'assignen a un SM específic i són executats en aquest nucli exclusivament. Quan es programa un bloc, tots els fils que pertanyen a aquest bloc es converteixen en residents del SM i s'assignen tots els recursos de maquinari disponibles per executar la tasca. Però no tots els fils d'un mateix bloc s'executen simultàniament. El SM distribueix els fils de cada bloc en unitats de 32/64 fils anomenades *warps*, on tots els fils de cada *warp* s'executen al mateix temps amb els recursos disponibles.

[3] Pàgina web amb detalls de la GeForce GTX 1050 https://www.nvidia.com/es-la/geforce/products/10series/geforce-gtx-1050/

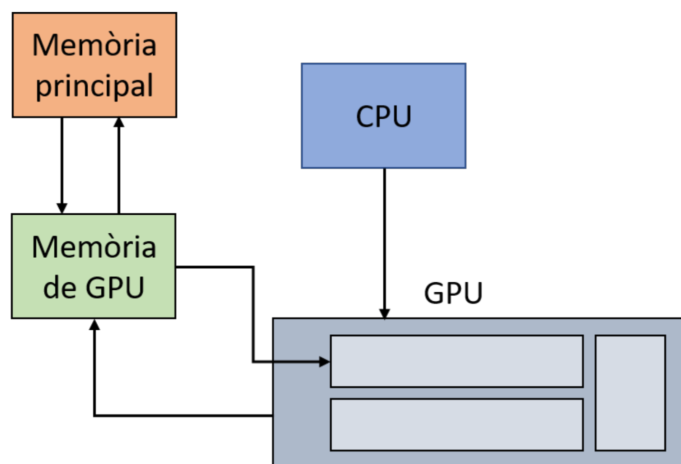

Figura 4.2: Flux de funcionament de l'arquitectura CUDA.

D'aquesta manera l'execució d'operacions matemàtiques altament costoses, com la multiplicació de matrius, es reparteix d'una manera òptima per tota la GPU i s'obtenen rendiments elevats en aquest dispositiu respecte a altres. El flux de processament és com es pot veure en la figura 4.2, on primerament es copien les dades de la memòria principal a la memòria de la GPU, després la CPU llança el kernel que serà executat en la GPU, la GPU executa aquestes instruccions en paral·lel en cada SM i, finalment, copia el resultat a la memòria principal.

4.2 TensorRT

El programari TensorRT és un conjunt de ferramentes de NVIDIA que faciliten el desenvolupament d'aplicacions d'inferència en xarxes neuronals en GPU. Naix en 2018 amb l'auge de les targetes gràfiques i de la IA. Les GPU aconsegueixen accelerar notablement el procés d'entrenament dels algoritmes d'aprenentatge profund, però aquesta ferramenta busca elevar les prestacions de les targetes gràfiques en el procés d'inferència de les xarxes neuronals artificials [29].

Realitzar el procés d'inferència en una xarxa neuronal pot ser costós. La majoria de plataformes de desenvolupament ofereixen funcions per realitzar aquest procés, però solen donar molt poques prestacions respecte a les quals podrien obtenir si es desenvolupara una aplicació específica per a la inferència. Per a aconseguir majors prestacions, la creació d'una aplicació per a executar la xarxa neuronal sol ser la millor opció, però això implica alts coneixements especialitzats a maquinar i xarxes neuronals a més de dedicació sols per a aconseguir bons resultats en una GPU moderna. L'API de TensorRT implementa de forma eficient en les GPU de NVIDIA, la majoria de funcionalitats que es requereixen en les xarxes neuronals profundes, com per exemple la convolució[4].

4.2.1. Instal·lació

Per poder fer ús de les eines de la ferramenta, cal tenir instal·lats al sistema alguns programaris i plataformes en el nostre sistema. En el nostre cas utilitzarem una computadora amb un processador de 8 nuclis, 16 GB de memòria RAM i 512 GB d'emmagatzematge

[4]API de Python de TensorRT utilitzada https://docs.nvidia.com/deeplearning/tensorrt/api/python_api/index.html (visitada 26/06/2021)

HDD. La targeta gràfica NVIDIA de la que farem ús és una NVIDIA GeForce GTX 1050[5], que implementa l'arquitectura Pascal.

La base de qualsevol aplicació que utilitze una unitat de processament gràfic de NVIDIA és *Compute Unified Device Architecture* (CUDA). CUDA és un model de programació i una plataforma de computació en paral·lel per al còmput en GPU [33]. La plataforma s'encarrega d'aprofitar al màxim les característiques de les GPU permitent l'execució simultània d'una gran quantitat de fils de processament, ja que la quantitat de nuclis que tenen les GPU permeten aquest nivell de paral·lelisme. Aquest programari de codi obert va ser publicat en 2006 pels desenvolupadors de NVIDIA i té una gran quantitat d'API i suport per a moltes plataformes, però la més destacada i amb el suport principal és la programació de CUDA en C++. TensorRT està desenvolupat sobre aquest model de programació paral·lel.

El funcionament de les aplicacions CUDA en plataformes heterogènies, el podem veure en la 4.2. La unitat central de processament s'encarrega d'executar el codi principal i la GPU executa els kernels que la CPU li indique, mentre que la comunicació de les dades es farà mitjançant còpies en memòria entre la memòria de la GPU i la de la CPU.

Amb la instal·lació de CUDA també tindrem disponible la llibreria cuBLAS(*Basic Linear Algebra Subprograms*. Aquesta API dóna suport a operacions bàsiques d'àlgebra lineal com la multiplicació de vectors i matrius, molt presents en el desenvolupament de xarxes neuronals artificials. El gran pes que tenen les matemàtiques en les xarxes neuronals profundes es veu molt alleugerit amb les funcions que ofereix aquesta API i de les que TensorRT fa ús en el procés d'inferència.

També cal tenir instal·lat al nostre sistema la llibreria cuDNN de NVIDIA. CuDNN és un conjunt de funcions optimitzades per a xarxes neuronals profundes, desenvolupades amb CUDA. Les funcions primitives de les xarxes neuronals com les convolucions o les funcions d'activació, estan altament optimitzades en aquesta ferramenta. La majoria de plataformes de codi obert per al desenvolupament de xarxes neuronals, com TensorFlow o Caffe, utilitzen les funcions de cuDNN per a la construcció de les seues aplicacions. [6]

Amb tots aquests programaris i una targeta NVIDIA, podem començar a optimitzar la inferència dels nostres models de xarxes neuronals. El sistema operatiu utilitzat és un Linux Ubuntu 18.04, la versió de CUDA 11.2 i la versió de TensorRT 8.0.0.

4.2.2. Procés d'optimització

TensorRT está desenvolupat per crear motors d'inferència de xarxes neuronals amb facilitat d'ús. Partint d'una xarxa neuronal artificial entrenada, el programari permet comprimir, optimitzar i desplegar la xarxa neuronal amb un temps d'execució reduït. Entre altres funcions, la ferramenta combina capes de la xarxa, optimitza els nuclis i realitza normalització i conversió a matrius optimitzades depenent de la precisió especificada per millor la latència, el rendiment i l'eficiència. TensorRT també ofereix la reducció dels pesos de la xarxa neuronal a precisió INT8 i FP16, aconseguint una latència significativament menor.

Les fusions de capes de les xarxes neuronals optimitzen els temps i l'espai que aquestes ocupen en memòria. Entre moltes altres, en optimitzar una xarxa amb TensorRT es realitzen les següents combinacions:

[5]Pàgina web de NVIDIA amb les especificacions de la GPU GeForce GTX 1050 https://www.nvidia.com/en-sg/geforce/products/10series/geforce-gtx-1050/

[6]Pàgina web de NVIDIA amb la informació de cuDNN https://developer.nvidia.com/cudnn

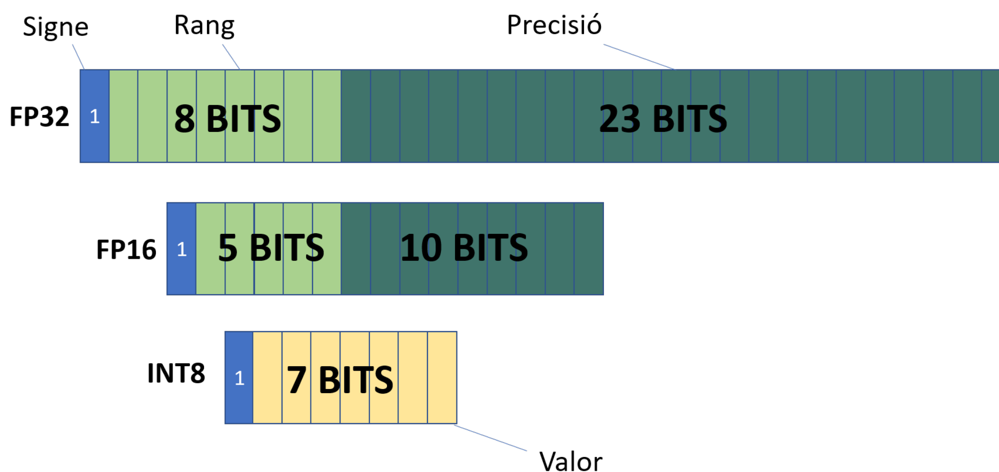

Figura 4.3: Mida en bits de les precisions FP32, FP16 i INT8.

- Capa convolució i capa ReLU: donades aquestes dues operacions, el programari s'encarrega que en una sola operació els resultats de la convolució siguen activats per la funció d'activació ReLU.
- Capa convolució i capa *pooling*: l'operació de convolució en moltes arquitectures està antecedida per l'operació de reducció, pooling. TensorRT aconsegueix que les dues funcions siguen una, afectant una gran quantitat de topologies amb aquest ordre.
- Capa ReLU i capa ReLU: en cas que una activació ReLU estiga precedida per una altra ReLU, aquestes passaran a ser una sola capa.

Per una altra part, a l'hora de construir un motor d'inferència de TensorRT podem optar perquè aquest utilitze precisió mixta. Podem triar entre FP32, FP16 o INT8 a l'hora d'elegir la precisió dels pesos de la nostra xarxa neuronal. *Half-precision floating-point format*(FP16) és un format de precisió molt adoptat per NVIDIA, on respecte al format *single precision* (FP32), es redueixen els bits de l'exponent i els bits de la fracció per a ocupar tan sols 16 bits en memòria. El format de coma fixa INT8, redueix encara més els bits necessaris per a representar els paràmetres de la xarxa neuronal, utilitzant soles 8 bits per a la representació dels pesos. Amb la tècnica de quantització es redimensionen els valors INT8 per a obtenir els valors reals dels tensors. En la figura 4.3 podem visualitzar les diferències de les precisions.

També destaquen altres formes amb les quals TensorRT optimitza els models de xarxes neuronals, com el *batching*. Per obtenir grans resultats de rendiment en la inferència, el millor en les GPU és processar tants resultats com siga possible en paral·lel. El programari de TensorRT està desenvolupat perquè en cada execució de la inferència es puguen computar molt resultats per cada lot d'execució.

Per fer l'optimització de xarxes neuronals cal seguir un flux de treball: entrenar una xarxa neuronal, convertir-la a un motor d'inferència de TensorRT i desplegar aquest motor sobre una plataforma NVIDIA. El primer pas d'entrenar la xarxa no involucra a TensorRT, podem utilitzar les diferents plataformes de desenvolupament de xarxes neuronals com TensorFlow o PyTorch per aquest pas. Cal especificar que TensorRT dona suport a les plataformes més comercials de desenvolupament de xarxes neuronals i als formats més utilitzats per aquestes. En aquest treball hem partit de xarxes neuronals profundes entrenades amb TensorFlow, igual que en la part d'optimització de models en FPGA amb ferramentes de Xilinx.

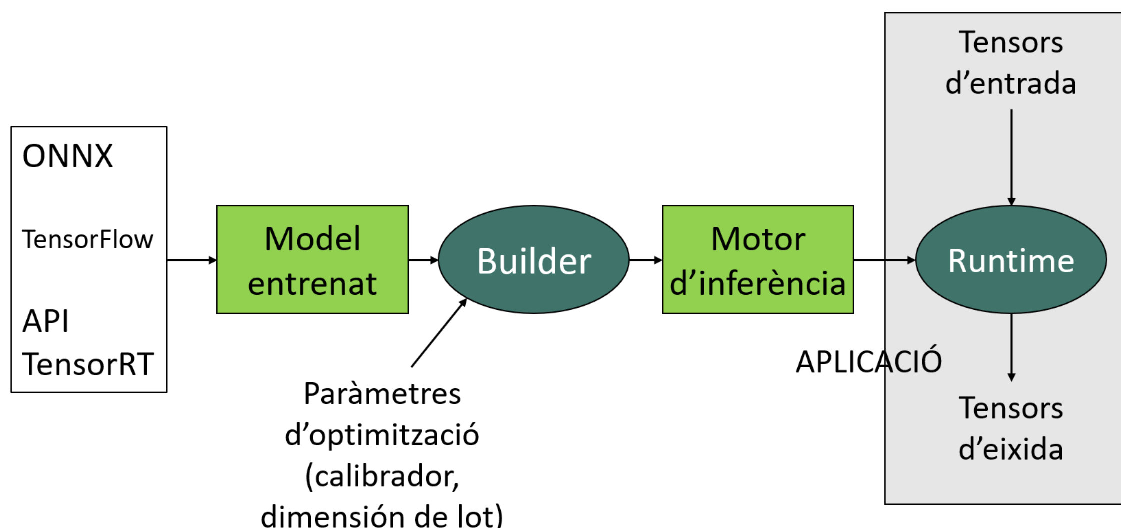

Figura 4.4: Flux d'optimització de models entrenats en TensorRT, on a partir del model entrenat, es crea un motor d'inferència amb un *Builder*, i en temps d'execució processa els tensors d'entrada per a produir els tensors d'eixida.

El motor d'inferència s'ha de construir a partir del model de la xarxa neuronal. Existeixen 3 formes de fer-ho: utilitzant una API de TensorFlow, TF-TRT, que ofereix les operacions bàsiques per a la conversió del model; conversió automàtica de xarxes especificades amb el format ONNX o construir una xarxa neuronal amb l'API de TensorRT. En aquest treball ens hem decidit per utilitzar la conversió automàtica amb el format ONNX, ja que ofereix menys sobrecàrrega que la conversió amb TF-TRT, compatible amb les xarxes que s'han utilitzat en el desenvolupament del treball. La Figura 4.4 representa el flux seguit.

El format *Open Neural Network Exchange* (ONNX)[7], és un estàndard per a la interoperabilitat de xarxes neuronals. Aquest format busca estandarditzar la representació de les xarxes neuronals per facilitar la interoperabilitat i el desplegament en maquinari d'aquests models. Suporta a les grans plataformes de desenvolupament com MATLAB o TensorFlow i ofereix desplegament dels models en plataformes que acceleren la inferència com TensorRT.

La conversió d'una xarxa neuronal entrenada amb TensorFlow al format ONNX es pot aconseguir amb la ferramenta tf2onnx, aquesta llibreria permet convertir models de Keras-TensorFlow a ONNX, en qualsevol dels diferents formats de guardar models de TensorFlow, com HDF5 o *frozen model*. Per exemple, si tenim un *freeze* model en format *protocol buffers*, on sabem els noms de la capa d'entrada i de la d'eixida, amb l'execució de:

```
python -m tf2onnx.convert --input model.pb --inputs nom/entrada:0 \
--outputs nom/eixida:0 --output model.onnx
```

Convertirem el nostre model *freeze* al estàndard ONNX. Una altra solució és guardar el model de Keras-TensorFlow a *SavedModel*. En acabar l'entrenament del nostre model, el guardarem en aquest format[8] i simplement amb l'execució de la següent ordre, obtindrem el model en format ONNX per a seguir el flux.

[7] Pàgina web d'ONNX amb els beneficis i especificacions del format. https://onnx.ai/ (visitada 26/06/2021)

[8] Com guardar un model de Keras TensorFlow 2 a *SavedModel* https://www.tensorflow.org/guide/saved_model (visitada 26/06/2021)

```
python -m tf2onnx.convert --keras ruta/a/saved-model \
--output model.onnx
```

El següent pas serà crear un motor d'inferència a partir de l'arxiu ONNX. Utilitzant l'API de python s'ha desenvolupat un codi que transforme els models al motor TensorRT. És en aquest pas on s'han d'especificar les característiques que optimitzaran la inferència de la xarxa, com la grandària de *batch* o la precisió utilitzada. En el nostre cas, treballarem amb la dimensió dels lots i amb la precisió utilitzada en la xarxa neuronal.

La gran quantitat de nuclis de la GPU permeten un alt paral·lelisme, així que per aprofitar més aquest paral·lelisme en les xarxes neuronals processarem diversos resultats al mateix temps. Per exemple, en una xarxa neuronal especialitzada en la classificació d'imatges, alimentar l'entrada del model amb més d'una imatge ens donarà els resultats de totes les imatges en l'eixida. Com hem comentat anteriorment, el nombre de dades amb què alimentem la xarxa neuronal a l'hora d'executar-la es coneix com a dimensió de *batch* o mida dels lots.

Per una altra part, per poder utilitzar la precisió reduïda INT8, cal seguir un procés de calibració dels pesos i de les activacions de la xarxa neuronal. Per a açò, necessitarem un conjunt d'imatges utilitzades durant l'entrenament o la validació. Es crea un calibrador [9] que mitjançant el conjunt de dades seleccionat, configurarà els pesos de cada capa perquè s'escalen a la precisió reduïda INT8. Aquesta calibració ha de seguir el mateix procés de preprocessat d'imatges que al moment d'entrenament i posteriorment en el moment d'execució de la xarxa neuronal. Una volta amb el calibrador configurat, sols quedarà crear el motor a partir d'aquesta configuració i guardar-lo en un arxiu amb extensió PLAN.

Per facilitar aquest procés s'ha desenvolupat un codi que indicant les següents ordres es crea el motor d'inferència. S'ha d'indicar la ubicació de l'arxiu ONNX i el nom de l'arxiu d'eixida. En l'annex A.1.3 està el codi desenvolupat.

```
python buildEngine.py --onnx_file fitxer.onnx --plan_file motor_trt.plan
```

A continuació sols quedarà utilitzar el motor d'inferència per a l'execució de la xarxa neuronal en dispositius GPU.

4.2.3. Desplegament en GPU

Per poder realitzar la inferència en la GPU amb un motor de TensorRT cal seguir un procés de comunicació amb la targeta gràfica utilitzant CUDA. En el nostre cas, hem treballat amb Python i una API que facilita l'ús de CUDA en aquest llenguatge, PyCUDA[10]. El primer pas serà reservar uns buffers de memòria en la host i en la GPU, per als tensors d'entrada i eixida del nostre model. Amb l'API de pyCUDA com a `cuda`, i la llibreria de TensorRT com a `trt` i el nostre valor de `batch_size` = 1 el codi resultant és el següent:

```
 # Buffers del host amb les dimensions dels tensors d'entrada i eixida
h_input_1 = cuda.pagelocked_empty(
            batch_size * trt.volume(engine.get_binding_shape(0)),
            dtype=trt.nptype(trt.float32))
```

[9]API de Python de TensorRT, amb les característiques del calibrador utilitzat en la quantització a INT8 https://docs.nvidia.com/deeplearning/tensorrt/api/python_api/infer/Int8/EntropyCalibrator2.html (visitada 26/06/2021)

[10]Pàgina web de PyCUDA amb les característiques de l'API i la documentació https://pypi.org/project/pycuda/ (visitada 26/06/2021)

```
h_output = cuda.pagelocked_empty(
           batch_size * trt.volume(engine.get_binding_shape(1)),
           dtype=trt.nptype(trt.float32))

# Buffers de la GPU amb les mateixes dimensions
d_input_1 = cuda.mem_alloc(h_input_1.nbytes)
d_output = cuda.mem_alloc(h_output.nbytes)

# Stream de CUDA on es copiaran els resultats i es farà la inferència
stream = cuda.Stream()
```

Els *streams* de CUDA són una seqüència d'instruccions que s'executen en ordre en la GPU [33]. Diferents *streams* poden executar-se concurrentment si no són dependents entre ells, d'una manera similar a com s'executen les instruccions en el processador.

Amb aquesta primera assignació de memòria, sols queda executar inferència amb el model en la GPU. Per a açò, haurem de transferir les dades a la GPU, fer que el motor execute la inferència i, finalment, copiar els resultats de nou en el sistema amfitrió de la GPU. Cal preprocessar les dades amb el mateix processament seguit a l'etapa d'entrenament i a l'etapa de calibració dels pesos, en cas d'haver fet la reducció de precisió. Les següents línies de codi especifiquen l'explicat, on pyCUDA està importada a `cuda`, el motor d'inferència està en la variable `engine` i les variables anteriors són efectives amb el contingut del codi anterior:

```
with engine.create_execution_context() as context:
  # Copiar dades a la GPU
  cuda.memcpy_htod_async(d_input_1, h_input_1, stream)

  # Executa inferència
  context.profiler = trt.Profiler()
  context.execute(batch_size=1, bindings=[int(d_input_1), int(d_output)])

  # Recupera les prediccions de la GPU
  cuda.memcpy_dtoh_async(h_output, d_output, stream)

  # Sincronització del stream
  stream.synchronize()
```

I així tindrem els resultats de l'execució de la inferència en el nostre sistema amfitrió. Per exemple, suposem una xarxa neuronal especialitzada en la classificació d'imatges, entrenada amb un conjunt de dades capaç de detectar 10 classes diferents i on l'última capa és la funció d'activació `Softmax`. En executar el codi que utilitza TensorRT, en els buffers resultats tindrem un tensor (vector) de longitud 10x1 on el valor màxim és la classificació realitzada per la xarxa neuronal. En cas de tenir una dimensió dels lots major, en alimentar el tensor d'entrada amb `N` imatges, en la capa d'eixida tindrem un vector amb dimensió (10*N)x1, on dividirem els resultats en `N` subvectors amb els resultats de la predicció del model. En l'annex A.1.4 està el codi desenvolupat per a inferir 1 imatge amb TensorRT.

CAPÍTOL 5
Implementació de noves funcionalitats en HELENNA

HELENNA és una plataforma de programari per al desenvolupament de processos d'entrenament i inferència de xarxes neuronals artificials. HELENNA és l'acrònim de *HEterogeneous LEarning Neural Networks Application*. L'objectiu de l'aplicació és l'ús d'arquitectures heterogènies de còmput.

5.1 Descripció de la plataforma

Malgrat que existeixen diverses solucions per a l'entrenament i inferència de xarxes neuronals com les que hem estat veient en aquest treball, HELENNA s'ha desenvolupat per a explorar nous mètodes d'entrenament i noves xarxes neuronals en el context de projectes d'investigació. L'ús d'una plataforma desenvolupada exclusivament pel grup d'investigació GAP permet un major control sobre els processos d'entrenament i inferència i, per tant, una millor adaptabilitat per a l'estudi i anàlisi de processos alternatius que milloren el seu rendiment [11].

L'aplicació està construïda majoritàriament en el llenguatge de programació C, si bé amb mòduls i seccions amb C++, CUDA, CLBLAS, i té una estructura de codi similar a la programació orientada a objectes encara que no s'utilitzen com a tal. En concret, HELENNA permet la definició de capes de xarxes neuronals aïllant el codi de la capa en un únic fitxer i implementant diferents funcions bàsiques per al suport de l'entrenament i la inferència de la capa. Des d'aquest punt de vista, crear el suport per a una nova capa s'ha de fer amb la creació d'un nou fitxer i la indicació de les funcions que la capa ha d'implementar. Cada capa suportada en HELENNA ha d'implementar cinc funcions bàsiques:

- Funció `parse`. Aquesta funció interpreta els paràmetres de definició de la capa i inicialitza les seues estructures.
- Funció `allocate`. Amb aquesta funció la capa crea els buffers (tensors) adequats per al suport de la capa.
- Funció `initialize`. En aquesta funció s'inicialitzen els buffers (tensors) de la capa.
- Funció `forward`. Aquesta funció realitza el procés de *forward* (inferència) de la capa.
- Funció `backpropagation`. Amb aquesta funció es calculen els gradients de la capa per a la seua posterior aplicació sobre els pesos.

- Funció `propagate_error`. En aquesta funció es calcula l'error de l'entrada de la capa a partir de l'error d'eixida de la funció.

Les tres últimes funcions serveixen per a realitzar el procés d'entrenament de la capa dins de l'estructura de la xarxa neuronal.

Un altre aspecte d'HELENNA és la gestió de memòria i l'abstracció de la ubicació dels buffers de memòria que s'utilitzen en un procés d'entrenament o inferència. Els tensors es creen de forma dinàmica en HELENNA i s'ubiquen en el dispositiu seleccionat (CPU, GPU, FPGA...). L'usuari no és conscient en el moment de la instanciació i en l'ús dels buffers d'on estan ubicats. HELENNA realitza internament les transferències necessàries per a una correcta utilització dels tensors/buffers. Cada tensor creat en HELENNA s'identifica amb un identificador únic i cada acció sobre el tensor utilitza exclusivament l'identificador.

HELENNA permet l'entrenament en CPU i GPU amb diferents optimitzacions, principalment el suport de biblioteques de càlcul matricial. Actualment HELENNA permet els següents dispositius:

- CPU. Aquest dispositiu utilitza la CPU i tots els seus núclis en l'entrenament i la inferència. No s'utilitza cap biblioteca de càlcul matricial. Aquest dispositiu és el bàsic.

- MKL. Amb aquest dispositiu s'utilitza la CPU juntament amb la llibreria de càlcul matricial MKL d'Intel. Aquesta llibreria explora el màxim rendiment dels processadors Intel i, com a tal, sols es pot utilitzar en aquestes CPU.[1]

- CuBLAS. En aquest dispositiu s'utilitza la GPU de NVIDIA. En concret, s'utilitza la llibreria matemàtica CuBLAs de NVIDIA per al càlcul matricial.

- CuDNN. La llibreria d'aprenentatge profund de NVIDIA també està suportada en HELENNA, donant suport a l'entrenament i inferència en HELENNA. Aquesta llibreria funciona sobre les GPU de NVIDIA i millora significativament les prestacions de les xarxes neuronals.

- OpenCL-FPGA. Amb aquest dispositiu s'utilitzen sistemes basats en FPGA per al procés d'inferència. En concret, aquest dispositiu permet utilitzar mòduls generats per a FPGA específics com convolucions i funcions d'activació.

Actualment, HELENNA suporta una varietat de capes de xarxes neuronals. Entre altres es suporten la capa convolucional, la capa `fully_connected` o *MultiLayer Perceptron*, la *batch normalization*, capes de funcions d'activació com la ReLU, la sigmoide o la softmax i moltes altres capes disponibles per dissenyar xarxes neuronals artificials.

A més, HELENNA suporta l'entrenament i inferència els conjunt de dades més populars dins del món de la IA, com són el conjunt MNIST[2], CIFAR10, CIFAR100[3] i Imagenet, que hem estat utilitzant en aquest treball fins ara.

En aquest projecte busquem donar suport a noves capes de funcions d'activació en HELENNA, explorarem alternatives a l'activació linear i la més comuna, la ReLU, que puguen donar millors resultats en models entrenats amb diferents conjunts de dades.

[1] Especificacions de la llibreria MKL d'Intel https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html

[2] Base de dades de MNIST http://yann.lecun.com/exdb/mnist/index.html

[3] Base de dades de CIFAR10 i CIFAR100 https://www.cs.toronto.edu/~kriz/cifar.html

5.2 Funcions d'activació

Les funcions d'activació afecten directament a la propagació de cada neurona, normalitzant la seua eixida i transmitint la informació per l'eixida. S'han implementat noves funcions en la plataforma HELENNA, per poder experimentar i veure el comportament de les xarxes neuronals en diferents activacions en els seus nodes.

5.2.1. Suport a noves funcions d'activació en HELENNA

L'entorn HELENNA està construïda en C principalment, on els membres del GAP han aconseguit emular el procés d'entrenament i inferència de xarxes neuronals artificials. Com ja s'ha comentat, l'entrenament d'una xarxa neuronal té dues parts principals, la propagació i la retropropagació. En la propagació les dades d'entrenament recorren tota la xarxa fins que el model computa una predicció. En la retropropagació s'utilitza aquesta predicció per calcular la diferència respecte al resultat esperat, i a partir d'ací es recorre la xarxa neuronal des del final fins al principi, on s'actualitzen els pesos de la xarxa utilitzant tècniques d'ajustament dels paràmetres, on s'utilitzen les derivades dels paràmetres [6]. És per açò que a l'hora d'especificar noves funcions d'activació en HELENNA, l'operació principal a implementar és la funció matemàtica de la funció i la seua derivada.

Per a crear una nova funció d'activacó en HELENNA cal seguir diversos passos. Les capes en HELENNA estan distribuïdes en un arxiu C, amb la implementació de la capa i un arxiu de capçalera (`arxiu.h`), amb la declaració dels mètodes que implementa cada capa. Els passos a seguir per implementar una nova funció d'activació en HELENNA són els següents:

1. Afegir la definició de la capa en `globals.h`. L'arxiu `globals.h` conté constants, estructures i variables globals que s'utilitzen en l'aplicació, com el tipus de cada capa.

2. Crear l'arxiu C i la capçalera, arxiu que implementarà les funcions bàsiques que implementa cada capa en HELENNA.

3. Afegir l'arxiu C a l'arxiu de compilació `CMakeLists.txt`. L'arxiu de `CMakeLists.txt` s'encarrega de compilar tot el projecte i de generar l'executable final que fa funcionar l'aplicació. Qualsevol capa o funció creada s'ha d'especificar ací dins perquè en temps de compilació es considere.

4. En `topology.c`, les funcions `fn_layer_set_functions` i `fn_parse_topology` es modificaran per indicar un nom al tipus de capa, en el nostre cas `LAYER_nom` on el nom serà la funció d'activació que es crea.

5. En l'arxiu `fit.c`, en la funció `fn_SGD` s'haurà d'indicar els paràmetres que s'actualitzen de la capa, si hi ha. Aquesta funció s'encarrega de l'actualització de paràmetres durant la fase de retropropagació.

6. Definir les operacions aritmètiques de la funció en `arithmetic.c`. Es crearan dos mètodes, que actualitzaran tots els pesos d'entrada, aplicant la funció d'activació o la derivada depenent del mètode. En l'arxiu C de la funció es cridaran aquestes funcions des de `fn_forward` la funció d'activació i des de `fn_propagate_error` la funció d'activació derivada, per actualitzar paràmetres que s'utilitzen en l'actualització dels pesos.

Cal destacar que des de l'arxiu `arithmetic.c`, els mètodes que implementen les funcions d'activació, criden a altres mètodes que realitzen la funció d'activació o la derivada en el dispositiu corresponent que ha sigut seleccionar al començar el procés d'entrenament o inferència. El codi que implementa cada funció ha d'estar en els arxius que tenen les operacions que realitza el dispositiu, com `cpu.c`, `cublas.c`, `gpu.c`, `mkl.c`. A més, en el cas de les implementacions en GPU, el codi que executaran els kernels de GPU ha d'estar o en `cublas_kernels.cu` o `clblas_kernels.cl`, especificats en el llenguatge CUDA o OpenCL, on s'implementa el codi per a funcionar paral·lelament en GPU.

En aquest treball s'han implementat totes les funcions per funcionar en CPU, amb una implementació en llenguatge C; Intel MKL, la llibreria matemàtica per a processadors Intel; CLBLAS, llibreria matemàtica d'OpenCL per a utilitzar en GPU i CUBLAS, llibreria matemàtica d'NVIDIA que dóna suport a GPU de NVIDIA.

L'objectiu d'implementar noves funcions és millorar la precisió final dels models neuronals i experimentar amb quines funcions ofereixen un millor rendiment en les mateixes situacions.

S'han implementat les funcions ReLU, Leaky ReLU, Parametric ReLU, ELU, Swish i GELU en l'aplicació, En l'annex A.2 estan les implementacions per a funcionar en cpu, totes en l'arxiu `cpu.c`. A continuació s'expliquen les característiques de cada funció.

ReLU

Una de les funcions més utilitzades en el context de l'aprenentatge profund és la ReLU (Unitat Lineal Rectificada). En 2011 un grup d'investigadors va descobrir que les rectificadores entrenaven millor els models respecte a les funcions utilitzades fins al moment, com la sigmoide [7]. La seua funció i la seua derivada la podem veure a continuació:

$$f(x) = max(0, x)$$

$$f'(x) = \frac{dy}{dx} = \begin{cases} 1, & \text{si } x > 0 \\ 0, & \text{altrament} \end{cases}$$

Un dels avantatges principals que té aquesta funció és que, el resultat de qualsevol neurona que siga menor que 0, passa a ser 0. Açò s'ha demostrat ser beneficiós per a l'entrenament, ja que un model amb gran quantitat de neurones a 0, activarà menys neurones en el moment de propagació, açò redueix la complexitat dels models profunds. A més, és eficient en l'àmbit de computació, ja que sols es realitza una comparació i una assignació a l'hora de calcular l'eixida.

Però, aquesta funció també pot ocasionar problemes. Si es dóna el cas que moltes neurones no s'activen, és a dir, sols donen com a resultat 0, en la retropropagació no s'actualitzaran els valors dels pesos i una gran part de la xarxa neuronal passa a estar inactiva. En el pitjor dels casos la xarxa neuronal pot arribar a morir i deixar d'aprendre, on es convertirà en una funció constant. Aquest problema es coneix com a *Dying ReLU*.

És una situació que és difícil que es done, ja que arribar a una situació tan extrema és difícil. Sempre que hi haja neurones actives, la xarxa neuronal pot continuar aprenent. No obstant això, amb les següents funcions s'exploren formes d'evitar que açò passe i d'augmentar la precisió dels nostres models d'aprenentatge profund.

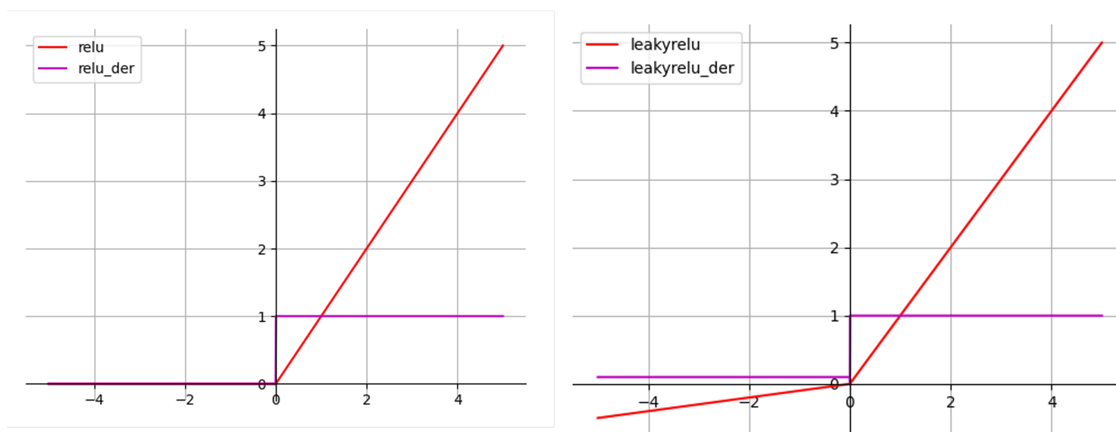

Figura 5.1: A l'esquerra: funció ReLU i la seua derivada. A la dreta, funció Leaky ReLU i la seua derivada.

Leaky ReLU

La Leaky ReLU és una funció proposada per investigadors de la universitat de Stanford en 2013. Aquesta funció és una variant de la rectificadora ReLU, que busca evitar el problema de *Dying ReLU*. Per a assolir-ho, la part negativa de la funció és controlada per un valor auxiliar, que multiplica els resultats per aquest valor i permet a les neurones continuar activant-se malgrat que tinguen resultats negatius [16]. Les fórmules són les següents, on el valor auxiliar té el valor α:

$$f(x) = \begin{cases} x, & \text{si } x > 0 \\ \alpha x, & \text{altrament, on } \alpha > 0 \end{cases}$$

$$f'(x) = \frac{dy}{dx} = \begin{cases} 1, & \text{si } x > 0 \\ \alpha, & \text{altrament, on } \alpha > 0 \end{cases}$$

Aquesta funció s'ha implementat en HELENNA amb la possibilitat d'elegir el valor auxiliar per als valors negatius de les neurones. A l'hora de definir un nou arxiu amb les especificacions d'un model de xarxa neuronal, podrem indicar de la següent forma el valor del paràmetre auxiliar, on α pot ser qualsevol valor de $[0, +\infty)$:

```
nom_capa leakyrelu 0.1
```

En la figura 5.1 podem veure representades la ReLU i la Leaky ReLU, on s'aprecia la diferència en els valors negatius.

Parametric ReLU

La funció Parametric ReLU va ser dissenyada en 2015 per un grup d'investigadors de Microsoft. És una variació de la ReLU i de la Leaky ReLU, però amb la gran diferència que introdueix un paràmetre α que s'aprén durant la fase d'entrenament. La idea és la mateixa que en la funció Leaky ReLU, però el paràmetre auxiliar, alfa, no és una constant sinó un valor que s'aprén mitjançant la retropropagació en l'entrenament [9]. La fórmula és la mateixa que per a la funció Leaky ReLU:

$$f(x) = \begin{cases} x, & \text{si } x > 0 \\ \alpha x, & \text{altrament} \end{cases}$$

$$f'(x) = \frac{dy}{dx} = \begin{cases} 1, & \text{si } x > 0 \\ \alpha, & \text{altrament} \end{cases}$$

En finalitzar cada lot en l'entrenament, s'actualitzarà el valor α al mateix temps que els pesos de la xarxa neuronal. Per a aconseguir-ho s'utilitza el gradient de l'activació. El gradient de l'activació és definit per la següent fórmula:

$$\frac{\partial f(y_i)}{\partial a_i} = \begin{cases} 0, & \text{si } y_i > 0 \\ y_i, & \text{altrament} \end{cases}$$

On cada y_i és una neurona de la capa. L'objectiu final és calcular el gradient del paràmetre α, que s'utilitza per actualitzar α. Per a aconseguir açò, es necessita el sumatori de tots els gradients d'activació d'una capa multiplicat pels gradients propagats de la capa anterior. Aquest segon paràmetre el tenim emmagatzemat en HELENNA en un buffer, accessible des d'un punter de memòria.

Per a la implementació en HELENNA s'ha reservat memòria per a un tensor de la longitud de les neurones per a emmagatzemar el gradient de l'activació i també per a un sol valor alfa, per mantenir actualitzat el valor en cada capa d'activació. El gradient d'activació es calcula al mateix temps que es realitza la propagació de les dades. Després, s'utilitza en la retropropagació. S'han afegit les operacions necessàries en la funció `fn_backward` de la capa PReLU, per a calcular el gradient d'alfa per a la capa, on es realitza el sumatori. Després, s'ha utilitzat aquest valor per actualitzar el valor alfa en la funció `fn_SGD` de `fit.c`.

L'actualització d'aquest valor té un cost computacional relativament menut comparat amb l'actualització dels pesos i altres operacions que es realitzen en la xarxa neuronal. No obstant això, en la majoria de casos el valor alfa convergeix a 0, convertint-se en una funció rectificadora ReLU amb un entrenament prolongat.

ELU

La funció Unitat Lineal Exponencial, ELU, és una funció introduïda per investigadors Austríacs de bioinformàtica. Aquesta funció té la mateixa característica lineal que la ReLU, Leaky ReLU i PReLU per als valors positius, però amb la diferència en la ReLU de tenir valors negatius que prolonguen l'activació de les neurones com les altres dues funcions. La gran diferència amb les funcions Leaky ReLU i PReLU, és que l'ELU se satura a un valor negatiu a partir de cert valor, reduint la influència que podent tenir valors negatius alts en la propagació dels resultats [4]. La funció matemàtica i la seua derivada són les següents:

$$f(x) = \begin{cases} x, & \text{si } x > 0 \\ \alpha(e^x - 1), & \text{altrament, on } \alpha > 0 \end{cases}$$

$$f'(x) = \frac{dy}{dx} = \begin{cases} 1, & \text{si } x > 0 \\ f(x) + \alpha, & \text{altrament, on } \alpha > 0 \end{cases}$$

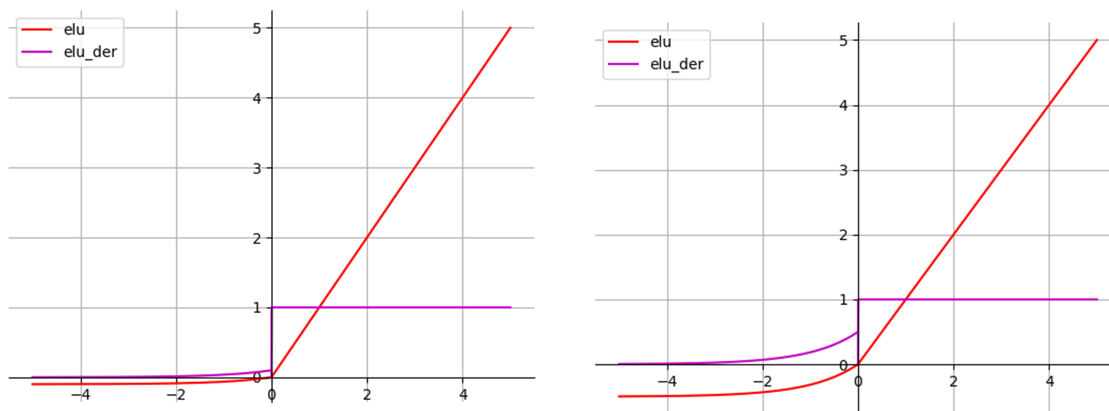

Figura 5.2: A l'esquerra: funció ELU i la seua derivada amb $\alpha = 0.1$. A la dreta, funció ELU i la seua derivada amb $\alpha = 0.5$.

Aquesta funció és més robusta enfront del soroll. Les dades sorolloses poden ser mostres errònies o dades estranyes que ocasionen problemes a l'hora de l'entrenament de la xarxa neuronal. Amb un limitador en els valors negatius es redueix aquest fenomen.

En la implementació en HELENNA s'ha introduït la mateixa característica que la Leaky Relu, sent possible seleccionar el valor alfa (α) amb el que s'inicialitza la funció. En la figura 5.2 podem veure dues variants de la funció ELU amb diferents valors d'alfa.

Swish

La Swish és una funció d'activació proposada en 2017 per investigadors de Google amb l'objectiu de millorar les prestacions i la precisió de les funcions d'activació [21]. La funció Swish destaca per tenir la mateixa funció tant per a la part positiva com per a la negativa, a diferència de les vistes fins ara. Aquesta característica fa que no siga completament lineal en la part positiva de la funció, però, a mesura que els resultats són més positius s'apropa a la funció lineal. Com totes les funcions estudiades fins al moment diferents de ReLU, busca solucionar el problema de *Dying ReLU* amb la incorporació de valors d'activació negatius. A més, a mesura que augmenten els valors negatius, la funció torna a convergir a 0. Aquesta característica la fa més estable enfront del soroll, com la funció ELU.

La funció matemàtica de Swish i la seua derivada les podem veure a continuació, junt amb una representació gràfica en la figura 5.3. Com podem observar, aquesta funció té un cost computacional major a les anteriors, ja que implica més operacions matemàtiques. Açò pot derivar en problemes si no es té suficient potència de còmput.

$$f(x) = x(1 + e^{-x})^{-1}$$

$$f'(x) = \frac{dy}{dx} = f(x) + (1 + e^{-x})^{-1}(1 - f(x))$$

GELU

La funció GELU va ser introduïda per investigadors nord-americans en 2016. Aquesta funció és la més matemàticament complexa, ja que utilitza la funció d'error de Gauss. Amb l'ús d'aquesta funció, l'activació té unes característiques molt paregudes a la funció

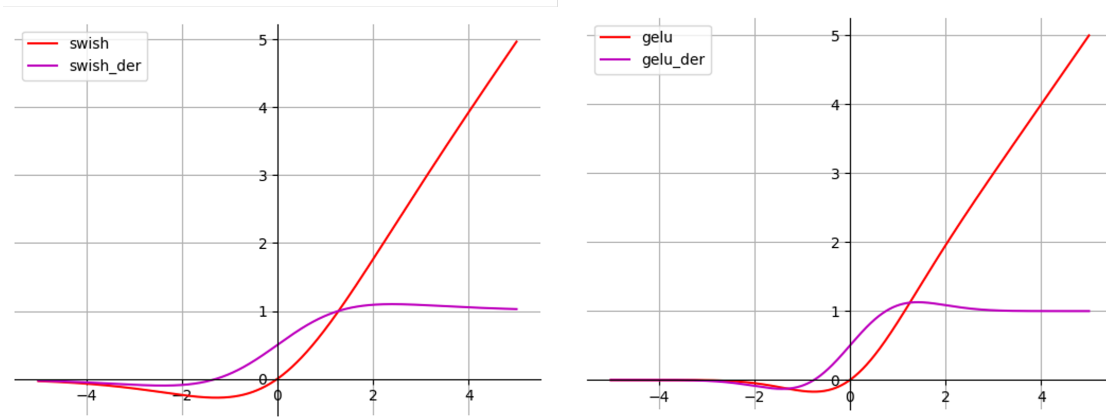

Figura 5.3: A l'esquerra: funció Swish i la seua derivada. A la dreta, funció GELU i la seua derivada.

Swish: no és lineal, però amb valors molt positius s'apropa a la linealitat i la part negativa evita el problema de *Dying ReLU*, però torna a convergir a 0 en adoptar valors molt negatius.

La complexitat de la funció d'error de Gauss fa que s'utilitzen aproximacions en alguns àmbits, per limitacions del maquinari o de còmput. La funció d'error de Gauss és la següent:

$$erf(z) = \frac{2}{\sqrt{\pi}} \int_z^0 e^{-t^2} \, dt$$

La funció d'error s'utilitza en la funció GELU i la seua derivada de la següent forma:

$$f(x) = x\frac{1}{2}\left[1 + erf\left(\frac{x}{\sqrt{2}}\right)\right]$$

$$f'(x) = \frac{dy}{dx} = \frac{1 + erf\left(\frac{x}{\sqrt{2}}\right)}{2} + x\left(\frac{1}{\sqrt{2\pi}} e^{-0.5} x^2\right)$$

Com podem observar, la complexitat matemàtica és molt alta i el cost computacional també, respecte a altres capes com la ReLU. No obstant això, és una de les capes que més popularitat estan prenent en els últims anys i els seus resultats en l'aprenentatge profund l'han portat a ser implementada i estudiada en aquest treball [12].

CAPÍTOL 6
Avaluació dels resultats

A continuació s'analitzen els resultats obtinguts amb les diferents ferramentes i implementacions realitzades en aquest treball. Per a avaluar cada apartat s'utilitzaran models de xarxes neuronals amb característiques diferents i diferents conjunts de dades.

Topologies utilitzades

Les topologies utilitzades són la `Resnet`, la `Inception`, la `VGG` i la `AlexNet`. Són topologies diferents presentades per investigadors de diferents companyies, com Google o Microsoft o institucions com la universitat d'Oxford. Aquestes xarxes van aparéixer per primera volta en diferents edicions del concurs anual que organitzen els creadors de la base de dades ImageNet, on es competeix per veure quin model de xarxa neuronal aconsegueix més precisió en classificar els 15 milions d'imatges que té la base de dades (ILSVRC) [24].

El model `Resnet` va ser presentat en 2015 per investigadors de Microsoft, on van introduir el concepte d'aprenentatge residual. Els resultats d'una capa no tan sols van a la capa següent, sinó que també van al resultat d'unes quantes capes posteriors, afegint-se a aquest. Així es permeten fer moltes operacions sobre les dades (convolucions, redimensions) sense perdre molta precisió, ja que van arreplegant el "residu"de capes anteriors. La idea bàsica la podem observar en la figura 6.1. En aquest treball s'han utilitzat dues versions amb 50 capes de profunditat, `Resnet50`, i amb 101 capes de profunditat, `Resnet101` [10]. Aquest model va guanyar la competició d'ILSVRC en 2015.

L'arquitectura `Inception` va ser presentada en 2014 per investigadors de Google, amb una versió anomenada `GoogleNet`. Proposaven una nova forma d'estructurar les xarxes neuronals, amb l'estructuració en mòduls *inception*. Aquests mòduls són combinacions de capes, que permeten a la xarxa créixer en amplitud en executar diferents operacions de convolució en els mateixos pesos, que després són concatenats en un tensor resultat [28]. En aquest treball s'han utilitzat la versió `InceptionV1`, amb mòduls *inception* com els mostrats en la figura 6.2 i una profunditat de 22 capes, i la versió `InceptionV4`, que utilitza mòduls més complexes amb altres combinacions de capes i té una profunditat de

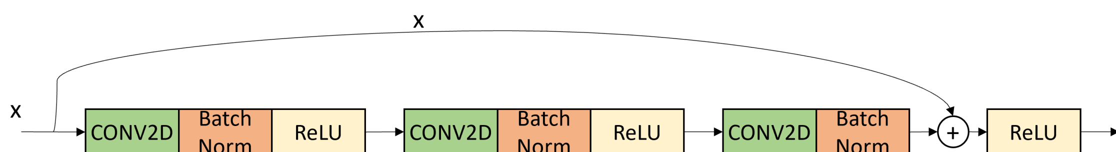

Figura 6.1: Concepte d'aprenentatge residual en les xarxes `Resnet`

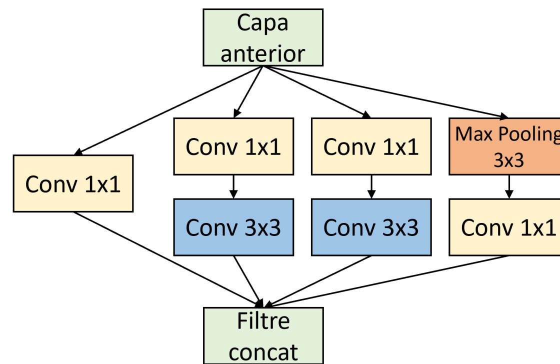

Figura 6.2: Mòdul *inception* amb reduccions de dimensió.

75 capes. Aquest model va demostrar els seus bons resultats guanyant l'edició de 2014 de ILSVRC.

Les topologies `VGG` i `Alexnet` són més simples comparades amb les dues anteriors, ja que simplement són capes concatenades sense cap implementació addicional. Una operació darrere d'una altra per processar les dades i entrenar els pesos de la xarxa. La `VGG16` té 16 capes de profunditat i la `VGG19` té, com el seu nom indica, 19 capes de profunditat [26]. L'arquitectura `AlexNet` va guanyar l'edició d'ILSVRC de 2012, mentre que la `VGG16` va quedar segona en l'any 2014.

En l'annex B tenim més informació sobre aquestes xarxes. En les ferramentes Vitis AI i TensorRT hem utilitzat les xarxes `Resnet50`, `Resnet101`, `InceptionV1`, `InceptionV4`, `VGG16` i `VGG19`. En l'estudi de funcions d'activació en HELENNA s'han utilitzat les xarxes `VGG16` i `AlexNet`.

6.1 Resultats amb Vitis AI

Per a l'experimentació s'ha seguit el flux de treball de Vitis AI per a desplegar models en FPGA, on el model es quantitza, la versió quantitzada es compila per generar un model capaç de ser desplegat en la targeta i, finalment, s'executa en la targeta en el procés d'inferència. Els models utilitzats són dues versions de la xarxa `Resnet`, dues versions de la xarxa GoogleNet o `Inception` i dues versions de la xarxa `VGG`.

Aquests models han sigut entrenats en la base de dades ImageNet i estan especialitzats en la classificació d'imatges. Són capaços de predir fins a 1000 classes diferents. Per a l'experimentació utilitzarem 5000 imatges del conjunt de dades de validació d'ImageNet. Cada model requereix un processament previ de les imatges diferent, en l'annex A.1.1 està el codi utilitzat `input_fn.py`, on les `Resnet` i `vgg` utilitzen `preprocess_imagenet`, `inceptionv1` utilitza `preprocess_inception` i `inceptionv4` fa ús de `preprocess_inceptionv4`.

S'ha realitzat inferència amb els models en CPU amb precisió de 32 bits, utilitzant la ferramenta TensorFlow i en FPGA, fent ús de Vitis AI i amb precisió de 8 bits. Tots els resultats han sigut obtinguts amb una FPGA Alveo U200 i comparats amb els mateixos experiments en una CPU Intel Core i7-7700 HQ amb una freqüència de funcionament de 3,80 GHz. Els experiments realitzats són els següents:

- **Precisió FP32 vs. INT8**: En aquesta primera prova s'utilitzaran les 5000 imatges per calcular la precisió dels models executats en la CPU i en la FPGA.

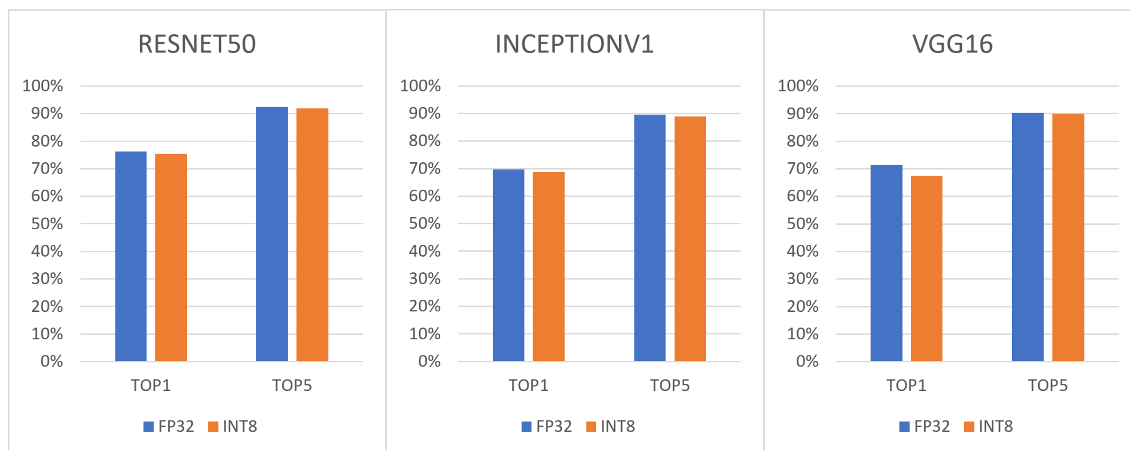

Figura 6.3: Diferència de top-1 i top-5 en precisió FP32 i INT8, en 3 dels models utilitzats en CPU i FPGA.

- **Latència CPU vs. FPGA**: 5 imatges diferents són inferides amb els models en CPU i FPGA, on mesurem el temps utilitzat per cada arquitectura.
- **Capacitat de còmput en FPGA**: Explorarem el paral·lelisme que ofereix la FPGA en fer inferència amb les 5000 imatges amb diferents dimensions de lot. El lot indica quantes imatges s'executen paral·lelament per execució.
- **Rendiment CPU vs. FPGA**: Mesurarem el temps utilitzat per cada arquitectura en processar i inferir en 5000 imatges.

Una de les característiques de la quantització és que al reduir la dimensió dels pesos i activacions, es perd precisió. Els models han sigut quantitzats amb 300 iteracions i 500 imatges, perquè la ferramenta Vitis AI obtinga les estadístiques necessàries per a poder funcionar en FPGA amb precisió reduïda. Per a mesurar la precisió utilitzarem les mètriques top-1 i top-5. El top-1 és el nombre de voltes que la predicció del model ha coincidit amb la predicció esperada de la imatge i top-5 és el nombre de voltes que la predicció correcta ha estat entre les 5 primeres prediccions del model. En la figura 6.3 podem observar la diferència entre la precisió FP32 i INT8 en tres dels sis models estudiats. Com podem observar, s'aprecia una reducció de la precisió en la versió quantitzada, tal com esperàvem, on la precisió top-1 dels models sol estar al voltant del 70%, i la de top-5 sobre el 90%. En la tabla 6.1 tenim la diferència entre top-1 de FP32 i INT8, i el mateix per a top-5. Els termes `LOSS1` i `LOSS5`, representen la pèrdua de precisió d'INT8 respecte a FP32.

Tal com era d'esperar, en tots els casos estudiats hi ha una pèrdua de precisió. Això no obstant, la pèrdua de precisió és menor en la predicció top-5 ($0\% - 1\%$), el que significa que la majoria de voltes que el model falla, la predicció correcta està relativament prop. En alguns casos com en la `VGG16`, la precisió top-1 pot arribar a perdre fins a quasi 4%, però com la precisió top-5 tan sols té una pèrdua del 0.3%, així que entre les 5 primeres prediccions del model.

En el següent experiment estudiem la latència dels models, realitzant inferència sobre una imatge aleatòria de les 5000 mostres de validació que tenim. Es mesurarà el temps que tarda el maquinari a realitzar una predicció amb cada model d'aprenentatge profund. A la figura 6.4 podem observar la diferència en la inferència de 5 imatges sobre CPU i FPGA amb tres de les sis xarxes neuronals. A simple vista sabem que l'execució en FPGA és molt més ràpida, amb centèssimes de segon, enfront de les execucions de CPU, que estan en les dècimes de segon. En la tabla 6.2 podem observar la mitjana de la latència

	RESNET50		RESNET101		VGG16	
	CPU_{FP32}	$FPGA_{INT8}$	CPU_{FP32}	$FPGA_{INT8}$	CPU_{FP32}	$FPGA_{INT8}$
TOP1	76.24%	75.44%	76.74%	75.32%	71.32%	67.54%
TOP5	92.38%	91.98%	92.66%	92.12%	90.26%	89.96%
LOSS1	0.80%		1.42%		3.78%	
LOSS5	0.40%		0.54%		0.30%	
	INCEPTIONV1		INCEPTIONV4		VGG19	
	CPU_{FP32}	$FPGA_{INT8}$	CPU_{FP32}	$FPGA_{INT8}$	CPU_{FP32}	$FPGA_{INT8}$
TOP1	69.76%	68.76%	79.84%	79.28%	70.90%	70.54%
TOP5	89.63%	88.92%	95.36%	94.69%	90.04%	89.76%
LOSS1	1.00%		0.56%		0.36%	
LOSS5	0.71%		0.67%		0.28%	

Taula 6.1: Diferència de la precisió top-1 i top-5 en inferència de 5000 imatges amb una CPU utilitzant TensorFlow amb precisió FP32 i una FPGA utilitzant Vitis AI amb precisió reduïda INT8, en 6 models de xarxes neuronals diferents.

	RESNET50		RESNET101		VGG16	
	CPU_{FP32}	$FPGA_{INT8}$	CPU_{FP32}	$FPGA_{INT8}$	CPU_{FP32}	$FPGA_{INT8}$
Temps (s)	0.55002	0.00896	0.85201	0.01200	2.94335	0.02776
Speedup	61.38630		71.00046		106.02853	
	INCEPTIONV1		INCEPTIONV4		VGG19	
	CPU_{FP32}	$FPGA_{INT8}$	CPU_{FP32}	$FPGA_{INT8}$	CPU_{FP32}	$FPGA_{INT8}$
Temps (s)	0.20664	0.01268	0.88558	0.03938	3.34081	0.02702
Speedup	16.29665		22.48814		123.64194	

Taula 6.2: Latència i *speedup* dels models en la inferència d'una imatge, utlitzant la FPGA amb Vitis AI o la CPU amb TensorFlow.

obtinguda en 5 execucions tant en CPU com en FPGA, a més de l'acceleració o *Speedup* definida per:

$$Speedup = \frac{L_1}{L_2}$$

On L1 és la latència en CPU i L2 la latència en FPGA. Podem observar com en tots els casos s'obté una millora de l'ordre de fins a 123 voltes més ràpid, en la xarxa `VGG19`, on l'execució en CPU és molt costosa. Les xarxes `INCEPTION` no milloren tant com les altres topologies, a pesar que l'execució en FPGA és quasi i més de 20 voltes més ràpida en el model quantitzat. La versió més ràpida és la `Resnet50` quan s'executa en la FPGA.

L'experimentació següent ha consistit en la inferència de 5000 imatges en la FPGA amb diferents dimensions de lot. La dimensió de lot o *batch size*, determina el nombre de mostres processades en paral·lel o al mateix temps en el maquinari, on s'obtindran totes les prediccions del lot al mateix temps. Hem executat amb dimensions 1, 2, 4 i 8. Les execucions amb nombres de lot superiors a 8 han resultat en fallides, per limitacions de memòria en la targeta. En la figura 6.5 podem vore el temps utilitzat en tres models diferents per a la inferència de 5000 imatges. Com podem observar, el rendiment augmenta a mesura que la dimensió del lot es major. La inferència en lot permet superposar el procés de transferència de dades que es realitza de CPU a FPGA i per tant maximitzar el rendiment del model a la FPGA.

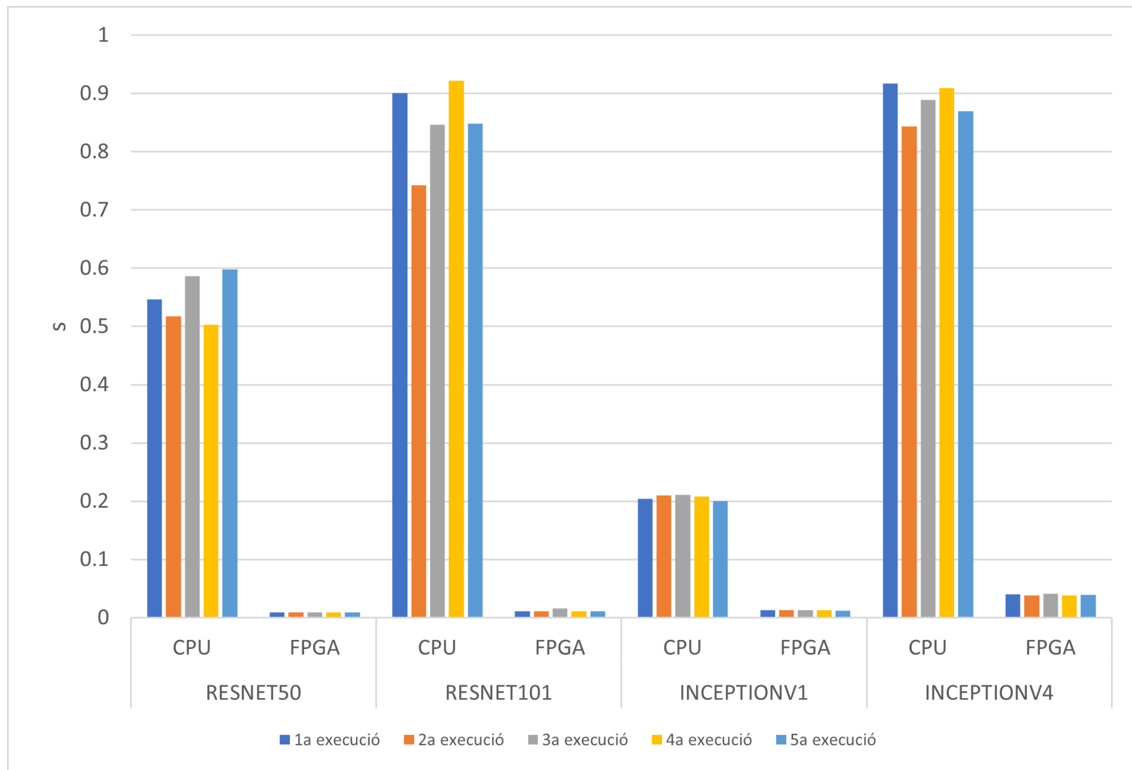

Figura 6.4: Latència de 4 models en la inferència d'una imatge durant 5 execucions, en FPGA i CPU.

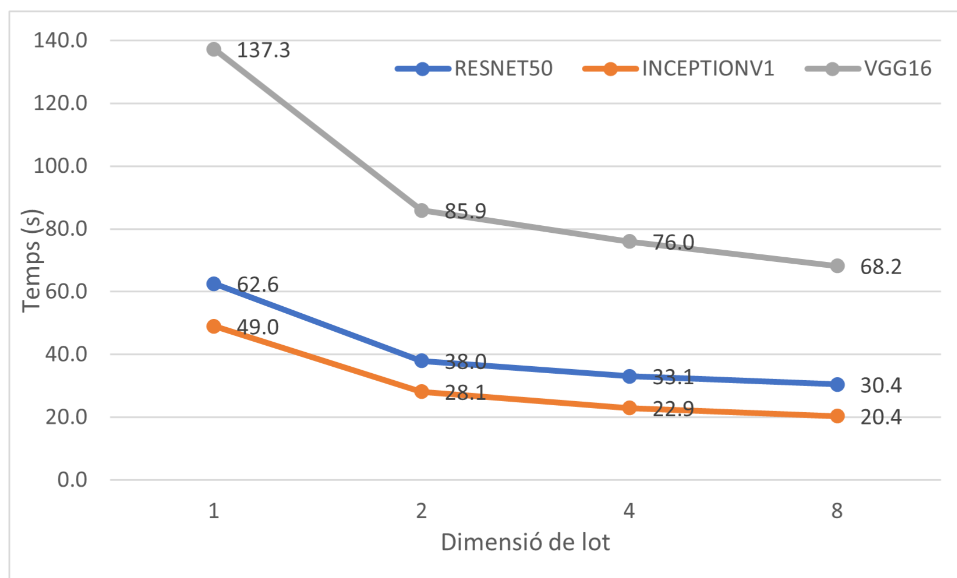

Figura 6.5: Rendiment de 3 models executats en la FPGA amb diferents dimensions de lot.

	RESNET50		RESNET101		VGG16	
	CPU_{FP32}	$FPGA_{INT8}$	CPU_{FP32}	$FPGA_{INT8}$	CPU_{FP32}	$FPGA_{INT8}$
Temps (s)	299.33537	30.4425	523.88671	37.2163	865.21828	68.231
	INCEPTIONV1		INCEPTIONV4		VGG19	
	CPU_{FP32}	$FPGA_{INT8}$	CPU_{FP32}	$FPGA_{INT8}$	CPU_{FP32}	$FPGA_{INT8}$
Temps (s)	142.16198	20.3557	964.51396	115.72320	1199.90377	73.4264

Taula 6.3: Temps d'execució dels models en la inferència de 5000 imatges, utlitzant la FPGA amb Vitis AI o la CPU amb TensorFlow.

L'últim experiment ha consistit en comparar el temps utilitzat en inferir 5000 imatges en CPU i en FPGA amb les sis xarxes neuronals, utilitzant una dimensió de lot de 8, ja que s'ha demostrat ser més òptim que altres configuracions. En la tabla 6.3 podem vore els temps utilitzats, on en la majoria de casos la millora és de l'ordre de 10 voltes més ràpid en la targeta.

6.2 Resultats amb TensorRT

L'experimentació amb la llibreria de TensorRT s'ha realitzat amb els mateixos models que en l'apartat anterior. Per a executar la inferència en GPU amb TensorRT, s'ha realitzat una conversió de models entrenats en TensorFlow a motor d'inferència de TensorRT. La llibreria de NVIDIA també permet la quantització dels pesos, reduint la precisió de FP32 a INT8, mitjançant un procés de calibració a l'hora de crear el motor d'inferència. Per als experiments s'ha utilitzat la versió INT8 dels models. A més, en alguns experiments s'ha utilitzat una versió FP32 de TensorRT. Els models utilitzats són dues versions de la xarxa `Resnet`, dues versions de la xarxa `GoogleNet` o `Inception` i dues versions de la xarxa `VGG`.

S'ha realitzat inferència amb els models en CPU amb precisió de 32 bits, utilitzant la ferramenta TensorFlow i en GPU, fent ús de TensorRT i amb precisió de 8 bits. Tots els resultats han sigut obtinguts amb una FPGA Alveo U200 i comparats amb els mateixos experiments en una CPU Intel Core i7-7700 HQ amb una freqüència de funcionament de 3,80 GHz. Els experiments realitzats són els següents:

- **Precisió FP32 vs. INT8**: En aquesta primera prova s'utilitzaran les 5000 imatges per calcular la precisió dels models executats en la CPU i en la GPU.
- **Latència CPU vs. GPU**: 5 imatges diferents són inferides amb els models en CPU i GPU, on mesurem el temps utilitzat per cada arquitectura. A més, s'observarà la diferència d'utilitzar la quantització dels pesos en GPU amb precisió INT8, i la versió amb precisió FP32 en GPU.
- **Capacitat de còmput en GPU**: Explorarem el gran paral·lelisme que ofereix la GPU en fer inferència amb les 5000 imatges amb diferents dimensions de lot. La GPU es caracteritza per la seua alta paral·lelització i ací podrem veure com afecta açò a la inferència.
- **Rendiment CPU vs. GPU**: Mesurarem el temps utilitzat per cada arquitectura en processar i inferir en 5000 imatges.

Tots els resultats han sigut obtinguts amb una GPU NVIDIA GeForce GTX 1050 i comparats amb els mateixos experiments en una CPU Intel Core i7-7700 HQ amb una

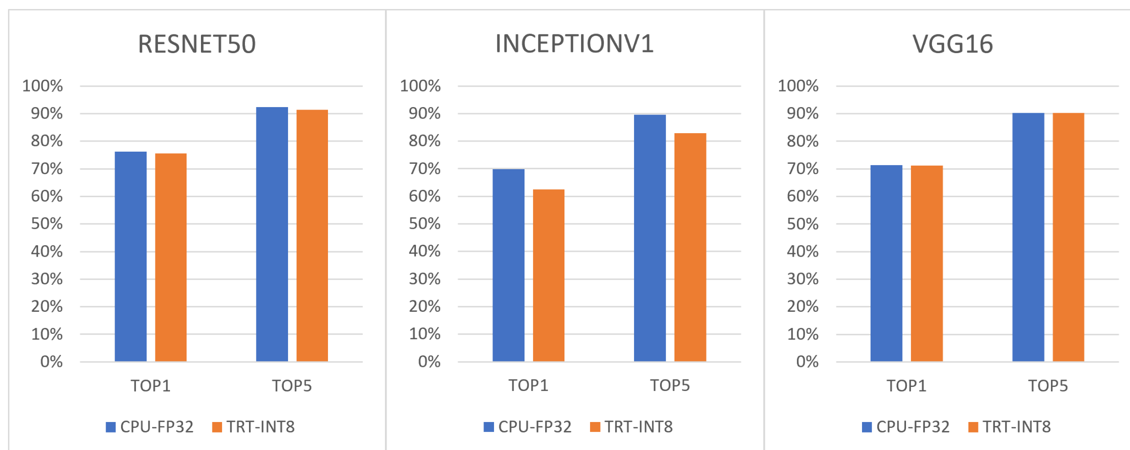

Figura 6.6: Diferència de top-1 i top-5 en precisió FP32 i INT8, en 3 dels models utilitzats en CPU i GPU.

freqüència de funcionament de 3,80 GHz. Els resultats en CPU són els mateixos que els utilitzats en l'apartat anterior. Per a inferir dades en els models, les imatges s'han de processar prèviament per a adaptar-les a cada xarxa. En l'annex està el codi de `input_fn` que conté les característiques per processar les imatges per a les xarxes.

El primer experiment ha consistit en comprovar la reducció de pressió de la xarxa neuronal en ser calibrada per a funcionar amb precisió reduïda INT8. A l'hora de construir el motor d'inferència de TensorRT s'ha utilitzat un calibrador amb l'ajuda de la llibreria, que fa un procés de propagació en la xarxa neuronal per a traure estadístiques que TensorRT utilitza per a escalar els pesos de FP32 a INT8, igual que en el cas de la FPGA. Açò pot comportar una pèrdua de precisió en la inferència. Com podem veure en la figura 6.6 i la taula 6.4, els models perden precisió en inferir amb precisió reduïda, però en els casos de les xarxes `Inception` la pèrdua és significativament major, fins a un 7% menys de precisió en la predicció. Com aquesta anomalia sols ha afectat aquesta topologia, podem suposar que és per les característiques d'aquestes xarxes o per un mal processament de les dades en l'etapa de preparació de les dades, ja que, com podem veure en els altres casos, la precisió es veu molt poc afectada per la inferència en precisió de 8 bits. També es dóna el cas en el model `VGG19` que la versió quantitzada ofereix major precisió que la sense quantitzar. Aquesta anomalia es deu al fet que la transformació dels pesos influeix en les prediccions i s'han realitzat algunes prediccions distintes, que han resultat ser les correctes.

La següent experimentació ha consistit a mesurar la latència i calcular el *speedup* dels motors d'inferència en GPU, respecte als tests en CPU. Per a aquest experiment, s'han desenvolupat dues versions, una versió amb la mateixa precisió en els pesos que CPU (FP32) i una amb precisió reduïda de 8 bits, que hem utilitzat en la prova anterior i continuarem utilitzant en les següents. Com podem observar en la 6.5, ambdues versions ofereixen una menor latència que la versió de CPU, però els models amb precisió INT8 són molt més ràpids que les versions FP32. L'acceleració és molt similar en les dues versions de cada model: 60 voltes més ràpid en les `Resnet`, 40 voltes més ràpid en les `Inception` i més de 200 vegades més ràpid en les `VGG`. El model que més ràpid s'executa és `InceptionV1`, ja que els mòduls *inception* són capes que poden executar-se en paral·lel i la GPU s'especialitza en la paral·lelització.

La següent prova ha consistit a mesurar el temps que necessita cada model per inferir 5000 imatges amb diferents dimensions de lot. La mida del lot és el nombre de mostres que es computen en paral·lel, on les prediccions s'obtenen en una sola propagació de la xarxa. En la figura 6.7 podem veure el temps requerit amb dimensions de lot 1, 2, 4 i 8.

	RESNET50		RESNET101		VGG16	
	CPU_{FP32}	GPU_{INT8}	CPU_{FP32}	GPU_{INT8}	CPU_{FP32}	GPU_{INT8}
TOP1	76.24%	75.56%	76.74%	77.00%	71.32%	71.26%
TOP5	92.38%	91.42%	92.66%	92.60%	90.26%	90.26%
LOSS1	0.68%		-0.26%		0.06%	
LOSS5	0.96%		0.06%		0.00%	
	INCEPTIONV1		INCEPTIONV4		VGG19	
	CPU_{FP32}	GPU_{INT8}	CPU_{FP32}	GPU_{INT8}	CPU_{FP32}	GPU_{INT8}
TOP1	69.76%	62.44%	79.84%	73.22%	70.90%	71.14%
TOP5	89.63%	82.88%	95.36%	91.10%	90.04%	90.18%
LOSS1	7.32%		6.62%		-0.24%	
LOSS5	6.75%		4.26%		-0.14%	

Taula 6.4: Diferència de la precisió top-1 i top-5 en inferència de 5000 imatges amb una CPU utilitzant TensorFlow amb precisió FP32 i una GPU utilitzant TensorRT amb precisió reduïda INT8, en 6 models de xarxes neuronals diferents.

	RESNET50			RESNET101		
	CPU_{FP32}	GPU_{FP32}	GPU_{INT8}	CPU_{FP32}	GPU_{FP32}	GPU_{INT8}
Temps (s)	0.55002	0.19555	0.00913	0.85201	0.21542	0.01288
S_{FP32}	2.81263			3.95501		
S_{INT8}	60.22976			66.15248		
	INCEPTIONV1			INCEPTIONV4		
	CPU_{FP32}	GPU_{FP32}	GPU_{INT8}	CPU_{FP32}	GPU_{FP32}	GPU_{INT8}
Temps (s)	0.20664	0.00962	0.00563	0.88558	0.04713	0.01921
S_{FP32}	21.49010			18.79092		
S_{INT8}	36.72718			46.09883		
	VGG16			VGG19		
	CPU_{FP32}	GPU_{FP32}	GPU_{INT8}	CPU_{FP32}	GPU_{FP32}	GPU_{INT8}
Temps (s)	2.94335	0.03682	0.01294	3.34081	0.04342	0.01465
S_{FP32}	79.94737			76.93980		
S_{INT8}	227.40073			228.05005		

Taula 6.5: Latència i *speedup* dels models en la inferència d'una imatge, utlitzant la GPU amb TensorRT o la CPU amb TensorFlow. La primera fila de *speedup* representa CPU vs GPU amb TensorRT utilitzant precisió de 32 bits i la segona fila de *speedup* representa CPU vs GPU amb TensorRT utilitzant precisió de 8 bits.

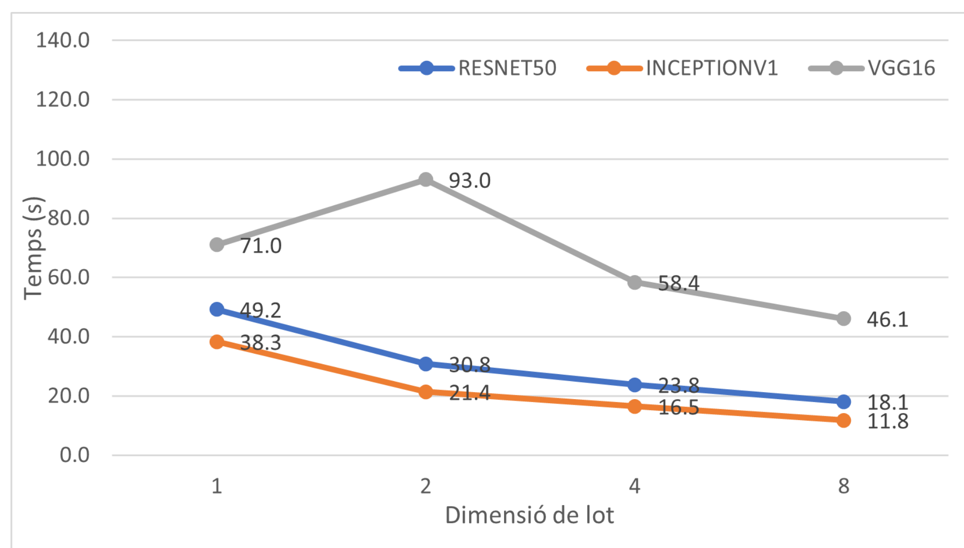

Figura 6.7: Rendiment de 3 models executats en la GPU amb diferents dimensions de lot.

	RESNET50		RESNET101		VGG16	
	CPU_{FP32}	GPU_{INT8}	CPU_{FP32}	GPU_{INT8}	CPU_{FP32}	GPU_{INT8}
Temps (s)	299.335	18.1255	523.887	26.6955	865.218	46.1299
	INCEPTIONV1		INCEPTIONV4		VGG19	
	CPU_{FP32}	GPU_{INT8}	CPU_{FP32}	GPU_{INT8}	CPU_{FP32}	GPU_{INT8}
Temps (s)	142.162	11.7982	964.514	39.3963	1199.904	52.589

Taula 6.6: Temps d'execució de la inferència de 5000 imatges en els models, utlitzant la CPU amb TensorFlow, o la GPU, amb TensorRT i precisió de 8 bits.

Com podem observar, a més grandària de lot, més ràpid es processen les imatges. La capacitat de còmput paral·lel de la GPU influeix molt en els temps obtinguts. Es dóna una anomalia en la xarxa `VGG16`, on amb una mida de 2 de lot, es tarda més temps que a l'inferir una imatge per propagació. No hem trobat una explicació a aquest error. Cal afegir que la GPU utilitzada no tenia suficient memòria per a experimentar amb més grandària de lot.

I per últim, s'han mesurat els temps d'execució en inferir 5000 imatges, amb dimensió de lot 8 per a tots els models. La tabla 6.6 conté els resultats. Com podem observar, la millora és molt significativa en la inferència amb TensorRT, utilitzant de 10 a 20 vegades menys temps en la majoria dels models per a processar el mateix nombre d'imatges que en CPU. Com hem pogut veure en l'estudi anterior, el paral·lelisme de la GPU dota a aquesta d'una gran velocitat en processos com aquest.

6.3 Impacte de les funcions d'activació

Per a l'avaluació de les funcions d'activació implementades, s'ha analitzat el seu impacte en dues topologies de xarxes neuronals artificials. S'ha avaluat l'evolució de l'entrenament de les 6 noves funcions d'activació i s'han utilitzat diferents conjunts de dades. Per a aquest estudi utilitzem la ferramenta de modelatge de xarxes neuronals HELENNA, anteriorment descrita. Les funcions a LeakyReLU i ELU s'han inicialitzat amb els valors 0.1 i 0.5 respectivament, ja que aquests valors aporten rellevància als valors negatius de l'eixida de les neurones, sense excedir-nos i crear inconvenients a l'entrenament. S'han

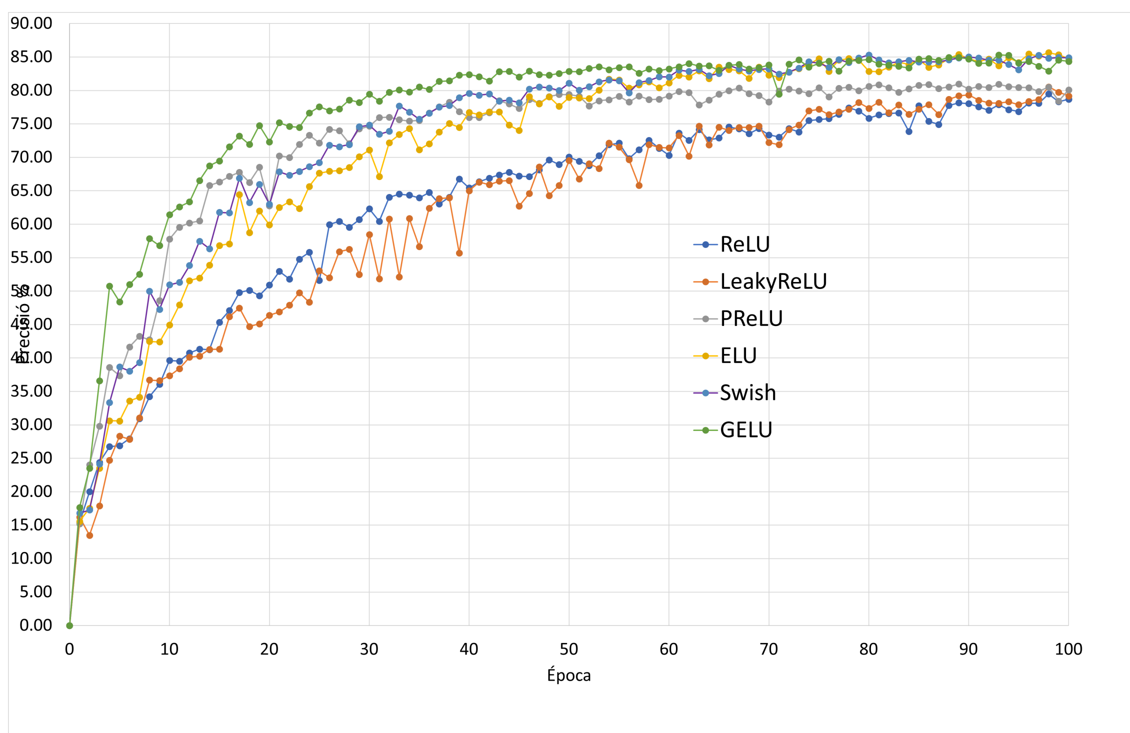

Figura 6.8: Evolució de la precisió en l'entrenament del model `VGG16` amb el dataset CIFAR 10 en les 6 funcions d'activació estudiades.

ReLU	LeakyReLU	PReLU
78.7	79.17	80.09
ELU	Swish	GELU
84.75	84.90	84.36

Taula 6.7: Precisió final en la inferència de les 6 funcions d'activació en el model `VGG16`

provat les funcions etiquetades en el conjunt de dades CIFAR10 i el conjunt de dades ImageNet.

Primerament, s'ha entrenat la xarxa `VGG16` amb el conjunt de dades CIFAR10 [19], durant 100 èpoques i amb les diferents funcions d'activació. En la figura 6.8 podem veure l'evolució de la precisió en l'entrenament. S'observa com les funcions que més ràpid aprenen són la GELU, la Swish i la PReLU. La GELU i la Swish comparteixen característiques, ja que tenen una complexitat matemàtica elevada, mentre que la PReLU té l'aprenentatge del valor alfa, que regula els valors negatius. Podem observar com les funcions ReLU i Leaky ReLU tenen un comportament molt similar, sent les funcions que pitjors resultats de precisió han donat al llarg de l'entrenament. En la taula 6.7 podem veure la precisió final que tenen aquestes xarxes en el conjunt de dades de validació, on es fa inferència amb el model per veure la precisió que té en cada època. Vegem com les funcions ELU, Swish i GELU són superiors a les altres 3, ja que necessiten menys èpoques per aprendre i, a més, aconsegueixen més precisió en finalitzar el seu entrenament.

També s'ha realitzat experimentació amb el conjunt de dades ImageNet, amb les funcions ReLU, Leaky ReLU i ELU. S'ha entrenat la xarxa AlexNet durant 10 èpoques. Podem observar el mateix comportament que en el cas anterior, la funció ELU té una corba d'aprenentatge més ràpida que les altres funcions, però en aquest cas la diferència final no és tan gran com en l'experimentació anterior. L'entrenament amb la base de dades

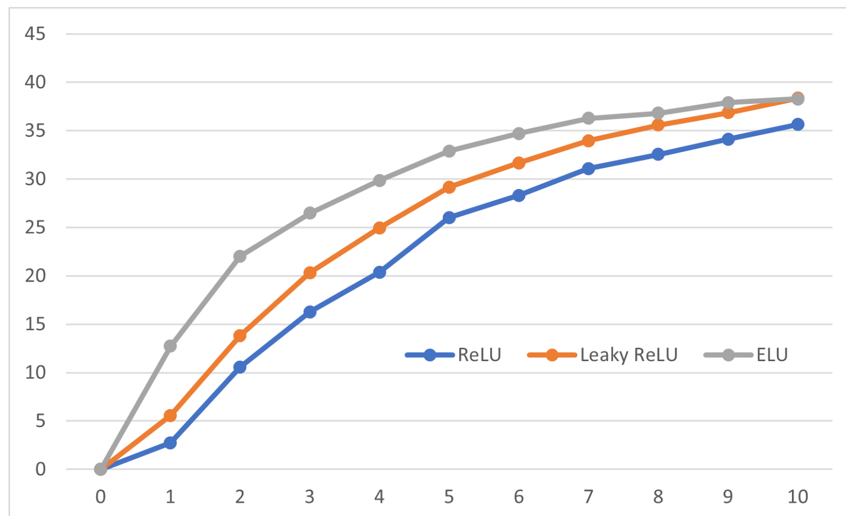

Figura 6.9: Evolució de la precisió en l'entrenament del model `AlexNet` amb el dataset ImageNet en 3 de les funcions d'activació estudiades.

	RESNET50		RESNET101		VGG16	
	$FPGA_{INT8}$	GPU_{INT8}	$FPGA_{INT8}$	GPU_{INT8}	$FPGA_{INT8}$	GPU_{INT8}
TOP1	75.44%	75.56%	75.32%	77.00%	67.54%	71.26%
TOP5	91.98%	91.42%	92.12%	92.60%	89.96%	90.26%
	INCEPTIONV1		INCEPTIONV4		VGG19	
	$FPGA_{INT8}$	GPU_{INT8}	$FPGA_{INT8}$	GPU_{INT8}	$FPGA_{INT8}$	GPU_{INT8}
TOP1	68.76%	62.44%	79.28%	73.22%	70.54%	71.14%
TOP5	88.92%	82.88%	94.69%	91.10%	89.76%	90.18%

Taula 6.8: Precisió top-1 i top-5 dels sis models estudiats en la inferència de 5000 imatges, utilitzant Vitis AI i TensorRT.

ImageNet és un procés molt costós que pot durar dies, així que hem ajustat el nombre d'èpoques. Les funcions PReLU, Swish i GELU ocasionaven errors de segmentació que no hem sigut capaços d'identificar els motius, possiblement per la complexitat matemàtica d'aquestes 3 funcions.

6.4 FPGA vs. GPU

A continuació es compararan els resultats obtinguts amb les diferents ferramentes i el diferent maquinari utilitzat en els mateixos models de xarxes neuronals. Els programaris Vitis AI i TensorRT tenen moltes característiques en comú, ja que ambdós fan ús del procés de quantització per reduir la precisió dels paràmetres d'una xarxa neuronal i poder aconseguir una inferència més ràpida, més eficient i amb menys consum de memòria. A continuació es veuran les diferències entre la inferència en GPU i en FPGA.

Primer podem veure la diferència de precisió, en la taula 6.8 vegem com els models executats en GPU i FPGA són molt similars en aquest aspecte, excepte les xarxes `Inception` on ja hem comentat el problema en l'apartat anterior. També destaca que els models `VGG` presenten més precisió en GPU que en FPGA, on podem suposar que els paràmetres han sigut calibrats més precisos en TensorRT.

A continuació veurem la diferència de latència tant en GPU amb TensorRT com en FPGA amb Vitis AI. En la figura 6.10 podem observar com en les xarxes neuronals amb

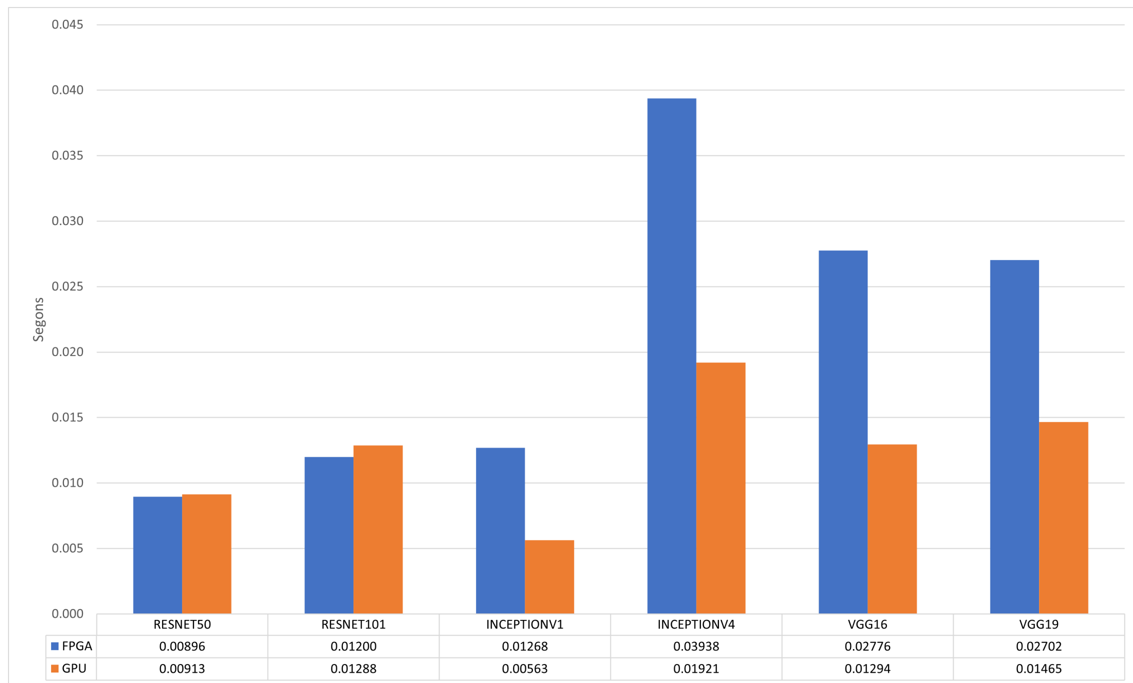

Figura 6.10: Diferència de latència en la inferència dels 6 models amb Vitis AI i TensorRT.

més paràmetres, la GPU ofereix millors resultats, però, en les xarxes `Resnet`, la FPGA és lleugerament millor que la unitat de processament gràfic. La gran diferència en les `Inception` podem deduir que és per la gran paral·lelització de la GPU, on les xarxes neuronals amb aquesta arquitectura espremen aquesta característica amb l'execució de capes en paral·lel. La gran densitat d'operacions de les `VGG` ofereix una latència menor.

I per últim observarem la diferència de rendiment entre la FPGA i la GPU. S'ha utilitzat una dimensió de lot de 8 imatges, per tant, en cada propagació tant en FPGA com en GPU es processen 8 imatges simultàniament. En la figura 6.11 s'aprecia com en tots els casos la GPU és capaç de processar i inferir les 5000 imatges més ràpid que la FPGA. Una volta més, la gran paral·lelització dels nuclis de la GPU permet oferir aquestes prestacions enfront d'altres maquinaris com la FPGA o la CPU.

Cal afegir que, en la majoria dels casos, la FPGA té un consum d'energia menor que les GPU. En aquest cas, hem pogut estimar el consum energètic de la GPU NVIDIA GeForce GTX 1050 des de 70 W fins a 160 W en operacions costoses, com és el procés d'inferència i, la FPGA Alveo U200 entre 100 W i 110 W. A l'hora de desenvolupar una solució en alguna d'aquestes arquitectures és una característica important a tenir en compte.Per exemple, en aplicacions de conducció autònoma on el consum energètic és un factor molt important i on no és tan important el paral·lelisme a nivell de lot, ja que les imatges arriben en temps real, les FPGA es presenten com una solució molt competitiva.

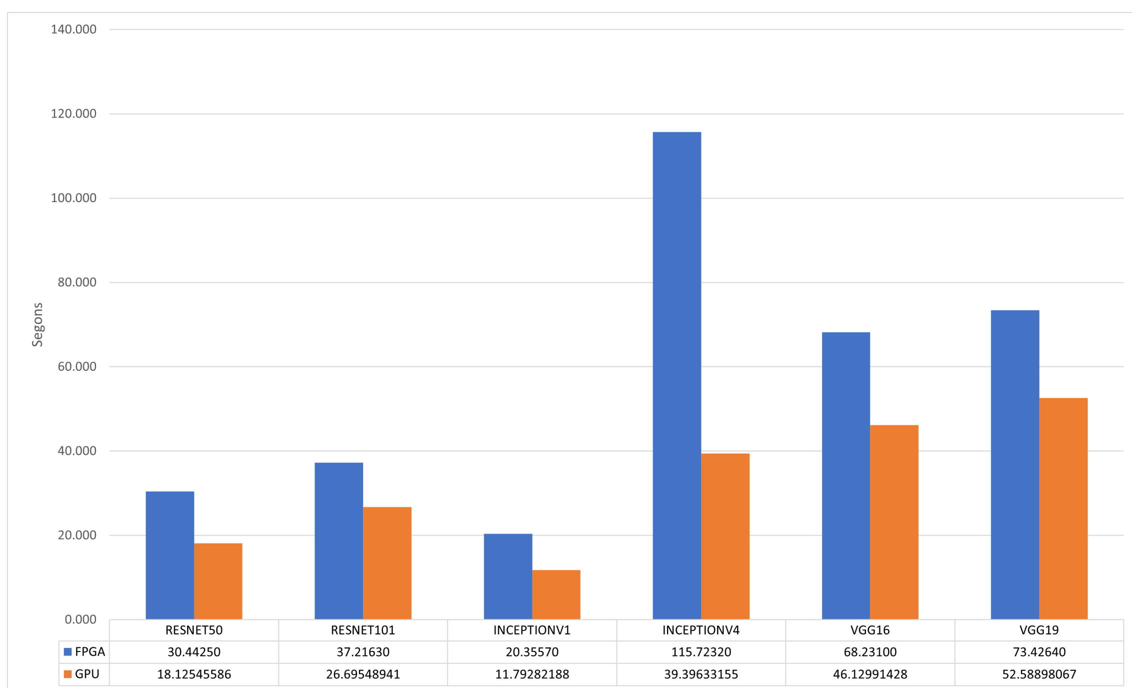

Figura 6.11: Diferència de rendiment en la inferència de 5000 imatges en els 6 models, amb Vitis AI i TensorRT.

CAPÍTOL 7
Conclusions

Aquest projecte ha consistit a explorar opcions del suport d'inferència de xarxes neuronals en diferents plataformes heterogènies, i l'ús de diferents funcions en xarxes neuronals per millorar la seua precisió final. Després del treball realitzat amb les diferents ferramentes, podem arribar a les següents conclusions:

- S'han adquirit coneixements en profunditat de les característiques principals de les xarxes neuronals, així com de l'estat actual de l'aprenentatge profund.
- S'ha fet un estudi de les arquitectures GPU i FPGA, presents en les plataformes heterogènies i utilitzades en aquest treball per a l'experimentació.
- S'han utilitzat amb èxit les ferramentes més avançades per a l'execució de xarxes neuronals en GPU i FPGA. S'ha detallat el flux de treball a seguir per desplegar un model entrenat en aquests dispositius, amb l'estudi de les ferramentes i el desenvolupament de codi. A més, s'han comparat els processos d'inferència d'ambdues ferramentes fent èmfasi en el que destaca cada un. Els resultats confirmen que l'optimització del programari de les xarxes neuronals té un impacte significatiu en el temps d'execució del procés d'inferència, tant a FPGA com a GPU.
- S'han implementat noves funcionalitats en una plataforma d'entrenament i inferència de xarxes neuronals, amb l'objectiu de millorar característiques dels processos d'inferència i entrenament. En particular s'han desenvolupat 5 noves funcions d'activació derivades de la ReLU. Els resultats obtingut corroboren la millora de la precisió en el proces d'inferència amb funcions d'activació més avançades.

7.1 Relació amb els estudis cursats

Gran part dels coneixements requerits ja havien estat adquirits en assignatures dels cursos anteriors i de l'actual.

Estructura de Computadors: els coneixements de les característiques del processador juntament amb les arquitectures més utilitzades en el món de la informàtica han sigut essencials per a aquest treball.

Arquitectura i Enginyeria de Computadors: en aquesta assignatura es detalla el concepte d'arquitectura i els paràmetres que afecten les prestacions d'aquesta, temes molt presents en el treball.

Arquitectures Avançades: aquesta assignatura aprofundeix més en les arquitectures del processador i l'impacte de les estructures del processador en el temps d'execució de les aplicacions, sent de gran utilitat en el desenvolupament del projecte.

Llenguatges i Entorns de Programació Paral·lela: l'ús de ferramentes de programació paral·lela i arquitectures amb gran paral·lelització han estat molt presents en el treball realitzat. Els coneixements adquirits en aquesta assignatura han sigut molt útils (anàlisi de dades i programació en C).

Disseny de Sistemes Digitals: en aquesta assignatura s'aprofundeix en l'arquitectura de les FPGA. En el projecte s'ha treballat amb aquesta arquitectura, així que gran part dels coneixements necessaris per a entendre el funcionament ja havien estat tractats en aquesta assignatura.

Sistemes Intel·ligents: en aquesta assignatura s'introdueixen conceptes tractats en aquest treball, com les xarxes neuronals i la intel·ligència artificial.

7.2 Treball futur

A continuació s'exposen uns estudis i implementacions que es poden realitzar en el futur d'aquest TFG.

7.2.1. Quantificar l'ús de memòria en les ferramentes

Un estudi de l'ús de memòria haguera enriquit el treball, ja que en l'experimentació realitzada en Vitis AI i TensorRT sobre FPGA i GPU no s'ha pogut quantificar aquesta característica. Tampoc s'ha pogut veure l'impacte energètic real que té la inferència de xarxes neuronals en aquestes ferramentes. En un futur cal explorar alternatives i mètodes per poder mesurar aquestes dues característiques.

7.2.2. Procés de quantització en HELENNA

En aquest treball s'han utilitzat ferramentes que automatitzen el procés de quantització de les xarxes neuronals, sense que l'usuari siga conscient de com està funcionant aquesta reducció de precisió en el model quantitzat. La gran complexitat d'aquesta junt amb els coneixements limitats en aquest camp al principi de la realització del treball han impedit l'exploració d'aquesta possibilitat. Amb la finalització del treball i els coneixements adquirits, quantitzar els pesos d'una xarxa neuronal dissenyada amb la ferramenta HELENNA és possible en un futur.

A més, açò permetria veure l'impacte de les funcions d'activació desenvolupades al costat d'aquesta tècnica, ja que en l'experimentació amb les ferramentes Vitis AI i TensorRT, no totes aquestes funcions estaven suportades, per tant es va descartar fer un estudi conjunt.

Bibliografia

[1] Especificacions de la targeta Alveo U200 i Alveo U250, versió 1.3.1, emeses el 5 de maig de 2020. Consultades a https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf (últtima visita 25/05/2021)

[2] Hideharu Amano *Principles and Structures of FPGAs.* Springer Singapore, Yokohama, 2018.

[3] Suren Chilingaryan, Evelina Ametova, Anreas Kopmann, Alessandro Mirone. Reviewing GPU architectures to build efficient back projection for parallel geometries *Journal of Real-Time Image Processing (2020) 17:1331–1373*, juny, 2019.

[4] Djork-Arne Clevert, Thomas Unterthiner, Sepp Hochreiter Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs) *arXiv preprint arXiv:1511.07289v5 [cs.LG]*, febrer, 2016.

[5] Deep Learning. Introducción práctica con Keras. Disponible a https://torres.ai/deep-learning-inteligencia-artificial-keras/

[6] David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams Learning representations by back-propagating errors. *Nature*, 323:533–536, octubre, 1986.

[7] Xavier Glorot, Antoine Bordes, Yoshua Bengio Deep Sparse Recifier Neural Networks *Proceedings of the Fourteenth Internation Conference on Artificial Intelligence and Statistics, PMLR 15:315-323*, 2011.

[8] Pàgina web del Grup d'Arquitectures Paral·leles de la UPV. http://www.gap.upv.es/ (última visita 31/05/2021)

[9] Kaiming He, Xiangyu Zhang, Shaoqing, Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *arXiv preprint arXiv:1502.01852v1 [cs.CV]*, febrer, 2015.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun Deep Residual Learning for Image Recognition. *arXiv preprint arXiv:1512.03385v1 [cs.CV]*, desembre, 2015.

[11] Repositori de la ferramenta HELENNA en Github. Disponible en https://github.com/PEAK-UPV/HELENNA/

[12] Dan Hendrycks, Kevin Gimpel. Gaussian Error Linear Units (GELUs) *arXiv preprint arXiv:1606.08415v4 [cs.LG]*, juny, 2016.

[13] Breu història de l'aprenentatge profund de 1943-2019 Consultada a https://machinelearningknowledge.ai/brief-history-of-deep-learning/

[14] Sergey Ioffe, Christian Szegedy Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift *arXiv:1502.03167v3 [cs.LG]*2, març, 2015.

[15] Raghuraman Krishnamoorthi Quantizing deep convolutional networks for efficient inference: A whitepaper *arXiv:1806.08342v1 [cs.LG]*, juny, 2018.

[16] Andrew L. Maas, Awni Y. Hannun, Andrew Y. Ng Rectifier Nonlinearities Improve Neural Network Acoustic Models *International Conference on Machine Learning, volume 30*, 2013.

[17] "The Machine Learning Dictionary" Disponible a www.cse.unsw.edu.au/~billw/mldict.html

[18] Referències comparatives d'entrenament i inferència de MLCommons. Consultades a https://mlcommons.org/en/training-normal-07/ (última visita 19/05/2021)

[19] Vinod Nair, Alex Krizhevsky, Geoffrey Hinton *The CIFAR-10 dataset* Disponible a https://www.cs.toronto.edu/~kriz/cifar.html (última visita 25/05/2021)

[20] Rajat Raina, Anand Madhavan, Andrey Y. Ng Large-scale Deep Unsupervised Learning using Graphics Processors *Disponible online http://robotics.stanford.edu/~ang/papers/icml09-LargeScaleUnsupervisedDeepLearningGPU.pdf*

[21] Prajit Ramachandran, Barret Zoph, Quoc V. Le. Swish: A Self-Gated Activation Function *arXiv preprint arXiv:1710.05941v1 [cs.NE]*, octubre, 2017.

[22] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain *Psychological Review* 65:6:386–408, novembre, 1958.

[23] Stephanie Rupprich The Xilinx UltraScale Architecture *[Online]*, octubre, 2015. https://ra.ziti.uni-heidelberg.de/cag/images/seminars/ss14/2014-rupprich-report.pdf

[24] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, Li Fei-Fei ImageNet Large Scale Visual Recognition Challenge *arXiv preprint arXiv:1409.0575v3 [cs.CV]*, gener, 2015.

[25] Warren S. McCulloch, Walter Pitts A logical calculus of the ideas immanent in nervous activity *The bulletin of mathematical biophysics* 5:115–133, desembre, 1943.

[26] Karen Simonyan, Andrew Zisserman Very Deep Convolutional Networks for Large-Scale Image Recognition *arXiv preprint arXiv:1409.1556 [cs.CV]*, abril, 2015.

[27] Nitish Strivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov Dropout: A Simple Way to Prevent Neural Networks from Overfitting *Journal of Machine Learning Research 15 (2014) 1929–1958*, juny, 2014.

[28] Christian Szegedy, Wei Liu, Yangqing jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich Going deeper with convolutions *arXiv:1409.4842v1 [cs.CV]* , setembre, 2014.

[29] Guia d'usuari del kit de desenvolupament de programari TensorRT de NVIDIA, versió 8.0.0 d'Abril de 2021. Consultada a https://docs.nvidia.com/deeplearning/tensorrt/pdf/TensorRT-Developer-Guide.pdf

[30] Alan Turing. Computing Machinery and Intelligence *Mind*, 59:236:433–460, octubre, 1950.

[31] Improving the speed of neural networks on CPUs Vincent Vanhoucke, Andrew Senior, Mark Z. Mao *Disponible online* *http://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/37631.pdf*, 2011.

[32] Guia d'usuari de la ferramenta Vitis AI de Xilinx, versió UG1414 emesa el 3 de febrer de 2021. Consultada a https://www.xilinx.com/support/documentation/sw_manuals/vitis_ai/1_3/ug1414-vitis-ai.pdf (última visita 22/05/2021)

[33] Nicholas Wilt *The CUDA Handbook: A Comprehensive Guide to GPU Programming* Addison-Wesley Professional, Crawfordsville, 2013.

APÈNDIX A
Funcions i codi

A continuació estan algunes de les funcions i el codi implementat en el desenvolupament d'aquest treball.

A.1 Vitis AI i TensorRT

A.1.1. Preprocessar imatges: input_fn.py

```
import cv2

_R_MEAN = 123.68
_G_MEAN = 116.78
_B_MEAN = 103.94

MEANS = [_B_MEAN,_G_MEAN,_R_MEAN]

def resize_shortest_edge(image, size):
  H, W = image.shape[:2]
  if H >= W:
    nW = size
    nH = int(float(H)/W * size)
  else:
    nH = size
    nW = int(float(W)/H * size)
  return cv2.resize(image,(nW,nH))

def mean_image_subtraction(image, means):
  B, G, R = cv2.split(image)
  B = B - means[0]
  G = G - means[1]
  R = R - means[2]
  return cv2.merge([R, G, B])

def BGR2RGB(image):
  B, G, R = cv2.split(image)
  return cv2.merge([R, G, B])

def central_crop(image, crop_height, crop_width):
  offset_height = (image.shape[0] - crop_height) // 2
  offset_width = (image.shape[1] - crop_width) // 2
  return image[offset_height:offset_height + crop_height, offset_width:
               offset_width + crop_width, :]

def normalize(image):
  image=image/256.0
```

```
38   image=image-0.5
39   return image*2
40
41 # preprocessa resnets i vgg
42 def preprocess_imagenet(image, channels=3, height=224, width=224):
43     img = resize_shortest_edge(image, 256)
44     img = mean_image_subtraction(img, MEANS)
45     return central_crop(img, 224, 224)
46
47 def preprocess_inception(image, channels=3, height=224, width=224):
48     img = BGR2RGB(image)
49     img = resize_shortest_edge(img, 256)
50     img = central_crop(img,224,224)
51     return normalize(img)
52
53 def preprocess_inceptionv4(image, channels=3, height=299, width=299):
54     img = BGR2RGB(image)
55     img = resize_shortest_edge(img, 342)
56     img = central_crop(img, 299, 299)
57     return normalize(img)
58
59 calib_image_dir = "/workspace/images/1000images/"
60 calib_image_list = "/workspace/images/tf_calib.txt"
61
62 def calib_input(iter):
63     images = []
64     line = open(calib_image_list).readlines()
65     curline = line[iter]
66     name = curline.strip()
67     image = cv2.imread(calib_image_dir + name)
68     images.append(preprocess_imagenet(image))
69     return {"input": images}
```

A.1.2. Inferència amb Vitis AI, codi de Xilinx modificat: `inferenceFPGA.py`

```
1  # Copyright 2019 Xilinx Inc.
2  #
3  # Licensed under the Apache License, Version 2.0 (the "License");
4  # you may not use this file except in compliance with the License.
5  # You may obtain a copy of the License at
6  #
7  #     http://www.apache.org/licenses/LICENSE-2.0
8  #
9  # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14
15 from ctypes import *
16 import cv2
17 import numpy as np
18 import runner
19 import os
20 import input_fn
21 import math
22 import threading
23 import time
24 import sys
25
26 # Calcular funcio Softmax en CPU
27 def CPUCalcSoftmax(data,size):
```

```
    sum=0.0
    result = [0 for i in range(size)]
    for i in range(size):
        result[i] = math.exp(data[i])
        sum +=result[i]
    for i in range(size):
        result[i] /=sum
    return result

def get_script_directory():
    path = os.getcwd()
    return path

# Obte els resultats TOP5
def TopK(datain, size, filePath):
    cnt  = [i for i in range(size) ]
    pair = zip(datain, cnt)
    pair = sorted(pair, reverse=True)
    softmax_new, cnt_new = zip(*pair)
    fp = open(filePath, "r")
    data1 = fp.readlines()
    fp.close()
    print ("")
    #print(name)
    for i in range(5):
        flag = 0
        for line in data1:
            if flag == cnt_new[i]:
                print("Top[%d] %f %s" % (i, (softmax_new[i]), (line.strip)("\n"
                    )))
            flag = flag+1

l = threading.Lock()
SCRIPT_DIR = get_script_directory()
calib_image_dir  = "/workspace/images/" + sys.argv[2]
label_file = "/workspace/images/words.txt"
IMAGE_WIDTH = 224
IMAGE_HEIGHT = 224
batchSize = 1
global threadnum
threadnum = 0
global runTotall
runRotal = 0

# Executa inferencia
def runInference(dpu, img, cnt):

    # dimensions dels tensors
    inputTensors  = dpu.get_input_tensors()
    outputTensors = dpu.get_output_tensors()
    tensorformat  = dpu.get_tensor_format()
    if tensorformat == dpu.TensorFormat.NCHW:
        outputHeight  = outputTensors[0].dims[2]
        outputWidth   = outputTensors[0].dims[3]
        outputChannel = outputTensors[0].dims[1]
    elif tensorformat == dpu.TensorFormat.NHWC:
        outputHeight  = outputTensors[0].dims[1]
        outputWidth   = outputTensors[0].dims[2]
        outputChannel = outputTensors[0].dims[3]
    else:
        exit("Format error")
    outputSize = outputHeight * outputWidth * outputChannel
    softmax = np.empty(outputSize)

```

```
91      global runTotall
92      count = cnt
93      while count < runTotall:
94          l.acquire()
95          if (runTotall < (count+batchSize)):
96              runSize = runTotall - count
97          else:
98              runSize = batchSize
99          l.release()
100         shapeIn = (runSize,) + tuple([inputTensors[0].dims[i] for i in range(
                inputTensors[0].ndims)][1:])
101
102         outputData = []
103         inputData  = []
104         outputData.append(np.empty((runSize,outputSize), dtype = np.float32,
                order = 'C'))
105         inputData.append(np.empty((shapeIn), dtype = np.float32, order = 'C'))
106
107         # copia image o images en buffer dentrada
108         for j in range(runSize):
109             imageRun = inputData[0]
110             imageRun[j,...] = img[count+j]
111
112         # executa inferencia en FPGA
113         job_id = dpu.execute_async(inputData,outputData)
114         dpu.wait(job_id)
115
116         # calcula softmax i top5 a partir del tensor deixida
117         for j in range(runSize):
118             softmax = CPUCalcSoftmax(outputData[0][j], outputSize)
119             TopK(softmax, outputSize, label_file)
120         l.acquire()
121         count = count + threadnum*runSize
122         l.release()
123
124 def main(argv):
125     global threadnum
126     # crea un runner
127     dpu = runner.Runner(argv[1])
128     listimage = os.listdir(calib_image_dir)
129     threadAll = []
130     threadnum = 1
131     i = 0
132     global runTotall
133     runTotall = len(listimage)
134
135     """image list to be run """
136     img = []
137
138     for i in range(runTotall):
139         path = os.path.join(calib_image_dir, listimage[i])
140         img.append(input_fn.preprocess_fn(path))
141
142     imgData = np.transpose(img, (0, 3, 1, 2))
143
144     # executa inferencia
145     time1 = time.time()
146     runInference(dpu, imgData, 0)
147     time2 = time.time()
148
149     timetotal = time2 - time1
150
151     fps = float(1 / timetotal)
152     print("%.4f seconds" %timetotal)
```

```
153     print("%.2f FPS" %fps)
154
155     del dpu
156
157 if __name__ == "__main__":
158     if len(sys.argv) != 3:
159         print("please input image folder and json file path.")
160     else :
161         main(sys.argv)
```

A.1.3. Crear motor d'inferència de TensorRT: `buildEngine.py`

```
1  import os
2  import glob
3  import argparse
4  import random
5  import numpy as np
6  import cv2
7
8  from input_fn import preprocess_imagenet
9  from onnx import ModelProto
10 import tensorrt as trt
11 import pycuda.driver as cuda
12 import pycuda.autoinit
13
14 TRT_LOGGER = trt.Logger(trt.Logger.WARNING)
15 trt_runtime = trt.Runtime(TRT_LOGGER)
16
17 # https://docs.nvidia.com/deeplearning/sdk/tensorrt-api/python_api/infer/Int8/
       EntropyCalibrator2.html
18 class ImagenetCalibrator(trt.IInt8EntropyCalibrator2):
19
20     def __init__(self, calibration_files=[], batch_size=32, input_shape=(224,
           224, 3),
21                  cache_file="calibration.cache", preprocess_func=None):
22         super().__init__()
23         self.input_shape = input_shape
24         self.cache_file = cache_file
25         self.batch_size = batch_size
26         self.batch = np.zeros((self.batch_size, *self.input_shape), dtype=np.
               float32)
27         self.device_input = cuda.mem_alloc(self.batch.nbytes)
28
29         self.files = calibration_files
30         if len(self.files) % self.batch_size != 0:
31             self.files += calibration_files[(len(calibration_files) % self.
                   batch_size):self.batch_size]
32
33         self.batches = self.load_batches()
34
35         self.preprocess_func = preprocess_func
36
37     def load_batches(self):
38         for index in range(0, len(self.files), self.batch_size):
39             for offset in range(self.batch_size):
40                 image = cv2.imread(self.files[index + offset])
41                 self.batch[offset] = self.preprocess_func(image, *self.
                       input_shape)
42             yield self.batch
43
44     def get_batch_size(self):
45         return self.batch_size
```

```
46
47      def get_batch(self, names):
48          try:
49              batch = next(self.batches)
50              cuda.memcpy_htod(self.device_input, batch)
51              return [int(self.device_input)]
52          except StopIteration:
53              return None
54
55      def read_calibration_cache(self):
56          # Si hi ha un arxiu cache, utilitzar  per no fer la calibracio de nou
57          if os.path.exists(self.cache_file):
58              with open(self.cache_file, "rb") as f:
59                  return f.read()
60
61      def write_calibration_cache(self, cache):
62          with open(self.cache_file, "wb") as f:
63              f.write(cache)
64
65  def get_int8_calibrator(calib_cache, calib_data, max_calib_size,
       preprocess_func_name, calib_batch_size):
66
67      if calib_cache != "" and os.path.exists(calib_cache):
68          calib_files = []
69      else:
70          if not calib_data:
71              raise ValueError("ERROR: No hi ha dades de calibracio")
72
73          calib_files = get_calibration_files(calib_data, max_calib_size)
74
75      import input_fn
76      if preprocess_func_name is not None:
77          preprocess_func = getattr(input_fn, preprocess_func_name)
78      else:
79          preprocess_func = input_fn.preprocess_imagenet
80
81      int8_calibrator = ImagenetCalibrator(calibration_files=calib_files,
82                                           batch_size=calib_batch_size,
83                                           cache_file=calib_cache,
84                                           preprocess_func=preprocess_func)
85      return int8_calibrator
86
87
88  def get_calibration_files(calibration_data, max_calibration_size=None,
       allowed_extensions=(".jpeg", ".jpg", ".png")):
89      calibration_files = [path for path in glob.iglob(os.path.join(
           calibration_data, "**"), recursive=True)
90                           if os.path.isfile(path) and path.lower().endswith(
                                allowed_extensions)]
91
92      if max_calibration_size:
93          if len(calibration_files) > max_calibration_size:
94              random.seed(42)
95              calibration_files = random.sample(calibration_files,
                    max_calibration_size)
96
97      return calibration_files
98
99
100 def build_engine(onnx_path, shape, preprocess, cache):
101     with trt.Builder(TRT_LOGGER) as builder, builder.create_network(1) as
           network, trt.OnnxParser(network, TRT_LOGGER) as parser, builder.
           create_builder_config() as config:
102         config.max_workspace_size = (256 << 20)
```

```
        with open(onnx_path, 'rb') as model:
            parser.parse(model.read())
        network.get_input(0).shape = shape
        config.set_flag(trt.BuilderFlag.INT8)
        config.int8_calibrator = get_int8_calibrator(cache, "./images/5000images
            /", 5000, preprocess, 128)
        engine = builder.build_engine(network, config)
        return engine


def save_engine(engine, file_name):
    buf = engine.serialize()
    with open(file_name, 'wb') as f:
        f.write(buf)

def main(args):
    engine_name = args.plan_file
    onnx_path = args.onnx_file
    prp_function = "preprocess_imagenet"
    cache ="./int8/int8.cache"
    batch_size = 1

    model = ModelProto()
    with open(onnx_path, "rb") as f:
        model.ParseFromString(f.read())

    d0 = model.graph.input[0].type.tensor_type.shape.dim[1].dim_value
    d1 = model.graph.input[0].type.tensor_type.shape.dim[2].dim_value
    d2 = model.graph.input[0].type.tensor_type.shape.dim[3].dim_value
    shape = [batch_size , d0, d1 ,d2]

    engine = build_engine(onnx_path, shape= shape, preprocess=prp_function,
        cache=cache)
    save_engine(engine, engine_name)


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('--onnx_file', type=str)
    parser.add_argument('--plan_file', type=str, default='engine.plan')
    parser.add_argument('--prp', type=str)
    parser.add_argument('--cache',type=str)
    args=parser.parse_args()
    main(args)
```

A.1.4. Inferència de 1 image TensorRT: `inferenceGPU.py`

```
import time
import sys
import random
import tensorrt as trt
import cv2
import numpy as np
import pycuda.driver as cuda
import input_fn

TRT_LOGGER = trt.Logger(trt.Logger.WARNING)
trt_runtime = trt.Runtime(TRT_LOGGER)

input_file = "ILSVRC2012_val_00000167.JPEG"
input_path = "/home/alizrui/TENSORRT/images/5000images/"

```

```
16 # words file
17 file_words = open("images/words.txt")
18 words = file_words.readlines()
19 file_words.close()
20
21 # file with names
22 file_images = open("images/list5000.txt")
23 images = file_images.readlines()
24 file_images.close()
25
26 serialized_plan_fp32 = sys.argv[1] #"resnet50.plan"
27 HEIGHT = 224
28 WIDTH = 224
29
30 _R_MEAN = 123.68
31 _G_MEAN = 116.78
32 _B_MEAN = 103.94
33
34 MEANS = [_B_MEAN,_G_MEAN,_R_MEAN]
35
36 def load_engine(trt_runtime, plan_path):
37     with open(plan_path, 'rb') as f:
38         engine_data = f.read()
39     engine = trt_runtime.deserialize_cuda_engine(engine_data)
40     return engine
41
42 def allocate_buffers(engine, batch_size, data_type):
43     # Buffers del host mab les dimesnions dels tensors dentrada i eixida
44     h_input_1 = cuda.pagelocked_empty(batch_size * trt.volume(engine.
         get_binding_shape(0)), dtype=trt.nptype(data_type))
45     h_output = cuda.pagelocked_empty(batch_size * trt.volume(engine.
         get_binding_shape(1)), dtype=trt.nptype(data_type))
46     # Buffers de GPU amb les mateixes dimensions
47     d_input_1 = cuda.mem_alloc(h_input_1.nbytes)
48     d_output = cuda.mem_alloc(h_output.nbytes)
49     # Stream de CUDA on es copiaran els resultats i es fara la inferencia
50     stream = cuda.Stream()
51     return h_input_1, d_input_1, h_output, d_output, stream
52
53 def load_images_to_buffer(pics, pagelocked_buffer):
54     preprocessed = np.asarray(pics).ravel()
55     np.copyto(pagelocked_buffer, preprocessed)
56
57 def do_inference(engine, pics_1, h_input_1, d_input_1, h_output, d_output,
      stream, batch_size, height, width):
58     load_images_to_buffer(pics_1, h_input_1)
59
60     with engine.create_execution_context() as context:
61         # Copiar dades a la GPU
62         cuda.memcpy_htod_async(d_input_1, h_input_1, stream)
63         # Executa inferencia
64         context.profiler = trt.Profiler()
65         context.execute(batch_size=1, bindings=[int(d_input_1), int(d_output)])
66         # Recupera prediccions de la GPU
67         cuda.memcpy_dtoh_async(h_output, d_output, stream)
68         # Sincronitzacio del stream
69         stream.synchronize()
70         # Torna el resultat
71         out = h_output
72         return out
73
74 # selecciona dada aleatoria
75 input_file = random.choice(images).split()[0]
76
```

```
# preprocess
image = cv2.imread(input_path + input_file)
image = input_fn.preprocess_imagenet(image)

# carrega el motor dinferencia
engine = load_engine(trt_runtime, serialized_plan_fp32)

# reserva buffers cuda
h_input, d_input, h_output, d_output, stream = allocate_buffers(engine, 1, trt.
    float32)

# inferencia
t1 = time.time()
out = do_inference(engine, image, h_input, d_input, h_output, d_output, stream,
     1, HEIGHT, WIDTH)
t2 = time.time()

# top 5 resultats
res = out.argsort()[-5:][::-1]

print("Image : "+ input_file + "\ntop[0] prob = " + str(out[res[0]]) + "  name
    = "+ words[res[0]] + "top[1] prob = " + str(out[res[1]]) + "  name = "+
    words[res[1]] + "top[2] prob = " + str(out[res[2]]) + "  name = "+ words[
    res[2]] + "top[3] prob = " + str(out[res[3]]) + "  name = "+ words[res[3]]
    + "top[4] prob = " + str(out[res[4]]) + "  name = "+ words[res[4]])
print("Total time: ", t2 - t1)
```

A.2 Funcions d'activació

A continuació està el codi C implementat en les funcions d'activació. La implementació en C utilitza clàusules de la llibreria OpenMP per a paral·lelitzar el codi. Els tensors d'entrada de la funció d'activació estan en x, mentre que el tensor resultat s'emmagatzema en y.

A.2.1. ReLU

```
int fn_matrix_relu_cpu(type *x, type *y, int rows, int cols) {
    int cells = rows * cols;
    #pragma omp parallel for
    for (int i = 0; i < cells; i++) {
        y[i] = (x[i] < 0.0) ? 0.0 : x[i];
    }
    return 1;
}

int fn_matrix_relu_der_cpu(type *m1, int rows, int cols, type *m2) {
    int cells = rows * cols;
    #pragma omp parallel for
    for (int i = 0; i < cells; i++) {
  m2[i] = (m1[i] > 0.0);
    }
    return 1;
}
```

A.2.2. Leaky ReLU

```
int fn_matrix_leakyrelu_cpu(type *x, type *y,
                            int rows, int cols, float alpha) {
    #pragma omp parallel for
    for (int i = 0; i < cells; i++) {
        y[i] = (x[i] < 0.0) ? alpha * x[i] : x[i];
    }
    return 1;
}

int fn_matrix_leakyrelu_der_cpu(type *m1, int rows, int cols,
                                type *m2, float alpha) {
    int cells = rows * cols;
    #pragma omp parallel for
    for (int i = 0; i < cells; i++) {
      m2[i] = (m1[i] > 0.0) ? 1 : alpha;
    }
    return 1;
}
```

A.2.3. Parametric ReLU

```
int fn_matrix_parametricrelu_cpu(type *x, type *y,
                                 int rows, int cols, float alpha, type *z) {
    int cells = rows * cols;
    #pragma omp parallel for
    for (int i = 0; i < cells; i++) {
        y[i] = (x[i] < 0.0) ? alpha * x[i] : x[i];
        z[i] = (x[i] < 0.0) ? x[i] : 0.0 ; // gradient of activation
    }
    return 1;
}

int fn_matrix_parametricrelu_der_cpu(type *m1, int rows, int cols,
                                     type *m2, float alpha) {
    int cells = rows * cols;
    #pragma omp parallel for
    for (int i = 0; i < cells; i++) {
      m2[i] = (m1[i] > 0.0) ? 1 : alpha;
    }
    return 1;
}
```

A.2.4. ELU

```
int fn_matrix_elu_cpu(type *x, type *y, int rows, int cols, float alpha) {
    int cells = rows * cols;
    #pragma omp parallel for
    for (int i = 0; i < cells; i++) {
        y[i] = (x[i] < 0.0) ? alpha * (exp(x[i]) - 1) : x[i];
    }
    return 1;
}

int fn_matrix_elu_der_cpu(type *m1, int rows, int cols,
                          type *m2, float alpha) {
    int cells = rows * cols;
    #pragma omp parallel for
    for (int i = 0; i < cells; i++) {
      m2[i] = (m1[i] > 0.0) ? 1 : alpha + (alpha * (exp(m1[i]) - 1));
```

```
16      }
17      return 1;
18  }
```

A.2.5. Swish

```
1  int fn_matrix_swish_cpu(type *x, type *y, int rows, int cols) {
2      int cells = rows * cols;
3      #pragma omp parallel for
4      for (int i = 0; i < cells; i++) {
5          y[i] = x[i] / (1 + exp(-x[i]));
6      }
7      return 1;
8  }
9
10 int fn_matrix_swish_der_cpu(type *m1, int rows, int cols, type *m2) {
11     int cells = rows * cols;
12     #pragma omp parallel for
13     for (int i = 0; i < cells; i++) {
14       m2[i] = (m1[i] / (1 + exp(-m1[i]))) +
15               (1/(1 + exp(-m1[i])) *
16               (1 - (m1[i] / (1 + exp(-m1[i])))));
17     }
18     return 1;
19 }
```

A.2.6. GELU

```
1  int fn_matrix_gelu_cpu(type *x, type *y, int rows, int cols) {
2      int cells = rows * cols;
3      #pragma omp parallel for
4      for (int i = 0; i < cells; i++) {
5          y[i] = x[i] * (0.5 * (1.0 + erf(x[i] / sqrt(2.0))));
6      }
7      return 1;
8  }
9
10 int fn_matrix_gelu_der_cpu(type *m1, int rows, int cols, type *m2) {
11     int cells = rows * cols;
12     #pragma omp parallel for
13     for (int i = 0; i < cells; i++) {
14       float omega = (1.0 + erf(m1[i] / sqrt(2.0))) / 2.0;
15         float px = (1.0 / sqrt(2 * M_PI)) * exp((-1.0/2.0) * pow(m1[i],2));
16       m2[i] = omega + m1[i] * px;
17     }
18     return 1;
19 }
```

APÈNDIX B
Topologies de xarxes neuronals

Per a descriure les topologies utilitzades s'han esquematitzat la primera de les dues versions de cada tipus. La segona versió de les dues que composen cada grup és una versió amb més capes i més profunda en tots els casos, però sols en la primera ens podem il·lustrar i entendre com són aquests models.

B.1 Resnet

La xarxa `Resnet50` està composta per blocs de capes, on l'operació principal és la convolució i on s'utilitza l'aprenentatge residual com s'ha comentat en la memòria. En la figura B.1 podem observar l'estructura d'aquesta xarxa. Com podem observar, els blocs `conv1`, `conv2` i `conv3` tenen una profunditat de 3 capes amb 3 convolucions seqüencials cada bloc. En la xarxa hi ha 16 blocs com aquest, el que ho fan $16 * 3 = 48$ convolucions de profunditat més 1 convolució que hi ha al principi i una capa *fully connected* al final, amb un total de 50 capes de profunditat. D'aquí el nom de la xarxa. En la `Resnet101` l'estructura és la mateixa però fins a arribar a 101 capes de profunditat.

B.2 Inception

L'arquitectura `Inception` és caracteritza per l'ús de blocs inception, format per varies capes. Com podem veure en la figura B.2 podem vore l'estructura dels blocs inception en la xarxa `InceptionV1`. La xarxa utilitzada té 24 capes de profunditat. La versió `InceptionV4` té molta més complexitat, on s'utilitzen diferents tipus de blocs inception amb diferents combinacions.

B.3 VGG

El model `VGG16` s'estructura en 13 capes convolucionals, combinades amb operacions de *pooling* i 3 capes *fully connected* al final de la xarxa, formant una profunditat de 16 capes. La versió `VGG19` conta amb 3 capes convolucionals més, afegint més complexitat computacional al model. En la figuera B.3 podem veure l'estructura de la `VGG16`.

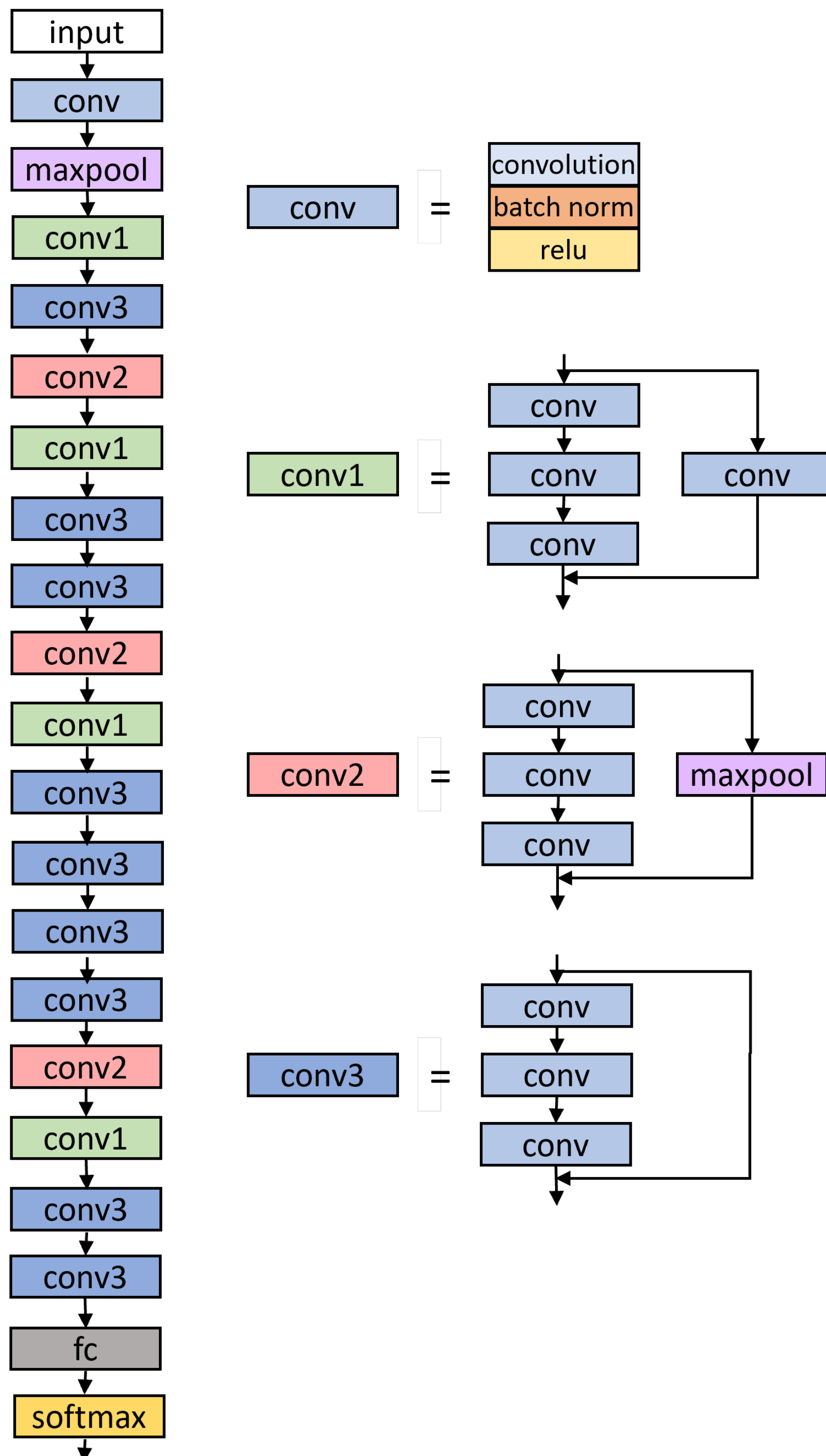

Figura B.1: Xarxa `Resnet50`

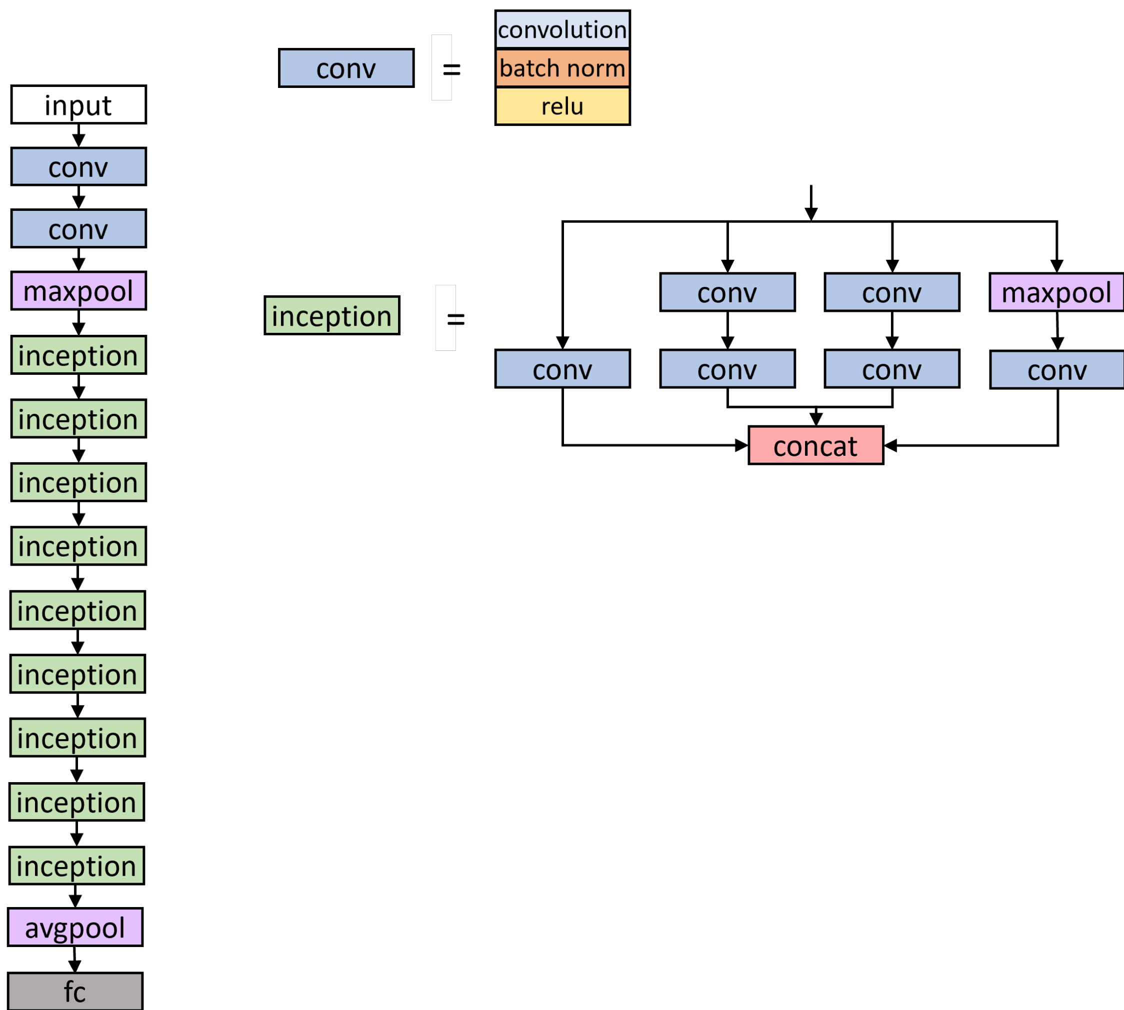

Figura B.2: Xarxa `InceptionV1`

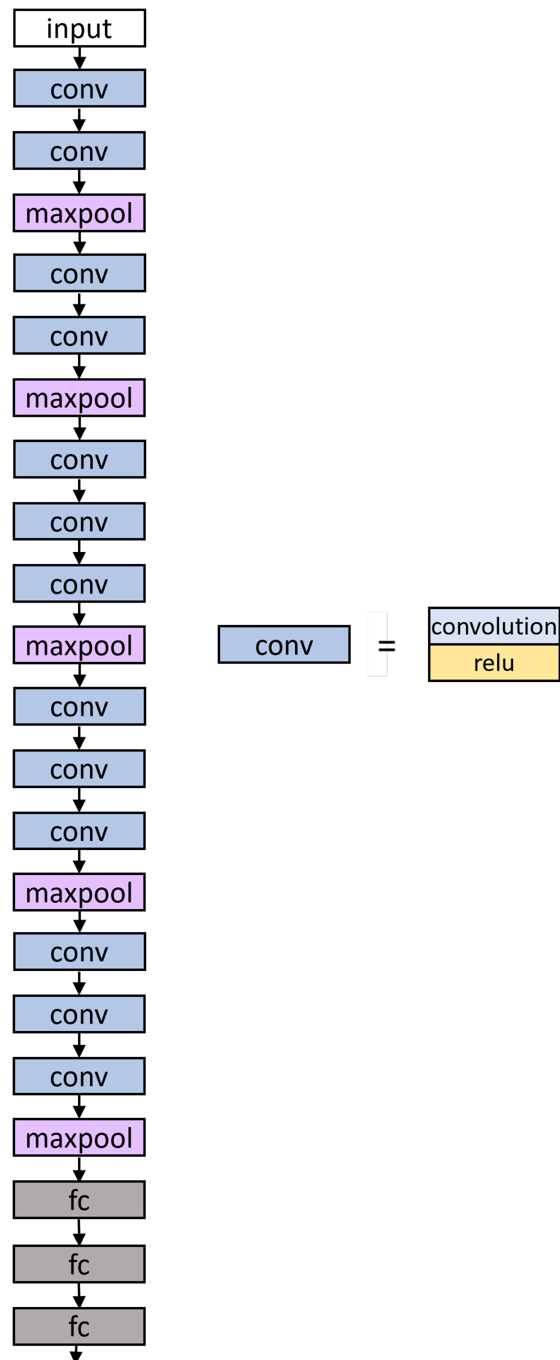

Figura B.3: Xarxa `VGG16`