Implementation of a Convolutional Neural Network (CNN) on a FPGA for Sign Language’s Alphabet recognition

By

PABLO CORREA GÓMEZ

Centro de Electrónica Industrial
UNIVERSIDAD POLITÉCNICA DE MADRID

A dissertation submitted to the Universidad Politécnica de Madrid in accordance with the requirements of the GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES in the Escuela Técnica Superior de Ingenieros Industriales.

JULY 208
Since 2012, with the introduction of Convolutional Neural Networks (CNN) for image recognition, great improvements have been made in terms of accuracy, topologies, and understanding of the challenges associated with the image recognition. Several situations where they have already been proven successful include self-driving cars, image tagging and face recognition. Most of the development have been centered in increasing precision while trying to have as little computations as possible. However, most of the topologies and applications still require expensive and power hungry Graphical Processors (GPUs) to be able to deliver fast responses. Therefore, these systems are most of the time located in the very same place where the data is generated or in specially designed data centers. Recently, there has been a growing interest and research towards low-resource architectures for its use in embedded systems, although most of it is still in a theoretical approach.

In addition although CNNs have applications different than image recognition, this last one has been proven quite controversial due to the use (or misuse) that some companies and governments have done of them, while most of the research done by universities has been more theoretical.

The objective of this bachelor thesis is to use the current theoretical knowledge about CNNs to prove their use in embedded systems while at the same time developing an application that can be beneficial for the society as a whole. According to this objective, the thesis aims to be able to get photos from the Swedish deaf’s people sign language alphabet and identify the letters associated with each of the signs, working on a real time system.

For that purpose, big amounts of data have been collected, analyzed and processed and the (embedded systems’ friendly) Zynqnet CNN topology has been modified to fit the application. All together allow more than 85% of the images to be successfully identified using a regular GPU training system.

In addition, a custom, high throughput hardware accelerator for that topology has been designed to be placed in an FPGA. Similar precision results than using the GPU have been gotten while reducing space, weight and power consumption. The FPGA accelerator will also reach real-time performance, computing the results for each image in less than 1 second.
DEDICATION AND ACKNOWLEDGEMENTS

To all the people that for any random reason spent some of their time following a video to help with the data collection. Anything would have been possible without you. You are the real heroes of this project. I would also like to thank both of my supervisors: Liang Liu for spending his time and letting me step into the project with no need and Andrés Otero for the patience and taking care of a thesis developed in another university. A mis padres, que pase lo que pase siempre están ahí.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Resumen en español</strong></td>
<td>1</td>
</tr>
<tr>
<td>Introducción</td>
<td>1</td>
</tr>
<tr>
<td>Objetivos</td>
<td>2</td>
</tr>
<tr>
<td>Contexto científico</td>
<td>2</td>
</tr>
<tr>
<td>Entrenamiento y recogida de datos</td>
<td>6</td>
</tr>
<tr>
<td>Acelerador en hardware</td>
<td>10</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>17</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>17</td>
</tr>
<tr>
<td>1.2 Scope and Structure</td>
<td>18</td>
</tr>
<tr>
<td><strong>2 State of the art</strong></td>
<td>21</td>
</tr>
<tr>
<td>2.1 Convolutional Neural Networks</td>
<td>21</td>
</tr>
<tr>
<td>2.1.1 Theoretical background</td>
<td>21</td>
</tr>
<tr>
<td>2.1.2 Training and frameworks</td>
<td>24</td>
</tr>
<tr>
<td>2.2 High Level Synthesis and FPGAs</td>
<td>24</td>
</tr>
<tr>
<td>2.3 Current solutions: GPUs vs Embedded systems</td>
<td>25</td>
</tr>
<tr>
<td>2.4 Zynqnet</td>
<td>27</td>
</tr>
<tr>
<td><strong>3 Training, data collection and environment setup</strong></td>
<td>29</td>
</tr>
<tr>
<td>3.1 Data set</td>
<td>29</td>
</tr>
<tr>
<td>3.1.1 Data augmentation</td>
<td>29</td>
</tr>
<tr>
<td>3.1.2 Data organization</td>
<td>31</td>
</tr>
<tr>
<td>3.2 Training</td>
<td>31</td>
</tr>
<tr>
<td>3.2.1 Matlab - First steps</td>
<td>32</td>
</tr>
<tr>
<td>3.2.2 Caffe + Digits - Professional framework + GUI</td>
<td>35</td>
</tr>
<tr>
<td>3.2.3 Ristretto - Fixed-point analysis</td>
<td>37</td>
</tr>
<tr>
<td>3.3 Results and analysis</td>
<td>39</td>
</tr>
<tr>
<td><strong>4 Hardware accelerator</strong></td>
<td>43</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

4.1 Design methodology ................................................. 43
  4.1.1 HLS methodology ............................................. 43
  4.1.2 Development methodology ..................................... 44
4.2 Design overview ................................................... 45
4.3 Pipelined core ..................................................... 46
  4.3.1 Loops organization ............................................ 46
  4.3.2 Caching strategy .............................................. 51
4.4 Pipelined core improvements ..................................... 52
  4.4.1 Parameter configuration: layer definition .................... 53
  4.4.2 Memory controller: arrays merging .......................... 53
  4.4.3 Parameter configuration: kernel 3×3 stride 2 and kernel 1×1 .... 53
  4.4.4 Fixed point implementation ................................. 54
  4.4.5 Fire modules: double memory scheme ......................... 55
4.5 Results and analysis .............................................. 55
  4.5.1 Synthesis results ............................................. 56
  4.5.2 Limitations and possible improvements ....................... 56
5 Conclusions and future development .......................... 59
6 Budget and planning ................................................. 61

List of Figures ......................................................... 63
List of Tables .......................................................... 65

A Data manipulation and augmentation .......................... 67
B Ristretto Models ..................................................... 73

Bibliography ............................................................ 85
Introducción

La inteligencia artificial ha sido un campo de desarrollo activo en sistemas informáticos desde hace más de 40 años [2]. Sin embargo, ha sido solo recientemente cuando su desarrollo ha saltado a otras disciplinas. Ello se debe, fundamentalmente, al cada vez mayor poder de cálculo disponible. A pesar de todo, la inteligencia artificial es todavía considerada por la mayoría de la sociedad como un tipo de *magia negra*, con reacciones que van desde el asombro y la incredulidad hasta el miedo y el rechazo. A ello han contribuido la falta de transparencia en la mayor parte de sus aplicaciones y diversos escándalos debidos tanto a fallos en las implementaciones como a estrategias opacas de recogida de datos. Sin embargo, las redes neuronales y todo el campo de desarrollo relacionado tienen una gran cantidad de aplicaciones y un extraordinario potencial para mejorar la calidad de vida de miles de ciudadanos.

Una de las posibles áreas de aplicación es la transcripción en tiempo real del lenguaje de signos (para la comunicación con personas sordas y/o mudas). Ello haría posible entablar una conversación a pesar de que alguna de las partes desconozca el lenguaje de signos. A día de hoy, con el estado actual de las tecnologías de inteligencia artificial para reconocimiento de imágenes y una importante capacidad de desarrollo, debería ser posible llevar a cabo dicha transcripción con distintos niveles de desempeño.

El lenguaje seleccionado ha sido el sueco por haber desarrollado el proyecto en este país. Sin embargo, el uso de redes neuronales para realizar la transcripción se basa en la facilidad de incluir nuevos idiomas al modelo, pues en caso de existir una implementación ya existente para un cierto idioma no sería necesario modificar la arquitectura, sino simplemente datos suficientes que permitan adaptación de la red.

En la actualidad los sistemas de inteligencia artificial tienen dos soportes fundamentales: centros de datos especializados o potentes infraestructuras cercanas al lugar de generación de datos. En el caso de la aplicación que se presenta, ninguno de estos soportes resulta apropiado, pues existe la necesidad de que el sistema sea relativamente portátil, disminuyendo la potencia disponible. Esto, junto con la experiencia con FPGAs de los departamentos de la Universidad de Lund y la ETSII de la UPM, ha llevado a desarrollar un acelerador de la red neuronal capaz de ser implementado en una FPGA para mejorar la velocidad de procesamiento habitual en un sistema de pocos recursos. Dentro de las distintas tecnologías que podrían hacer esto posible, se
ha escogido desarrollar el proyecto usando High Level Synthesis (Síntesis de Alto Nivel o HLS por sus siglas en inglés), que permite el diseño de hardware usando un lenguaje de alto nivel como C, y que será explicado con más detalle en la Sección.

**Objetivos**

Poder desarrollar un sistema capaz de analizar y transcribir video de una persona utilizando un cierto lenguaje de signos resulta claramente fascinante. Sin embargo, la complejidad del proyecto, la falta de experiencia con redes neuronales de los departamentos y la incapacidad para recoger los millones de datos que serían necesarios, obligaron a reducir el alcance del proyecto en dos ámbitos:

- En lugar de transcribir todo el lenguaje, solo se usará el alfabeto, consistente en 26 caracteres fijos y 3 que implican movimiento. La falta de conocimiento del lenguaje de signos sueco junto con la inviabilidad de recoger suficientes datos suponen las principales razones para esta reducción. Además, el análisis del alfabeto supone un buen punto de partida para estudiar la posibilidad de llevar el proyecto al siguiente nivel de desarrollo.

- El uso de los principales 26 caracteres del alfabeto permite además realizar un análisis en estático del problema, sin la necesidad de estudiar movimientos, lo que complicaría en exceso la red neuronal para el estado actual de la tecnología. De esta manera se decide utilizar solo los principales 26 caracteres del alfabeto latino.

Una vez que el alcance del proyecto ha sido definido, es necesario fijar tanto la metodología como los objetivos tangibles. Al tratarse de un proyecto con dos partes: entrenamiento de la red neuronal y diseño del acelerador en hardware, el las que los resultados están relacionados y la segunda depende de la primera, se comenzará con el desarrollo de la red neuronal y la metodología de cada parte se expondrá en la sección correspondiente. En cuanto a los objetivos, se establece un mínimo de un 80% de precisión para la red neuronal mientras que el rendimiento mínimo que se espera obtener con la FPGA es poder procesar una imagen por segundo.

**Contexto científico**

**Redes neuronales convolucionales**

Las redes neuronales son una familia de arquitecturas de computación inspiradas en el cerebro. En ellas, cada uno de los elementos (o neuronas) recibe una serie de señales de entrada de otras neuronas, con las cuales genera una señal de salida (o activación) que se conecta a una cierta cantidad de otras neuronas. Aunque existen infinidad de arquitecturas, las más importantes y las únicas utilizadas para el desarrollo de este TFG organizan las neuronas en capas sucesivas.
De esta manera una cierta neurona solo puede enviar y recibir señales de las neuronas de las capas posterior y anterior, respectivamente. Además, en general todas las neuronas de cada una de las capas están conectadas entre sí, generando millones de posibles conexiones en un proceso conocido como conexión completa o fully connectivity.

**Funcionalidad y entrenamiento** En el caso de estudio que nos interesa, las redes neuronales se utilizan para clasificar una cierta información de entrada dentro de una serie de opciones predefinidas. Para ello son necesarios lo que se conoce como pesos, que son los valores que utilizan (habitualmente a través de operaciones de multiplicación y suma) cada una de las neuronas para generar su salida partiendo de las numerosas entradas. Sin embargo, la obtención de los valores necesarios para todos los pesos (se cuentan por millones) que generan una clasificación correcta para cualquier entrada es un proceso complejo. Habitualmente se dispone de una serie de datos de los que se conoce la clasificación correcta y de los que se hace uso para actualizar sucesivamente los pesos, mejorando la precisión y disminuyendo el loss, que es una medida de la diferencia entre la clasificación real y el resultado devuelto por la red. Este proceso es conocido como entrenamiento o aprendizaje y es la razón por la que es necesaria la obtención de una gran cantidad de datos para el uso de las redes neuronales.

**Redes neuronales convolucionales** Las redes neuronales convolucionales (o CNNs por sus siglas en inglés) son un tipo especial de redes neuronales especialmente útiles en el procesamiento de información bidimensional, como imágenes.

Cuando las redes neuronales son utilizadas para procesar imágenes, el uso de conexión completa o fully connectivity genera una inmensa cantidad de conexiones que hacen inviable la implementación. Por suerte, la localidad de información en las imágenes hace innecesario el uso de la conexión completa. Ello se debe a que para localizar un perro en el centro de una imagen, no es necesario considerar los píxeles de las esquinas. De esta manera, las redes neuronales convolucionales sustituyen las capas completamente conectadas por capas convolucionales, reduciendo las operaciones y pesos sin afectar al rendimiento.

**Capa convolucional** La entrada de una capa convolucional consiste en un stack de imágenes ($ch_{in}$) de dimensiones $h_{in} \times w_{in}$ llamados input feature maps. Cada capa produce como salida un stack de imágenes ($ch_{out}$) de dimensiones $h_{out} \times w_{out}$, llamados output feature maps. Cada capa también contiene un stack de $ch_{in} \times ch_{out}$ kernels (o filtros en 2-D) de tamaño $k \times k$ (con $k$ habitualmente primo impar entre 1 y 11), que contiene los pesos entrenados. Además, a cada elemento de salida se le suele sumar un bias, único para cada $ch_{out}$.

Cada pixel de entrada se convoluciona con uno de los filtros en 2D, que se desplazan por la imagen de entrada un cierto valor conocido como stride para general cada uno de los píxeles de salida. Para generar la salida completa, los resultados de la convolución se suman a lo largo de todos los canales de entrada y se les suma el bias. En caso del uso
de filtros superiores a $1 \times 1$, las imágenes de entrada se suelen completar con píxeles de valores cero para evitar que se reduzca el tamaño de la imagen resultado. El número de píxeles añadidos se conoce como padding. Un ejemplo gráfico se puede observar en la Figura 0.1.

Es también habitual que tras cada capa convolucional se aplique una cierta función no lineal a todos los output feature maps. El objetivo es permitir la codificación de patrones más complejos que no pueden ser identificados utilizando simplemente operaciones de multiplicación y suma. La función más habitual así como la única utilizada en este proyecto es conocida como ReLU, y su función matemática es: $y = \max(0,x)$.

**FPGAs y High Level Synthesis**

Field Programmable Gate Arrays o FPGAs es como se conoce a unos dispositivos hardware que permiten la implementación de circuitos electrónicos personalizados. Estos dispositivos contienen una gran cantidad de recursos lógicos configurables, como LUT, DSPs o BRAMs, cuyo uso y cuyas interconexiones pueden ser configurados para modificar su funcionamiento general. A día de hoy, la gran cantidad de recursos lógicos que contienen las FPGAs permite generar extensos y potentes circuitos, por lo que serán el soporte para la implementación del acelerador.

Durante los últimos 30 años, estos circuitos han sido diseñados utilizando un conjunto de lenguajes de descripción hardware conocidos como HDL. En ellos el funcionamiento del circuito que se desea generar se describe mediante descripciones principalmente de transferencia entre registros (RTL), permitiendo un gran control en el diseño. Sin embargo, la metodología asociada a estos lenguajes requiere extensos recursos de desarrollo y resulta excesivamente lenta para grandes proyectos.

Por esta razón, para este TFG se ha decidido utilizar la tecnología de High Level Synthesis, que permite describir hardware utilizando un lenguaje de alto nivel como C. A este se le añaden directivas de compilador o pragmas que permiten enviar cierta información añadida para tener un mayor control sobre el circuito que generará la herramienta utilizada en el diseño. En su primer paso, cualquiera de las herramientas disponibles actualmente en el mercado hacen...
una traducción del lenguaje de alto nivel a uno de los lenguajes HDL, lo que se conoce como síntesis. Posteriormente, y una vez comprobado el correcto funcionamiento, se puede proceder a la implementación en la FPGA.

**Soluciones actuales**

A día de hoy la implementación de redes neuronales convolucionales en tarjetas gráficas o GPUs para reconocimiento de imagen está ampliamente desarrollada. Recientemente existe también cierto interés por el uso de FPAGs para crear circuitos que aceleren las redes neuronales con un carácter bastante general, especialmente con aplicación en centros de datos[17]. A pesar de ello, salvo un trabajo de ETH Zurich[6] que presenta una arquitectura específica para FPAGs conocida como Zynqnet, así como un acelerador capaz de procesar una imagen cada dos segundos; poca información ha sido publicada, por lo que sigue siendo un campo con mucho potencial de desarrollo.

**Zynqnet** Para este trabajo, Zynqnet, la arquitectura presentada por D. Gschwend de ETH Zurich se usará como base para el entrenamiento de la red neuronal para la que posteriormente se diseñará el acelerador. Sin embargo, importantes cambios se han desarrollado con respecto al paper de Zynqnet:

- **Arquitectura** Originalmente Zynqnet está diseñada para clasificar imágenes en más de 1000 categorías de una base de datos pública usada habitualmente para el entrenamiento y comparación de distintas redes, conocida como ImageNet[15][5]. Para la implementación aquí presentada la última capa, que es la que realiza la clasificación, ha sido sustituida y será entrenada para clasificar únicamente los 26 símbolos del alfabeto de signos sueco. Los pesos del resto de capas serán reutilizados, pues en principio son capaces de detectar con precisión líneas y contornos en las imágenes. Además, la última operación/capa de la red, consistente en una exponencial, y usada para el cálculo de probabilidades será eliminada, pues el único interés de este proyecto es obtener la predicción de mayor probabilidad, lo que se puede extraer de la capa anterior.

- **Alcance del hardware** Aunque durante el desarrollo de Zynqnet también se diseña un acelerador en hardware, una parte importante del diseño está implementado en software (organización y correcta ejecución de las capas y gestión de la memoria RAM), al hacer uso de una FPGA que dispone también de un procesador ARM, dentro de la familia de Xilinx® Zynq. Para este trabajo se ha decidido realizar una implementación en hardware puro. Ello se debe a que aunque habitualmente es posible que exista una CPU disponible para trabajar con el acelerador, esta no tendrá dedicación exclusiva y se desconoce la cantidad de memoria y tiempo de ejecución de que se dispondrá.
Punto fijo
La mayor parte de las infraestructuras para redes neuronales así como Zynqnet hacen uso de datos en punto flotante de 32 bits para llevar a cabo todas las operaciones, con el consiguiente coste en términos de potencia y complejidad que estas requieren. Sin embargo, la implementación que aquí se presenta utiliza únicamente operaciones en punto fijo. Para realizar un análisis preliminar sobre la viabilidad, se usará Ristretto[7], una herramienta pensada precisamente para ello.

Para finalizar y con el objetivo de que el lector pueda comprender con mayor facilidad el diseño del acelerador, se presenta la arquitectura de Zynqnet modificada. Esta cuenta con una primera capa convolucional seguida de 8 módulos idénticos conocidos como fire modules, el clasificador sustituido para la presente aplicación y una última capa que realiza la media de cada una de las salidas con el objetivo de decidir cuál es la letra asociada con la imagen de entrada. Estos 8 módulos constan cada uno de 3 capas convolucionales: una primera (squeeze layer) a cuya salida están conectadas las otras dos (expand layers), cuyas salidas se concatenarán para servir de entrada al siguiente módulo al clasificador. Esta arquitectura es herencia de una red anterior conocida como SqueezeNet[8] que fue modificada para funcionar más eficientemente en hardware. Además, salvo el clasificador, todas las capas cuentan con una ReLU al final.

Entrenamiento y recogida de datos

Una vez definidos los objetivos, es necesario recoger hacer una recogida de datos para entrenar la red neuronal. La falta de experiencia inicial con redes neuronales ha supuesto el uso de distintas herramientas para el entrenamiento, comenzando por aquellas más sencillas y posteriormente avanzando hacia otras de uso profesional. Además, se dispondrá durante el proyecto de un ordenador moderno y muy potente con una GPU de última generación (Nvidia GeForce GTX 1080 Ti) para la aceleración y obtención de resultados.

Base de datos

Uno de los principales problemas del entrenamiento de redes neuronales es la obtención y aprovechamiento de la base de datos. Para la recogida se desarrollaron diversas reuniones con voluntarios a los que se les hizo fotos o vídeos de las manos para procesarlos posteriormente. Finalmente se obtuvo una base de datos de más de 2600 imágenes para el entrenamiento y 160 para validar los resultados.

Aumento de datos y overfitting

Uno de los principales problemas que existen en el entrenamiento de las redes neuronales es lo que se conoce como overfitting. Consiste en la existencia de una diferencia importante entre la precisión que se obtiene para las imágenes que son entrenadas y aquellas que solo se utilizan para validar los resultados. Realizar entrenamientos de un número
reducido de pesos es una de las estrategias seguidas cuando no existen suficientes datos. Sin embargo, su eficacia puede estar limitada y casi siempre se recurre a lo que se conoce como aumento de datos o data augmentation[18][14]. Esta estrategia consiste en realizar modificaciones a las imágenes que se utilizarán para entrenar la red neuronal, y añadirlas al entrenamiento. Las modificaciones que se han realizado consisten reflejar las imágenes, añadir ruido, modificaciones en el brillo y saturación, rotaciones, etc.

**Entrenamiento**

El proceso de entrenamiento de las redes neuronales es un proceso complejo en el que influyen muchos factores cuya relación habitualmente no es trivial, por lo que la experiencia resulta habitualmente un factor determinante. Para poder contar con esa experiencia, los primeros pasos se darán utilizando Matlab® Neural Network Toolbox™[10], un entorno sencillo y con numerosos ejemplos para posteriormente utilizar Caffe[9], mucho más poderoso y desarrollado por UC Berkeley junto con Digits, un entorno gráfico desarrollado para Caffe por Nvidia.

**Matlab - Primeros pasos** Durante el entrenamiento con Matlab se aprendió a utilizar y comprender los parámetros más importantes para el entrenamiento. Todos ellos están relacionados con lo que se conoce como learning rate o ratio de entrenamiento. Este valor es el parámetro que determina en qué cantidad se modificarán los pesos cada vez que sean actualizados. Este valor se puede alterar de diversas maneras durante el entrenamiento y ello suele tener un efecto importante sobre la precisión alcanzada.

Durante el entrenamiento con Matlab y todavía con una base de datos reducida, los primeros resultados fueron alentadores, con precisiones superiores al 50%. Tras dedicar bastante tiempo a la familiarizarse con los distintos parámetros, ajustarlos convenientemente y aumentar la base de datos hasta unas 6500 imágenes, se obtuvieron los resultados presentados en la Figura 0.2.
**Caffe + Digits**  Una vez que las distintas opciones disponibles en Matlab han sido agotadas y todavía existiendo un overfitting importante, es necesario mejorar los entrenamientos utilizando distintas estrategias de modificación del ratio de entrenamiento, pues lo más recomendado es utilizar una modificación lineal[11], no disponible en Matlab. Todo ello con un aumento progresivo de la base de datos debe permitir alcanzar el 80% de precisión que se presenta como objetivo.

Tras realizar distintas modificaciones, aumentar la base de datos hasta las 2600 imágenes que se convierten en más de 13000 tras el aumento y ante la respuesta de la red a los entrenamientos únicamente del clasificador (la precisión en el entrenamiento no alcanza el 100% a pesar de existir un overfitting importante), se decide entrenar también otras capas, obteniendo excelentes resultados, pues la precisión en las imágenes utilizadas para la validación supera el 86%, con margen suficiente para la aplicación de las operaciones en punto fijo. Los resultados pueden observarse en la Figura 0.3


Una vez la red ha sido entrenada con Caffe, se utiliza Ristretto para estudiar la viabilidad de las operaciones en punto fijo. Los primeros resultados son todo un éxito, pues la precisión se mantiene prácticamente constante mientras que solo son necesarios 8 bits para realizar las operaciones, reduciendo en un factor 4 la memoria necesaria para la red. A pesar de ello, el hecho de que Ristretto simule punto fijo supone algún pequeño problema de coherencia entre los resultados de capas que serán concatenadas, lo que obligó a modificar ligeramente el modelo resultante. Además, durante la implementación en hardware se detectaron algunos errores importantes de precisión que pudieron ser arreglados tomados un mayor número de bits significativos durante las primeras capas de la red. Tras reentrenar ambos modelos usando punto fijo, el segundo dio resultados extraordinarios, superando en un 87% en la validación, como se puede ver en la Figura 0.4. Además, los dos modelos usados se encuentran disponibles en el Apéndice B.
Análisis y resultados

El resultado final del entrenamiento supera en más de un 7% al objetivo fijado al principio del proyecto, por lo que debería haber margen suficiente para la pérdida de precisión que se espera de la implementación en hardware. Sin embargo, el proceso de entrenamiento no puede considerarse perfecto, pues todavía existe un overfitting importante. Ello, sumado a que la precisión de las imágenes de entrenamiento alcanza con facilidad un 100%, sugiere que todavía existe margen de mejora. Para poder explotarlo sería necesario aumentar la base de datos, no disponiendo de los recursos temporales y humanos para ello.

En cuanto al análisis de resultados que no resultan obvios a vista de las gráficas, cabe mencionar dos puntos importantes. El primero es que una disminución en el loss supondría seguramente una reducción en el número de bits necesarios para las operaciones en punto fijo, pues distintas pruebas con ristretto y los modelos en los que solo se entrenan el clasificador o la primera capa dieron lugar a soluciones con 16 bits. Además, el entrenamiento de todas las capas tiene como resultado adicional una diferenciación mucho más brusca entre las distintas predicciones. Así como cuando solo se entrena el clasificador para la mayor parte de las imágenes existen varias letras con probabilidades apreciables, al entrenar todas las capas habitualmente existe una única letra con una probabilidad muy cercana al 100%. Este comportamiento, que podría considerarse un error en la mayor parte de los modelos matemáticos que se utilizan para el análisis de redes neuronales puede, por el contrario, resultar beneficioso en este proyecto, pues hace los errores de precisión a la hora de la implementación en hardware mucho menos críticos.
Acelerador en hardware

Metodología

Debido al uso de la tecnología de HLS y a la complejidad del acelerador, es muy importante definir la metodología con anterioridad para evitar errores y asegurar una buena organización del proyecto. En cuanto al código, debido a recomendaciones obtenidas, se utilizará C++ para poder utilizar principios de programación orientada a objetos. Sin embargo, la incapacidad de HLS de gestionar los punteros que generan los objetos, se hará uso de \textit{namespaces} para ofrecer modularidad al proyecto.

\textbf{HLS} Para asegurar una buena relación entre el código escrito en C++ y el hardware diseñado, es necesario seguir la metodología recomendada por Xilinx®, que consta de tres etapas: simulación, consiste en compilar y el ejecutar el código en software utilizando un banco de pruebas o \textit{test-bench} para garantizar su correcta funcionalidad; síntesis, que genera los diferentes archivos en un lenguaje HDL a partir del código y las pragmas; y co-simulación, que ejecuta una simulación en hardware de los archivos generados en la síntesis y comprueba los resultados utilizando el mismo banco de pruebas que durante la simulación software.

\textbf{Diseño} En cuanto al acelerador en sí, se ha decidido seguir una metodología en la que inicialmente se desarrolla y paraleliza una convolución básica con un kernel de $3 \times 3$ con stride 1 y posteriormente se añaden mejoras que permiten la ejecución completa de la red. La principal razón para la elección de esta metodología es que realizar la paralelización del hardware es posiblemente la parte más compleja de todo el proyecto. Retrasarla demasiados pasos en la implementación podría suponer una importante fuente de problemas y errores que se evitan realizándola cuando el proyecto es todavía sencillo.

\textbf{Visión global del diseño}

Desde una perspectiva general, el diseño cuenta con 4 partes destacadas, que intercambian información entre ellas para un correcto funcionamiento:

- **Núcleo paralelizado** Se encarga de la ejecución de las convoluciones, recibiendo los valores de entrada, salida y los pesos, pero sin tener detalles de las memorias que se acceden.

- **Módulo de control** Responsable de la correcta ejecución de arquitectura de la red, ejecutando de manera secuencial distintas capas en el núcleo, al que envía las memorias que serán accedidas. Finalmente realiza la media de la última imagen para dar como resultado la letra predicha. También permite modificar de manera externa los pesos cuando sea necesario.
• **Controlador de memoria** Recibe información del núcleo y del módulo de control para determinar las posiciones de memoria que tienen que ser leídas o escritas por el núcleo.

• **Memorias** Son 5: una ROM para guardar la arquitectura de la red y los detalles de cada capa (en total 1456 bits); 2 RAM para guardar los pesos (1.789.884 palabras) y los bias (3.450 palabras), pudiendo ser ambos modificados de forma externa en caso de que sea necesario actualizar el modelo; y 2 RAM (196.608 y 1.048.576\(^1\) palabras) para guardar los valores de entrada y salida de cada capa. El hecho de que para la entrada/salida solo sean necesarias dos memorias radica en la ejecución secuencial de las distintas capas, pues una vez finalizada una capa, su entrada ya no es útil y la memoria en la que está guardada puede ser reutilizada como la salida de la siguiente. El objetivo del proyecto es que todas las memorias sean internas (**on-chip**) para evitar el uso de memoria RAM y así simplificar el diseño.

**Núcleo paralelizado**

Como punto más importante del acelerador es también aquel que requiere una mayor complejidad de diseño y de análisis preliminar.

**Bucles** Un análisis de la operaciones involucradas en una capa convolucional (ver Sección ) resulta en la identificación de 6 bucles anidados al final de los cuales existe una operación de multiplicación y acumulación (MAC). Estos bucles suponen iterar sobre los canales de entrada y salida; sobre las filas y las columnas de las imágenes de entrada (o salida); y sobre las filas y columnas del kernel. La posición en la que estos bucles sean colocados y la cantidad y extensión de su paralelización determinará la arquitectura final del acelerador. Un análisis de las operaciones revela que solo hay dos arquitecturas posibles al desenrollar cualquiera de los bucles, que pueden observarse en la Figura 0.5

• **Una única instancia de la operación MAC con numerosas entradas** Ocurre al seleccionar el bucle de los canales de entrada o los de los kernels. Es la solución más sencilla pero provoca un gran critical path, por lo que tiene que ser usada con moderación.

• **Múltiples instancias de la operación MAC con una única entrada** Como consecuencia de desenrollar cualquiera de los otros tres bucles. Esta solución supone un mapeado más complejo, con diferentes opciones de reutilización de las entradas, pero no tiene prácticamente influencia en el critical path.

Por ser los dos bucles más sencillos, se selecciona los kernels como los más interiores. Con todo, es necesario poner en paralelo, al menos parcialmente, otro bucle más. Esta selección hace

\(^1\) Debido a la estructura squeeze-expand de la red una de las memorias puede reducir su tamaño considerablemente
que se descarte el de los canales de entrada y que se estudien las opciones de reutilización de las entradas de los otros tres bucles.

Figure 0.5: Diferentes arquitecturas resultado de desenrollar distintos bucles

(a) Una única MAC con múltiples entradas

(b) Varias MAC con entrada única

Reutilización de las entradas El bucle de los canales de salida se comporta de manera distinta a los de las filas y columnas de las imágenes. El primero requiere una mayor cantidad de datos, pues es necesario un kernel (9 elementos) para cada canal, así como 9 píxeles de entrada comunes a todos ellos. Mientras, los segundos necesitan un único kernel para todas las filas y columnas, pero distintas entradas. Sin embargo, existen píxeles compartidos entre diferentes iteraciones de los bucles, lo que permite un uso más eficaz de la información.

Finalmente se selecciona el bucle de las columnas para ser también paralelizado. Para hacer esto posible y fomentar una reutilización de las entradas, es necesario dividir en dos ese bucle e intercalar en medio el bucle de las filas, tal y como puede verse en el Algoritmo 1.

Mejoras

En su estado actual el núcleo es completamente funcional a pesar de que su funcionalidad está bastante restringida. Para completarlo y permitir la implementación completa de la red neuronal con la reconfiguración dinámica requerida es necesario realizar una serie de mejoras:

- **Parametrización dinámica: definición de la capa** Con la creación de la ROM para guardar los detalles de las capas de la red, cada ejecución del núcleo recibe información
sobre los límites de bucles, los detalles del kernel y la cantidad de bits fraccionales para la entrada, salida y pesos.

**Controlador de memoria: arrays bidimensionales** Permite el correcto acceso a las diferentes memorias, que en el caso de las entradas y las salidas, varían de tamaño entre capas. Para ello recibe información del módulo de control y del núcleo, conociendo en todo momento las posiciones que tienen que ser leídas o escritas. Pues para todas las memorias es necesario acceder a numerosos elementos de manera simultánea, estas son representadas con arrays en dos dimensiones que se dividirán en varias submemorias utilizando la directiva de compilador array_partition.

**Parametrización dinámica: kernel 3×3 stride 2 y 1×1** Para finalizar la reconfiguración dinámica es necesario permitir que el núcleo también ejecute otros dos tipos de kernels. Esto es posible haciendo mínimas modificaciones a la implementación para un kernel 3×3 con stride 1: seleccionando elementos alternos en la salida para el kernel del mismo tamaño con stride 2 y utilizando solo uno de los elementos de entrada para el kernel unidimensional.

**Implementación en punto fijo** Xilinx® Vivado HLS dispone de un tipo de data que implementa operaciones en punto fijo. Sin embargo, requiere que se definan el número de bits y la parte fraccional de manera estática, impidiendo la reconfiguración dinámica. Por esta razón, la implementación en punto fijo se debe hacer manualmente, utilizando los datos obtenidos de la definición de cada capa, bit shifts y operaciones con enteros. Los registros que guardan la información intermedia tienen una gran cantidad de bits para evitar overflows, mientras que las memorias mantienen la precisión de 8 bits y son utilizadas para almacenar los valores intermedios. Todo esto supone dos problemas para la precisión: el hacer shifts de números que pueden ser negativos (por no estar el comportamiento

---

**Algorithm 1 Loop order for implemented pipelined core**

```plaintext
for all ch_out do
    for all ch_in do
        for all 8-pixels-wide subcolums do
            for all rows do  # All inner loops are unrolled in hardware and this one pipelined
                for all pixels in 8-pixels-wide subcolum do
                    for all ker_h do
                        for all ker_w do
                            out += in * kernel
                        end for
                    end for
                end for
            end for
        end for
    end for
end for
```

sobre los límites de bucles, los detalles del kernel y la cantidad de bits fraccionales para la entrada, salida y pesos.

**Controlador de memoria: arrays bidimensionales** Permite el correcto acceso a las diferentes memorias, que en el caso de las entradas y las salidas, varían de tamaño entre capas. Para ello recibe información del módulo de control y del núcleo, conociendo en todo momento las posiciones que tienen que ser leídas o escritas. Pues para todas las memorias es necesario acceder a numerosos elementos de manera simultánea, estas son representadas con arrays en dos dimensiones que se dividirán en varias submemorias utilizando la directiva de compilador array_partition.

**Parametrización dinámica: kernel 3×3 stride 2 y 1×1** Para finalizar la reconfiguración dinámica es necesario permitir que el núcleo también ejecute otros dos tipos de kernels. Esto es posible haciendo mínimas modificaciones a la implementación para un kernel 3×3 con stride 1: seleccionando elementos alternos en la salida para el kernel del mismo tamaño con stride 2 y utilizando solo uno de los elementos de entrada para el kernel unidimensional.

**Implementación en punto fijo** Xilinx® Vivado HLS dispone de un tipo de data que implementa operaciones en punto fijo. Sin embargo, requiere que se definan el número de bits y la parte fraccional de manera estática, impidiendo la reconfiguración dinámica. Por esta razón, la implementación en punto fijo se debe hacer manualmente, utilizando los datos obtenidos de la definición de cada capa, bit shifts y operaciones con enteros. Los registros que guardan la información intermedia tienen una gran cantidad de bits para evitar overflows, mientras que las memorias mantienen la precisión de 8 bits y son utilizadas para almacenar los valores intermedios. Todo esto supone dos problemas para la precisión: el hacer shifts de números que pueden ser negativos (por no estar el comportamiento
completamente definido) y el reducir la cantidad de bits de valores intermedios. Estos dos comportamientos, junto con el hecho de que Ristretto simula punto fijo, deben ser las razones que marquen la diferencia entre la precisión de los dos sistemas.

• **Fire modules** Para finalizar la implementación de la arquitectura de la red, es necesario hacer posible que el módulo de control envíe al núcleo las memorias de manera alternativa conforme a la lógica de los fire modules. Debido a diversos problemas surgidos con el manejo de punteros en HLS, la decisión final es hacer iteraciones sobre los módulos en lugar de sobre las capas y conectar las memorias directamente al tipo (expand o squeeze) correspondiente.

**Análisis y resultados**

Tras más de 185 horas de simulación, el acelerador obtuvo una precisión del 80%, necesitando 26 millones de ciclos para el procesamiento de una imagen. Debido a que el critical path no es excesivamente grande, es posible trabajar a 100MHz, lo que permitiría procesar cerca de 4 imágenes por segundo. De tal forma se considera que los resultados han sido todo un éxito, al estar dentro de los márgenes que se plantearon como objetivos al inicio del proyecto.

A pesar de todo, el acelerador tiene una serie de limitaciones, como ser capaz de operar con kernels de 3 configuraciones distintas o imágenes con tamaños que sean múltiplos de 8 (debido a la disposición de la memoria), que no afectan al proyecto pero que provocan que no sea un acelerador absolutamente general para redes neuronales.

Para la obtención de los resultados anteriores se ha procedido a co-simular, utilizando Vivado HLS, toda la base de datos utilizada para la validación. Debido a diversas circunstancias relacionadas con las FPGAs disponibles, no ha sido posible realizar la implementación en hardware y una demostración final. En caso de que la implementación en hardware hubiese podido realizarse, para finalizar completamente el proyecto es necesario completar los siguientes pasos: conectar una cámara, utilizar un programa que recorte la imagen a 256x256 píxeles, les reste la media de los píxeles por canal (esto es debido a la red neuronal y podría incluso implementarse en hardware) y los envíe al módulo de la red en un orden establecido.
<table>
<thead>
<tr>
<th>Table of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Entrenamiento solo del clasificador</td>
</tr>
<tr>
<td>(b) Entrenamiento del clasificador y la primera capa</td>
</tr>
<tr>
<td>(c) Entrenamiento de todas las capas</td>
</tr>
</tbody>
</table>

Figure 0.3: Influencia del entrenamiento de distintas capa en la precisión
1.1 Motivation

Artificial Intelligence (AI) has played a big role in computer systems development during the last 40 years [2]. However, the interest towards it has only jumped to other fields in recent times, when the increasing availability of computing power and data has made it feasible to develop machine learning algorithms and Neural Networks in a high scale. That being said, we had the feeling that society still sees AI and Neural Networks as some kind of black magic, basically because most of the applications where it is useful (speech recognition, language translations, self-driving cars...) are not transparent to the users and data needed for the training is often obtained in opaques ways (at least until the new EU data protection regulation, GDPR). However, the improvements that Neural Networks and data analysis can make to our lives are still highly underdeveloped and there are several areas where the society can get big benefits.

One of these areas is the deaf’s people sign language. There are two different possibilities where technology can help the communication and understanding between deaf people and anybody else not able to understand sign language or even between deaf people that do not know the same sign language:

- The first of them is speech recognition. Nowadays most phones include this feature by default, which means that this technology is mature enough to be applied in situations where it can help the understanding of a speech by a deaf person.

- The second one is image recognition. This technology has been extensively research in recent times, specially since the publication of ImageNet dataset [15][5], but there is still a lot of potential improvements and applications that can be explored with current
technology. It could be applied to translate a conversation in sign language to any other way of communication: text, voice synthesis, etc.

Our project has been centered in image recognition. The long time objective would be to create a framework that, taking a video of someone talking in sign language, would be able to translate it to a chosen way of communication. The approach taken, by using Neural Networks, would allow a fairly easy portability between different languages, as there is no need of having a big technical knowledge to include a new language. The only thing that is needed is a big enough amount of data to retrain the network for the desired language.

To avoid the overheads of Internet communications and because the data being used by the framework is probably private and, in consequence, undesirable to be processed in a data-center, the framework should be able to work on a low resources system (e.g: a phone). This reason, together with the growing interest to get Neural Networks into embedded systems and the experience of both Lund’s and Madrid’s departments with FPGAs leaded to decide to use them as the support for our development. From the different possibilities that currently are available for hardware design, we decided to use what is known as High Level Synthesis (HLS). This approach allows us to design the hardware using a high level language like C, which increases productivity compared to RTL languages, and will be explained more in detail during section 2.2.

1.2 Scope and Structure

The idea of developing a full framework able to analyze and translate video in real time was completely fascinating. However, the complexity of the task and the lack of experience with Neural Networks in the department forced to cut down the scope in two ways, leaving the completion of the project to future thesis projects that could be backed on this one:

- Instead of trying to translate the whole language, the data set was restricted to just the alphabet. There are two different reasons to this restriction: the difficulty of getting enough data for a bigger data set and our lack of knowledge, advisory and experience with the Swedish sign language, which would have made it impossible to identify all the features needed for the translation. Anyway, the 26 letters of the alphabet should make it possible to prove the feasibility of the project and is a good starting point.

- Using only the alphabet also allowed to use a static analysis, based just on individual frames instead of video captures. The main reason for this decision is that although there are Neural Networks able to successfully analyze video, they have increased complexity and the analysis for their use in embedded systems is still not fully developed, which would add an extra hazard to the project.
Once the scope has been cut down for being able to fit a Bachelor/Master thesis, there is a need to divide the project in two subproblem that can be independently addressed, but that placed together will allow to obtain the results expected:

- In first place, there is a need to collect data, customize or design a network topology for the implementation and for training it so that the precision goals can be achieved. This neural network would also need to consume as little resources as possible to make it feasible to be implemented on a FPGA. For this part of the project, the system in use will consist on a GPU and a mature framework for neural network training, as they provide high precision with a minimum development time.

- In second place, when the neural network is ready for use on the GPU, a hardware accelerator for implementation on a FPGA will be designed. It will take the information obtained from the training and use it to improve the ratios cost/performance and power/performance compared to the usual GPU system. However, the functionality should not be compromised. For that reason the minimum acceptable performance is set to compute 1 frame per second, so close to real-time applications can be deployed out of it.
The following chapter will explain the theoretical background of some of the topics directly involved in the development of the project and will give a basic introduction to the tools that have been used for the development of the thesis and to all the resources and previous work in which the project is based.

2.1 Convolutional Neural Networks

2.1.1 Theoretical background

Convolutional Neural Networks are considered an Artificial Intelligence, classified as part of the Machine Learning discipline and also into the Deep Learning category.

Neural Networks  Neural networks are a family of computation architectures inspired by the brain. Each neuron receives input signals from several other neurons and produces output signals that also connect to several other neurons. This process is known as a synapse. The synapses control the information flow between the different neurons and together, all those millions of connections are which produce the thinking and behavior of the humans and all different species with a complex brain.

Artificial Neural Networks take advantage of the way neurons interact to build up systems where each of the building blocks (usually called neurons) get several inputs that are seized using what are called weights and produces an output that is sent to several other building blocks.
CHAPTER 2. STATE OF THE ART

**Functionality**  Neural networks are most of the time used to classify an input from a predefined set of possibilities. This is possible due to the existence of the weights associated to each of the neurons in the net. The specific group of weights that make a successful classification possible are determined during a previous step that is known as *training*. During this process, several inputs with a known output are issued into the network and the weights slowly updated until an optimal solution is reached.

**Organization**  To give all the neurons a proper structure able to analyze the input data, they can be organized in several different ways. For the interest of our project, we can conclude that they are placed in ordered layers, where they are only able to get input from the neurons in the previous layer and send output to the ones in the following layer.

A neural network topology is, in consequence, determined by the way the layers are organized and by the different operations that are executed in each of them, where the weights that have been previously obtained are often used.

**Convolutional Neural Network**  Convolutional Neural Networks (CNNs) are a special type of neural networks particularly suited for operation on 2D input data such as images. They are widely used for image classification, object recognition and scene labeling tasks.

When neural networks are employed for image-related tasks, their input usually consists of pixel data. Most of the times images without extensive resolution are used, however, even for a 256×256 RGB pixels, the resulting input consists of nearly 200 000 elements. Usually all the neurons in layer are connected to all the neurons in the following layer, leading to the, so called, fully-connected layer. However, taking that approach with images would lead to the need of billions of weights. Luckily, the locality of information in images allows a different, yet simpler approach, as the important information in images can be captured from local neighborhood relations. In order to decide whether there is a car in the center of an image it is not needed to consider the color of the top-right corner pixel. Strong contrasts indicates edges, aligned edges result in lines, combined lines can result in circles and contours, circles can outline a wheel and multiple nearby wheels can point to the presence of a car. This locality of information in images is exploited in convolutional neural networks by replacing the fully-connected layers with convolutional layers.

In the scope of this project, all the neural networks are built using convolutional layers (and non-linearity layers).

**Convolutional layer**  The input to each layer in a convolutional layer consists of a stack of 2D images (ch_in) of dimension h_in×w_in, the so-called input feature maps. Each layer produces a stack of 2D images (ch_out) of dimension h_out×w_out, called output feature maps. Each layer also contains a stack of ch_in×ch_out kernels (or 2-D filters) of size k×k (typically 1×1, 3×3, 5×5, 7×7 or 11×11), which contains the trained weights.
2.1. CONVOLUTIONAL NEURAL NETWORKS

Each input channel is convolved with a distinct 2-D filter from the stack of kernels; this stack of 2-D filters is often referred to as a single 3-D filter. The results of the convolution at each point are summed across all the channels. In addition, a bias can be added to the filtering results. This approach, requires \((k \times k) \times (ch_{in} \times ch_{out})\) weights, instead of requiring \((h_{in} \times w_{in} \times ch_{in}) \times (h_{out} \times w_{out} \times ch_{out})\) weights, which would be needed for a fully-connected layer. In addition, the independence from the input image dimensions also enables large images to be processed without an increase on the number of weights. For filters larger than \(1 \times 1\), border effects reduce the output dimensions. To avoid this effect, the input image is typically padded with zeros on each side. The filters can be applied with a stride \(s\), which reduces the output dimensions to \(w_{out} = w_{in} \div s\) and \(h_{out} = h_{in} \div s\).

**Non-linearity layer** Usually just after a layer that contains some weights (such as a convolutional one) is computed, some kind of non-linearity is applied to the output. The purpose of this non-linearity is to help the encoding of complex patterns, as if there were not applied, all the inference process would just be linear and unable to solve most problems, as mathematics and physics has taught us that real-world problems are often not linear.

Several different non-linear functions have been explored, like sigmoids or hyperbolic tangents. However, the most widely used one is know as ReLU and it consists on the identity for positive outputs and zero for negative ones. It is mostly used due to its simplicity and because most research has proved that more complex functions do not provide greater levels of accuracy.
2.1.2 Training and frameworks

During the development of a Neural Network for any application, the training is often one of the most complex processes and is often considered “more art than science” (Matthew Zeiler, winner ILSVRC 2013), with experience taking a big role in that process. Due to its complexity, and because it is extremely computational expensive, several amount of frameworks have been developed and even some engineering tools allow extensions for that purpose. Some of the most important ones are Caffe and Tensorflow.

For the development of this thesis, several of them have been explored as the lack of previous knowledge has influenced a progressive approach, slowly moving from simple ones with little amount of features to enterprise level frameworks. The ones used will be explained in detail in section 3.2

Training types  As it has already been stated, the process of training a neural network can be very complex, with several parameters influencing the training and interacting with each other in a complex way, so it is not always clear the influence that each of them have in the training. However, we can identify two clearly different kinds of trainings:

- **Full training** When enough data is available, all the weights in the net can be trained, so results are optimal for the application.

- **Transfer learning** However, usually there is not enough data available to train successfully all the weights in a neural network. In this cases the common approach is to use a network that has been previously trained for a previous application and *recycling* the weights of most of the layers, modifying only the last one, that will be customized for the desired application.

2.2 High Level Synthesis and FPGAs

Field Programmable Gate Arrays, or FPGAs are hardware devices widely used in the industry both for development and implementation of hardware circuits that are produced in low quantities. They consist in several reconfigurable logic resources like LUT, DSPs or BRAMs, which can be connected to each other and configured in several different ways. As those logical resources are most of the times counted on several thousand hundreds or even millions, the number of connections that can be done between them are countless and virtually allow the implementation of any electronic circuit.

For the last 30 years, the electronics circuits implemented in FPGAs have been designed using Hardware Description Languages (HDL) where the behavior of every piece of hardware is described in detail, consuming a considerable time to develop while having very detailed information about the hardware being design. In recent years, another approach for designing
hardware is gaining popularity, were the description is done in a high-level programming language like C and a there exists a tool which is responsible of generating a description in an RTL language, extracting the information from the high-level language. This new approach is known as High Level Synthesis (HLS).

HLS has not been popular in the past due to the poor results that it used to have and because of designers being afraid of losing control over the hardware being generated. However, nowadays most of the tools that exist in the market for this purpose have reached a state of maturity and stability that allow a sufficient understanding of the hardware being generated, while at the same it is usually more efficient than developers code, or at least enough to outcome the cost of a shorter development cycle compared to HDL. For all this reasons HLS has been the technology chosen for this thesis.

**How does it work** Probably the most asked question about HLS, specially when the reader has some knowledge about either some HDL language or the differences between hardware and software, is how is it possible to model hardware using a software oriented language. In other words, how is it possible to move from for and while loops, if statements, functions calls and even some object-oriented principles to state-machines, counters, memories, and signals? This is clearly not straight forward, but requires a complete rewrite of the code to allow hardware parallelization. However, that is usually not enough to generate a customized circuits. For that purpose, all the tools include a certain set of compiler pragmas that control loop unrolling, pipeline creation, memory organization, etc. that give the designer a better control over how the hardware will be implemented.

## 2.3 Current solutions: GPUs vs Embedded systems

In the field of neural networks, both processes of training and inference have been widely discussed and taken to practice using different approaches. One the most interesting aspects of neural networks is that the great independence between most of the computations allows a high level of parallelism, which goes in favor of the implementation on high-parallel platforms. For that reason the most appropriate platforms for implementation are:

**GPUs** Graphical Processing Units. With great difference, GPUs are the most popular platform for neural networks, both in data centers and smaller applications. GPUs put together between hundreds and thousands of parallel computation units with big quantity of distributed memory between them. They can offer speedups greater than 100 times compared to a sequential execution of all the operations involved in a neural network. This big advantage combined with the flexibility of operations that can be executed and with the interest that manufacturers have showed over AI, has influenced that most of the neural networks’ frameworks have, nowadays, support for GPUs.
CHAPTER 2. STATE OF THE ART

(a) Matrix multiplication in HLS

```c
#pragma ARRAY_RESHAPE -variable=m cyclic -factor 4 -dim 1
#pragma ARRAY_RESHAPE -variable=res cyclic -factor 4 -dim 1

// Iterate over the rows of the A matrix
Row: for(int i = 0; i < MAT_ROWS; i++) {
    // Iterate over the columns of the B matrix
    Product: for(int k = 0; k < MAT_ROWS; k++) {
        Col: for(int j = 0; j < MAT_COLS; j++) {
            res[i*MAT_COLS+j] = 0;
            // Do the inner product of a row of A and col of B
            res[i*MAT_COLS+j] = a[i*MAT_COLS+k] * b[k*MAT_COLS+j];
        }
    }
}
```

(b) Matrix multiplication in software

```c
// Iterate over the rows of the A matrix
for(int i = 0; i < MAT_ROWS; i++) {
    for(int j = 0; j < MAT_COLS; j++) {
        // Iterate over the columns of the B matrix
        sw_result[i][j] = 0;
        // Do the inner product of a row of A and col of B
        for(int k = 0; k < MAT_ROWS; k++) {
            sw_result[i][j] += in_mat_a[i][k] * in_mat_b[k][j];
        }
    }
}
```

Figure 2.4: Difference between code in HLS and software

However, these platforms suffer from two big drawbacks: high power consumption and high cost.

FPGAs  Field Programmable Gate Arrays. FPGAs can easily solve the power consumption and cost problems associated with GPUs while having even greater possibilities of parallelization. However, they offer a lot less flexibility than GPUs, having the need of redesigning at least some of the implementation when changing the application purpose. For this reason the interest in their use is growing over time, specially for being applied in data centers [17] or in highly regular applications, but none of the general purpose frameworks have developed support for them.

Although, as it has been mentioned, there exists some projects involving the implementation of neural networks and FPGAs, little amount of that development has been made publicly available, so most of the research is still on a theoretical side.
2.4 Zynqnet

The only implementation that has been made publicly available of neural networks in FPGA is a master thesis from ETH Zürich, where a neural network for an FPGA, named as Zynqnet, is designed [6]. Their work is based on a previously published topology known as SqueezeNet [8] that was originally thought to be applied on embedded systems. During Zynqnet development, SqueezeNet is modified to be made more “FPGA friendly”, and later a general accelerator is designed using HLS.

The Zynqnet topology consists in a first convolutional layer, 8 identical modules known as fire modules and a final classification layer (also convolutional) followed by an average pooling layer and an exponential layer used for probabilities calculation purposes. Each of the fire modules is a group of 3 convolutional layers in which the first layer’s (known as squeeze layer) output is connected to the other’s (known as expand layers) input. Afterwards, the expand layers’ output is concatenated and used as input for the next fire module. All the convolutional layers but the classifier are followed by a ReLU. It is also important to note that in their interest to make Zynqnet hardware friendly, the authors modified most of the net’s hyper parameters so they are either power or multiples of 2.

In this thesis, the main heritage and improvements made to their model can be summed up in the following points:

- **Topology** The original Zynqnet implementation is designed to classify 1000 categories from an important public dataset that is usually used as a standard for comparisons in image recognition, ImageNet [5][15]. In our implementation, the last layer of the net had to be modified to classify just the 26 symbols that we want to identify from the Swedish Sing Language Alphabet. In addition, the last layer, consisting in exponential functions and used for probability calculation has been removed, as we are only interested in finding the letter with the highest probability and that information can be extracted from the average pooling layer.

- **HW definition** The hardware accelerator that is designed during Zynqnet development is only partially implemented in hardware. They make use of a Xilinx Zynq board [6], so that they have an ARM processor that works tightly coupled with the hardware accelerator and taking care of some calculations and the control flow of the system. However, the accelerator presented in this thesis is fully designed in hardware. During runtime it is adapted to execute layers with different parameters without the intervention of any software system.

- **Fixed point** Zynqnet computations are done using 32 bit floating point implementation, as it is the standard data type for computations in neural networks. GPUs are very efficient handling 32bit floating points operations, so it comes at no cost for those systems. However, 32 bit operations would produces a big overhead in an FPGA. For that reason, fixed point
has been used. The decision about the bit width and fractional bits has been left for a tool designed for that purpose, Ristretto [7], with some manual fine-tuning being applied afterwards.

- **Data vs Mem** The reduction of the classification items together with the use of 8 bit fixed point weights allowed a significant reduction on the model size and the memory needed for the accelerator. This allowed several optimizations aiming to simplify the system (Zynqnet has a 51 steps pipeline as a core of the computation unit) and avoid access to external memory, being more important the speed of the computations than the amount of memory accesses.
This chapter covers a complete explanation on how the data collection and manipulation has been developed together with the process of training: from the first steps using Matlab to the final analysis of the model to determine the fixed-point bits needed for a minimal precision loss.

### 3.1 Data set

The first problem that is faced when dealing with a project involving Neural Networks is the availability of data to be able to deliver a successful training. For this project, a lot of people have volunteered their time to be taken photos or videos of their hands while they were doing the symbols of the Swedish Sign Language Alphabet. Thanks to all this people, the final dataset consist in more than 2600 photos that have been taken during the first two months of the project in several different sessions. For the data collection, a video was shown\(^1\) to the volunteers and photos/videos were taken of their hands. The correctness of the video was assured by checking it with an experienced person with Swedish Sign Language.

However, although 2500 may sound like a big number, it is a tiny amount compared to the more than 1.5 Million weights contained in Zynqnet. For that reason, the only reasonable training strategy that can be considered is transfer learning. Luckily, the Zynqnet team published the weights that achieved a 63% accuracy for the Imagenet Data Set.

#### 3.1.1 Data augmentation

Data augmentation is a common strategy\(^{[18][14]}\) used to increase the performance in terms of accuracy of a certain data set. It consists in expanding the original dataset by modifying the

---

\(^1\)ABC Swedish Sign Language: https://www.youtube.com/watch?v=0TGvDf9hoPk
images in different ways, such as increasing/decreasing saturation or rotating them, so that more and more variable images than the original ones can be used for the training, increasing the possibilities of a successful training.

Data augmentation is used to solve a very usual problem known as overfitting. It consists on the network being able to classify the images that are used during training with a high accuracy, while failing to correctly classify any other images. The most remarkable reasons for the overfitting problem to happen are related to both having a very little data set compared to the amount of weights being trained or to the data set being very regular, so that images that may differ from those common images will get wrongly classified. As increasing the data set is often a hard and time consuming task when dealing with images, data augmentation is a fairly popular method that is used during most training processes.

For the project, data augmentation was used from the very beginning. The transformations being applied have been horizontal flips of the images, zooms in and rotations (both clockwise and counterclockwise) of 10, 20 or 30 degrees. These modifications were done to increase the variability of positions and zooms where an image might have to be classified. After a certain
point in the project, also modifications in hue, saturation and brightness were applied in order to make the image classification less dependent on the camera and the lighting being used. The process of data augmentation is a highly repetitive task that has been automated using a simple bash script (see Appendix A) with ImageMagick[3] as the image manipulation tool and GNU parallel[1] to increase the performance.

3.1.2 Data organization

In order to evaluate the training’s success, images have been split in two datasets, one for training and one for testing/validation purposes. During the first trainings, the augmented dataset was randomly split up, taking 90% of the images for training and 10% for validation. However, that approach would not be able to reveal if some overfitting happened during the training. With the help of a more experienced PhD student that mistake was solved and a new validation dataset was used until the end of the project. After that point, and to make sure there is no correlation between validation and training images, all the augmented data was used for the training, while some more images were taken and used only for validation. Doing it in that way, any correlation between the training and the validation data can be avoided and similar results than in real-life operation should be achieved.

3.2 Training

This section explains, with further detail, the process of training the neural network, the acknowledgments gained during each step, improvements done and results obtained from them. By the end of the section, a reader should be able to understand the basics of a convolutional neural network training process; the hazards involved and the influence that different parameters have on the training. It should also be possible to have an idea of different frameworks available for the training, their scope, limitations and flexibility. It is also important to note that the data collection is developed concurrently to the training. Therefore, as the project moves forward, the data available is, slowly, increased.

Environment setup While during the first steps, getting acquainted with neural networks, performance was not a problem and training stages could be executed in reasonable time, a general purpose CPU was use. However, as the data kept increasing and training times started moving from hours to days, the necessity for a better system arose. For that purpose a Nvidia GeForce GTX 1080 Ti GPU was ordered and integrated in a 32GB RAM workstation, running CentOS. The GPU was chosen both due to performance reasons and to the possibility of being integrated in mostly any framework to speed up the training, as Nvidia has developed a specific library aimed to speed up neural networks[13] and drivers are also available for CentOS.
CHAPTER 3. TRAINING, DATA COLLECTION AND ENVIRONMENT SETUP

Methodology  As previously stated, the training has been developed using different frameworks that could fit the knowledge at a certain step while providing enough features to get a successful training. The first steps were done using Matlab and its Neural Network Toolbox[10]. This framework is simple enough to get started and provides extensive documentation about transfer learning. At a certain point, the training data requested some more complex training patterns which were limited by the tool. This forced a migration to a more complex framework. Out of all the ones available, Caffe, with support for the Nvidia 1080Ti and developed by UC Berkley[9] was chosen. This framework, developed in C++, provides both a command line and python interface which made it very simple and handy to use. In addition, Nvidia has developed a Graphical User Interface (GUI) that efficiently takes care of most of the repetitive tasks involved (such as preparing the databases for the training) but that only has support for a limited number of frameworks, which Caffe is part of. As a last step, Ristretto, a Caffe based tool that analyzes fixed-point feasibility and simulates fixed-point computations[7] was used to determine the bit width needed in each of the layers. Once the analysis was successfully done, some more training had to be done to adjust the fixed-point values for a better performance. In addition, due to the reduced amount of data available for training at the beginning, and to follow the transfer learning principles explained in Subsection 2.1.2, only the classifier’s weights will be trained unless stated differently.

3.2.1 Matlab - First steps

The training with Matlab started learning how to use the Neural Network Toolbox. For the trainings in Matlab, SqueezeNet[8] topology was used. The main reason behind that decision is that the performance difference between SqueezeNet and Zynqnet is negligible and Squeezenet has been ported to several different frameworks, which allowed to easily import the model using a function from the Neural Network Toolbox: Import Keras. For that use case, the SqueezeNet model in Keras (another neural networks framework[4]) was needed and easily found on the web[16] due to the great community using and publishing neural networks. Afterwards that step, with an important amount of data already collected (approx 1500 images), and spending some time fine-tuning the training parameters, the first trainings were delivered with encouraging results but showing a clear problem of overfitting, as in Figure 3.2

Training parameters  Although being quite simple, Matlab’s Neural Network Toolbox allow several training parameters to be chosen. Fine-tuning them required a lot of time and an extensive explanation and comparison between different values is out of the scope of this report. However, the most remarkable ones will be outlined:

- **Momentum** Influence that previous loss have on the actual loss update when updating the weights. 0.9 is the default and modifying it does not have great influence.
3.2 TRAINING

Figure 3.2: Training 1500 images; random split for validation; no augmentation

- **MiniBatchSize** Number of images that will be sent through the network before updating the weights. Usually, the bigger, the better, but is limited by the system memory. 128 is big enough to homogenize the training while fitting in most systems.

- **MaxEpochs** An epoch is considered to be completed when all images in the data set have been seen by the neural network once. After a certain amount of epochs, the accuracy is usually not increased but kept constant, so the maximum number of epochs is optimal when it is the minimum at which that behavior occurs. With our amount of images, it can be set between 8 and 12 epochs.

- **InitialLearnRate** Clearly one of the most important parameters, is used to obtain the value that the weights have to be modified based on the loss. Its value is, in some way, related with the Batch Size. After a lot of modifications, it is set around $1e^{-4}$, being increased in a factor between 10 and 20 times in the classifier.

- **LearnRateSchedule** Matlab only allows the learning rate to be kept constant or to be decreased by a factor every certain amount of epochs (piecewise). Different experiments have shown that nets of Alexnet, which SqueezeNet is based on, and also SqueezeNet achieve better accuracy with linear decrease in the learning rate[12][11], so a piecewise is the chosen schedule for being the most similar one.

- **LearnRateDropFactor** The pace at which the learning rate is decreased. It is strictly related to the learning rate value. Between 0.8 and 0.5 has been proven the best.
• **LearnRateDropPeriod** Amount of epochs after which the learning rate is decreased. It is co-related with the MiniBatchSize and the number of images. 2-3 epochs is the optimal for our 1500 images and 128 as batch size.

• **L2Regularization** Value added to certain images to avoid problems related with some of them being very similar to each other. The data augmentation certainly includes some images that might be quite similar, so a value of 0.1 is welcome.

• **Shuffle** When set to *every-epoch* the images being trained are placed in a different order every epoch, generating a more smooth training.

To try to fix the overfitting problem, data augmentation was done on the data set, increasing the amount of images from 1500 to 6500. The results were afterwards nearly perfect, with overfitting having completely disappeared and accuracy getting over a 90%, Figure 3.3.

![Training Progress (25-Feb-2018 23:56:23)](image)

**Figure 3.3:** Training 6500 images, random split for validation; augmentation done

However, at that point, the validation images were still chosen splitting one single dataset in 90% for training and 10% for validation. When doing the validation with images that were different to the ones seen by the net, another problem arose, as validation would only reach a 60% accuracy. This had to be with the validation dataset being wrongly chosen before. However, data augmentation had clearly improved the final accuracy on the proper dataset, Figure 3.4. This showed that the training was probably going on the right direction. However more images and better training strategies were probably needed.

At this point, the simplicity of Matlab’s toolbox started to become a problem, as no more experiments could be done with the learning rate due to the limited amount of schedules available. In addition, at some point the SqueezeNet model had to be replaced by Zynqnet and the lack of
3.2. TRAINING

(a) Validation with never seen images; no augmentation  
(b) Validation with never seen images; augmentation

Figure 3.4: Influence of data augmentation using a correct validation dataset

model’s availability made it very difficult to import the model to Matlab. However, at this point the project was mature enough to move to a more advanced framework.

3.2.2 Caffe + Digits - Professional framework + GUI

Out of all the frameworks available, Caffe was chosen because of several different reasons: being Free as in Freedom\(^2\); having support for the Nvidia GPU available; having a GUI developed by Nvidia and freely available; shipping with a python and command line interface; Zynqnet having been published in Caffe format; the existence of a project based on it able to analyze and simulate fixed-point calculations; and being developed by a university. All these reasons made Caffe the most suitable framework for our project.

The purpose of using Caffe was to be able to increase the current accuracy from 60% to over 80%, expecting it to be slightly decreased when using the fixed-point implementation on hardware, but still performing with more than the 80% accuracy that has been set as a goal for the project. In order to make all the following experiments comparable to each other and having enough variety of images, the validation dataset was updated to contain 163 images taken from different people and in different environments.

The first experiment consisted in training the network with the new and more complex validation set, while trying different learning rates and training strategies. The best result achieved still had overfitting but even with the harder validation, the results were slightly better than with Matlab, see Figure 3.5.

Once data augmentation has been exploited for increased performance, there is still between a 15% and 20% accuracy that needs to be found. As the overfitting is present and the training accuracy is very close to 100%. The conclusion that can be extracted is that, even training only the last layer, the net is able to memorize all the training images without extracting enough information to achieve a high accuracy in the validation.

\(^2\)Software project that can be used for any purpose and whose code can be accessed, modified and redistributed: https://www.gnu.org/philosophy/free-sw.html
In consequence, the only option available able to achieve that extra 15% accuracy is to increase the dataset. For that purpose, new sessions were scheduled getting up to 2160 images that became 10183 after augmentation. Several trainings were deployed again, with accuracy being slightly improved. However, at this point the training accuracy was not getting to 100%, which could mean that the ability to detect the images by training just one layer had reached its limit. One possible solution was to train, with a lower learning rate, some of the intermediate layers. Very good results were achieved training the input layer together with the classifier, getting up to 70% validation accuracy. See Figure 3.6

As the results were getting better but still far from the expectation and following the same reasoning as before the training dataset was updated again up to 2668 images and 13743 once data augmentation is performed. Results were clearly improved, specially when it came to training both the classifier and the input layers. However, as the training accuracy which that many images would not get over 90%, it was decided to train all the layers in the net (although having a higher learning rate in the classifier than in the other images to try staying closer to the concept of transfer learning) with extremely successful results, as the goal performance was achieved without any other modifications. See Figure 3.7
3.2. TRAINING

(a) Training 10000 images only in the classifier (b) Training 10000 images: classifier and input layers

Figure 3.6: Influence of training several layers in validation accuracy

3.2.3 Ristretto - Fixed-point analysis

Once the results were successful training the net in floating point and the GPU, the remaining task was to be able to reduce the amount of memory and resources needed for the inference by using fixed-point arithmetics. For that purpose Ristretto [7] is available. Ristretto is a tool based on Caffe for the condensation of neural networks. It has support for three different approximation schemes:

- **Reduced precision floating point** Each layer is defined by having a certain amount of bits for both exponent and mantissa. It is not of interest in our project as we are willing to avoid floating point calculations.

- **Power-of-two parameters** This solution is based on having weights which have only values that are power of two. This would avoid multiplications, as all operations could be done with bit shifts and pluses. This method is of a great interest in the project. However, there were several tries to make it work, with all completely failing, and could not be used.

- **Dynamic fixed point** Each layer inputs, outputs and weights are defined by their bit width and fractional bits, considering fixed-point arithmetics. It has been the one chosen for the project and its functionality will be explained in more detail.

Ristretto computations have three use cases independent of the approximation scheme used. However, the implementation is different for each of them and only the dynamic fixed point will be explained for being the only one of interest for the project:

- **Analysis** The main goal of Ristretto is to able to determine the minimum bits precision needed in each layer in order to still deliver a similar precision than with fixed point. It
CHAPTER 3. TRAINING, DATA COLLECTION AND ENVIRONMENT SETUP

gets an already trained net, an error margin and an approximation scheme and outputs a new model with bits quantized in each layer.

• **Inference** Ristretto simulates the inference of a neural network using fixed-point arithmetics. The weights and inputs, that are both provided in floating point, are converted to fixed-point (with a certain bit width and fractional bits set in the net definition), computations executed in floating point (GPUs do not have fixed-point computation units, only floating point) and the output again rounded to the closest fixed-point value and forwarded to the next layer.

• **Training** Ristretto authors recommend to always retrain the network after an analysis has been done, as they claim that it is able to recover at least part of the lost precision. During training, the forward passes are done in the same way as the inference but the weights updates are fully computed in floating point for a better precision.

In accordance to the previous explanation, the resulting net from the Caffe trainings was analyzed using Ristretto with a maximum error margin of a 3% accuracy. The results were outstanding, as the net would only require 8 bits both for the weights and for inputs-outputs of every layer. This means that the model as well as the memory for the activations can be reduced in a factor of 4. In addition, the computation power can be reduced in a power of 16, as only $8 \times 8$ bits need to be multiplied instead of $32 \times 32$. After the analysis, the net was retrained as suggested by Ristretto’s authors, with results in Figure 3.8.

However, after some work with the fixed-point implementation in hardware and a better analysis of the results given by Ristretto, there were two enhancements that needed to be done to the model.

On one hand, the fractional bits for the output of layers that needed to be concatenated were not equal in all the modules. This might not be a problem if intermediate values are stored in floating point (as Ristretto does), but it is just not feasible when implementing the net in fixed point. For that reason, several layers were modified so their output could be concatenated.

In second place, there were some big precision errors that will be explained more in detail in Subsection 4.4.4. The precision was mostly lost in the first layers. A closer analysis of those revealed that, for the inputs and outputs, not enough fractional bits were selected, having a greater precision loss than expected. That problem was easily fixed adding more precision (one bit) to the layers affected and leading to a model more hardware friendly.

The new model was again retrained reaching an accuracy of over 87% even higher than the original floating point model! The results can be compared in Figure 3.8 and the model comparison found in Appendix B.
3.3 Results and analysis

The main conclusion that can be extracted is that the performance achieved by the net, even using the fixed point implementation, is way above the lower limit set at the beginning of the project. In addition, there is an error margin of a 7% that can be lost during the hardware implementation while still fulfilling the specifications.

However, the training process has not been perfect overall and some overfitting can still be seen in the last experiments, leading to, approximately, a 15% difference between training and validation. This, together with training reaching easily a 100% accuracy suggests that greater performance could still be achieved. In order to do so, probably a more extensive dataset could be used, but due to the lack of time and (human) resources, it was not possible. In addition, more experiments could be done involving training several but not all layers but were not developed due to time restrictions.

It is also possible that a training able to further decrease the validation loss would lead to reducing even more the bits needed for the fixed-point implementation. Some experiments were done with Ristretto analysis and showed that most of the models in which only the classifier or one extra layer had been trained (which reach less accuracy both in training and validation), needed a higher bit width for the activations. In any case, the 4 times reduction in terms of memory and 16 in terms of power are considered to meet the project goals and would make it possible to implement the net in a modern large enough FPGA.

As a last point of the analysis, it is worth mentioning a difference in the validation performance’s behaviour noticed between training all the layers or just some. When training only the classifier and checking the predictions for different images, the identification was usually quite blurry. Even when images got rightly identified, other letters would still have reasonable
probabilities of being chosen, behaving in a similar way when predictions were wrong. Opposite to it, when training all the layers in the net, the accuracy was increased and the loss reduced but in this situations there were very little middle point. Rightly predicted images would most of the time get a prediction of 100%, behaving in a similar way when images were wrongly predicted. Following the mathematical models of prediction usually used for neural networks, this (extremely) biased behaviour might be considered a mistake and is most probably consequence of the net’s overfitting. However, in our *engineering* problem it can even be considered a great improvement because both the accuracy is increased and images clearly identified, making precision problems during hardware implementation a lot less critical.
Figure 3.7: Influence of training different layers in validation accuracy
This chapter covers all the information related to the design and implementation of the hardware accelerator that has been designed from scratch: methodology followed, an overview of the accelerator and an explanation of its modules and design decisions.

### 4.1 Design methodology

In any big code project that involves several different instances and with a growing complexity over time, it is very important to define both the order and scope of the steps that will be done and how the execution of those steps will be delivered. Both of them, combined, consist in the project's methodology.

#### 4.1.1 HLS methodology

As explained previously in Section 2.2, HLS is, for several reasons, the technology that will be used for the hardware implementation. The fact that there is less control over the hardware being designed makes it specially important to follow a certain methodology for the design. Although there are several HLS tools in the market, the one used for the design, Xilinx® Vivado HLS, documents and recommends their own methodology, which has been the one followed in the project. It consist in three steps that should be executed with different frequencies after code modifications:

- **Software simulation** Consists in testing the code’s execution using a regular software compiler and CPU. Input is provided by a test-bench that executes the function which wants to be tested and compares the output with the expected results. This test helps identifying
misbehaving code and is most of the time the first test being execute when big chunks of code are modified as it has short execution times.

- **Synthesis** Generates the HDL files needed to implement the code and pragmas written in HLS. It is the most critical step, as different ways of writing a piece of code might lead to different HDL definitions (not all of them having the expected behaviour and probably all of them having different performances) or might even not be synthesizable. Depending on the size of the project, it might need some time and is only executed when the software simulation for a piece of code has passed (as it is not efficient to synthesize a piece of code that does not behave as expected).

- **Co-simulation** It is the most important step, because it tests the correct functionality of the code that has been synthesized in the previous step. It makes use of the test-bench from the software simulation to generate the inputs in HDL form, executes a hardware simulation using the HDL files from the synthesis and compares the output with the one provided by the test-bench. If successful, it means that the hardware is behaving in the same way as the software behaves and the design can be considered approved. It takes the most amount of time. For example, every image from our dataset would need more than one hour to be simulated in hardware, compared to some seconds spent in software simulation and 1 to 2 minutes in the synthesis.

### 4.1.2 Development methodology

The accelerator being designed has several different ways of being approached. For this project it was decided to first design a pipelined and parallelized core able to execute a convolutional layer and its ReLU with fixed size input, output and a $3 \times 3$ kernel with stride 1; operating with integers. Afterwards, different improvements will be made to: execute layers with arbitrary parameters; increase the kernel possibilities to $3 \times 3$ with stride 2 and $1 \times 1$; use fixed point arithmetics; and implement the fire modules' structure of the net in hardware. This way, some modularity concepts can be used for the project, simplifying the management and avoiding mistakes. Due to the methodology followed, the analysis of the pipelined core is based on a convolutional layer of arbitrary input/output pixels/channels and a $3 \times 3$ kernel with stride and padding of 1, which keeps input/output pixel dimensions constant.

The main reason for the methodology chosen lays in executing the most complex task first, so mistakes are done in an early stage of the project and are easier to identify and manage. The most complex task in this project is parallelizing the convolution core and being able to execute the multiply-accumulate operations as fast as possible, maximizing the accelerator throughput. This problem can be easily identified reading the Zynqnet paper[6], where a 51 stages pipeline is developed. It suffers from a flushing issue which is considered to have an impact of reducing 6 times the potential throughput. In addition, if all the other improvements mentioned above were
developed before the parallelization is done, the amount of modifications needed to be done could easily overwhelm the developer and the possibility of writing synthesizable and efficient code would be extremely hard to achieve.

**Code organization** Having studied different projects developed using Xilinx® Vivado HLS and counting in the department with a PhD experienced in HLS, the project was chosen to be written in C++, using a C sequential style for algorithms and making use of namespaces for modularity purposes. The reasons behind is basically practical: to be able to use object oriented principles for the coding while avoiding the use of C++ objects that cannot be synthesized in HLS due to its lack of ability to handle the deep pointers needed for them.

### 4.2 Design overview

Before getting into details about the pipelined core implementation and all the other improvements, it is important to present how the module will work from the *top-level* perspective. A representation in pseudocode of this perspective can be found in Algorithm 2. The system consists of 3 different modules and a group of memories:

- **Pipelined core** It is the module responsible for executing the computations. It gets the layer, weight and input information from the flow control module, but it has no information about the memories being accessed, which is made transparent by the flow control module. At the same time, it is connected to the memory controller, that is able to identify the memories' positions needing to be accessed.

- **Convolutions flow control** The module responsible for the correct execution of the net topology. It can determine if the weights have to be updated or if an image has to be classified, performing the necessary steps for either case. It has access to all the memories and to each of the layers’ parameters. In case of image classification, it executes the layers sequentially (fetching the parameters from the corresponding memory) and sets the memories that will be used for input and output by the pipelined core. Once all the convolutions have been executed, it does the average pooling of the resulting channels and returns the one holding a maximum value, which corresponds to the predicted letter.

- **Memory controller** The module responsible for identifying which position has to be read/written from/to each memory all the time. It receives information from the control flow and the pipelined core modules in order to identify whether the layer has finished execution, a row’s computation has been execute, etc. During the pipeline core execution, it is requested the offsets of every memory in order for them to be accessed with the right pattern.

- **Memories** Used to store the weights, the layer’s parameters and input/output. The layer’s parameters are stored in a ROM, as it is not expected that the net’s model has to be modified
during runtime. However, it might happen that more data for training has been made available and in consequence the weights might have to be modified. For that reason, they are stored in RAMs and an interface’s options is made available for them to be written from the outside. For layer’s input/output only two memories are needed. The reason behind is that when one layer has finished execution, its output is forwarded by the control flow module to the next layer. At this point, the previous layer’s input is no longer needed and the memory storing it will be overwritten to store the current’s layer output. This method also works when it is needed to concatenate the expand layers of the fire modules. Their input is equal and the second one’s output can be stored contiguous to the first one without the need of a third memory. Due to the big reduction in the memory needed achieved with Ristretto, it is possible for all of it to be on-chip, reducing the overhead and simplifying the memory access. The amount of memory needed for each of the memories is detailed in Table 4.1.

### 4.3 Pipelined core

As the central point of the hardware accelerator, the pipelined core is the most complex and important module of the system. It will set the maximum throughput and minimum power consumption achievable. For its design it was necessary to study the convolutional layer’s computations insights to maximize data reuse, minimize memory accesses and overall, study the different possibilities of parallelization.

#### 4.3.1 Loops organization

A convolutional neural network consist in 6 nested loops with an operation of multiply and accumulate (MAC) in the innermost loop. In addition, the bias (unique for each output channel) has to be added once to the MAC operation. As all layers but the classifier have a ReLU after the convolution, it might also be interesting to integrate the comparison operation into the core to minimize its overhead. An example of a convolutional layer pseudocode can be found in Algorithm 3 and the six loops involved are:

- **Output channels** Each convolution layer generates a previously defined number of output channels, each of them with the same amount of pixels or output features.
RAM memories in red; ROM memories in green; execution modules in blue

Figure 4.1: Top level block diagram
Algorithm 2 Top-level function description

1: function INCREMENT(image I, bool ld_weights, kernels K, bias B)
2:     if ld_weights = true then
3:         bias ← B
4:         kernels ← K
5:         return 0
6:     end if
7:     mem_i ← I
8:     mem_o
9:     mem_ctr::config_layer(lay[0])
10:    pipelined_core(lay[0], mem_i, mem_o, kernels, bias) ▶ First convolutional layer
11:    for i in fire_modules do
12:        mem_ctr::config_layer(lay[i])
13:        pipelined_core(lay[i], mem_o, mem_i, kernels, bias) ▶ Squeeze layer
14:    end for
15:    mem_ctr::config_layer(lay[i + 1])
16:    pipelined_core(lay[i + 1], mem_i, mem_o, kernels, bias) ▶ First expand layer
17:    mem_ctr::concat_output(lay[i + 1])
18:    mem_ctr::config_layer(lay[i + 2])
19:    pipelined_core(lay[i + 2], mem_i, mem_o, kernels, bias) ▶ Second expand layer
20:    end for
21:    mem_ctr::config_layer(lay[last])
22:    pipelined_core(lay[last], mem_o, mem_i, kernels, bias) ▶ Classifier
23:    return average_pooling(mem_i)
24: end function

- **Pixel rows and columns** For the output generation, all the pixels are taken into account. Small differences exist between using the input or output pixels as loops’s references, but the concept that each input pixel will be used to generate (at least) one output pixel is common to all implementations.

- **Input channels** They are all used to generate each of the output channels.

- **2D kernel rows and columns** For each output pixel all the kernel elements have to be used adding another two loops to the hierarchy.

The order these six loops are placed defines the parallelization possibilities as well as the access pattern for each of the memories. For simplification and due to the independence between different 2D kernels, the two kernel loops are placed in the innermost positions. These two loops are also very easy to parallelize in hardware, as all the input elements for the MAC operations are independent to each other and have to be added to a single element (which is easily implemented using an adder tree). This would allow to execute several operations at the
Algorithm 3 Loop order example for convolutional layer + ReLU

\[
\text{for all ch\_out do}
\text{ \hspace{1cm} for all h\_out do}
\text{ \hspace{2cm} for all w\_out do}
\text{ \hspace{3cm} output[ch\_out][h\_out][w\_out] = bias[ch\_out]}
\text{ \hspace{3cm} for all ch\_in do}
\text{ \hspace{4cm} for all ker\_h do}
\text{ \hspace{5cm} for all ker\_w do}
\text{ \hspace{6cm} out[ch\_out][h\_out][w\_out] = in[ch\_in][h\_out + ker\_h][w\_out + ker\_w] *}
\text{ \hspace{6cm} kernel[ch\_out][ch\_in][ker\_h][ker\_w]}
\text{ \hspace{5cm} end for}
\text{ \hspace{4cm} end for}
\text{ \hspace{3cm} end for}
\text{ \hspace{1cm} if out[ch\_out][h\_out][w\_out] < 0 then}
\text{ \hspace{2cm} out[ch\_out][h\_out][w\_out] = 0}
\text{ \hspace{1cm} end if}
\text{ \hspace{1cm} end for}
\text{ \hspace{1cm} end for}
\text{ \hspace{1cm} end for}
\]

same time (9 for a 3×3 kernel), increasing the throughput. However, for a net with over 500 million MAC operations, this speedup is still not enough. In order to decide which loop has to be placed next to be parallelized, two things have to be taken into consideration: the unrolling scheme and data reuse maximization.

Loop unrolling scheme Unrolling different loops have a different influence on the parallel hardware being designed. However, an analysis of the loops and computations reveals that the six loops only lead to two different hardware architectures that are represented in Figure 4.2:

- **Single MAC instance with several inputs** This is the consequence of unrolling either the kernel or input channel loops. This architecture is generated because to compute a single output pixel, for every input channel all the elements of a kernel have to be multiplied by the corresponding input pixels. This solution is quite simple, as only one output element is needed. However, the main drawback is that abusing from it can create a long critical path which would have a big impact on the system’s performance.

- **Multiple instances of a single MAC operation** This is the consequence of unrolling the pixel or output channel loops, as for all those loops, each of the output pixels are independent to each other. In addition, some reuse of the input could be possible, but it varies depending on the loops. This solution requires a better mapping effort while having a fairly short critical path that can be beneficial for the design.
The two innermost loops have already been chosen (kernel ones). In consequence, analyzing the loop unrolling scheme discards the input channel loop as the other loop to be parallelized to avoid a longer critical path. This also means that the final architecture will be an overlapping of the ones presented above.

**Data reuse in loops** There are still three loops that can be parallelized to increase throughput and data reuse. Each of them offer a different possibility in terms of the data access pattern and how to reuse the input/output pixels and the weights:

- **Output channels** Each of this loop’s iterations makes use of a different 2D kernel and generates independent output pixels. However, the multiplications can be organized in such a way that the 9 input pixels are shared between all the MAC instances. In consequence, considering 8 output channels there are needed: 9 input pixels, $9 \times 8$ weights and 8 output pixels to a total of 89 elements. The combination of more output channels than the amount being parallelized with an outer non-unrolled loop would make unnecessary to load all the weights every time, reducing the amount of elements that need to be loaded every iteration to 17.

- **Pixel rows and columns** Each of this loop’s iterations makes use of the same 2D kernels and share some of the input pixels, while generating independent output. Considering an
image of 8×8 output pixels to be comparable the previous calculations, there are needed: 9 weights, 8 output pixels and 3×(8+2\(^1\)) input pixels to a total of 47 elements. The combination of this loop with an outer non-unrolled loop would also make unnecessary to load all the 30 input pixels every iteration, reducing the amount of elements to 18.

With all the information exposed above, the decision was made to parallelize and pipeline (not all the operations can be done on a single cycle) the pixel columns loop, as it needs a lot less local memory to execute the pipelined core and the data reuse between iterations is still significant, with little difference in the number of memory accesses needed compared to using the output channels. Once the inner loops are decided, the placement of the outermost loops only affects the data access pattern, which is not a problem in this project as all the data will be stored in the On-Chip memory.

Before continuing with the core’s development, the feasibility of the strategy chosen has to be checked. In first place, it is possible to unroll the columns loop because the minimum input/output image size between layers is 8×8 and all the other dimensions are multiples of 8. Splitting the column loop in two sub-loops, it is possible to create 8 parallel MAC instances and to execute several iterations when the images are greater than 8 pixels. In second place, to be able to load only 18 elements each iteration, the rows’ loop has to be placed between the two column sub-loops. With this configuration, the final access pattern for a 16×16 pixels image is as in Figure 4.3 and the final loops organization as in Algorithm 4.

### 4.3.2 Caching strategy

Once the loop order has been organized and to take advantage of data reuse, the information being reused has to be stored locally to the MAC instances to avoid the overhead of accessing On-Chip memory. For every type of memory, one cache is needed:

- **Kernel and bias caches** The most simple ones, as the channel loops are placed in the outermost position. The kernel weights only need to be loaded at the beginning and when there is a change in the channels, while the bias is only updated when moving to the next channel out.

---

\(^{1}\)Note that 3×3 kernels are considered, so there is a need of an extra pixel both on left and right sides.
Algorithm 4 Loop order for implemented pipelined core

```
for all ch_out do
    for all ch_in do
        for all 8-pixels-wide subcolumns do
            for all rows do  # All inner loops are unrolled in hardware and this one pipelined
                for all pixels in 8-pixels-wide subcolumn do
                    for all ker_h do
                        for all ker_w do
                            out += in * kernel
                        end for
                    end for
                end for
            end for
        end for
    end for
end for
```

- **Output cache** It is more complex than the previous caches due to the less regular access pattern. However, as the output pixels cannot be reused, every time there is a new iteration of the rows loop the values have to be updated. In order to maximize the performance, the bias is loaded to the cache instead of to the memory and the ReLU is also computed using this cache. To make it possible, the first time a certain output element is loaded into the cache, instead of loading a zero value, the bias is stored before starting the MAC operations. In addition, the last time the cache values are stored into the memory, it is checked if any value is negative and a zero stored in its position if it is the case.

- **Input cache** Out of all the caches, it is the most complex and problematic one, as some (but not all) of the data can be reused. The problem is solved generating a group of 3 registers that displace the information every iteration. This group of registers has a latency of 3 cycles to be fully loaded but then only 10 values need to be loaded every iteration, reusing 20 of the previous ones. A model of this cache can be found in Figure 4.4.

### 4.4 Pipelined core improvements

Although the pipelined core is now functional for a certain layer parameters, it very limited and has no flexibility in terms of kernel types and amount of input/output pixels/channels. In addition, all computations are still done only with integers. To create a fully dynamically reconfigurable accelerator there is still certain steps to deliver.
4.4. PIPELINED CORE IMPROVEMENTS

4.4.1 Parameter configuration: layer definition

This improvement is focused on making it possible to update the loop limits depending on the layer input and output channels and pixels. For this purpose, the first implementation of the control flow module was designed and the ROM storing the layer definition created. It is described in HLS using a C struct with different fields. By the end of the project, the layer definition will have fields to describe: if the layer has a $3 \times 3$ or $1 \times 1$ kernel; the stride value; amount of input and output channels and pixels; and the amount of fractional bits needed for input, output and the weights once the fixed point implementation is finished.

4.4.2 Memory controller: arrays merging

As different layers have different memory requirements, the access pattern has to be modified between layers. In addition, to be able to get the maximum throughput from the core, it has to be possible to access several elements at the same time.

To fulfill the first improvement, the memory controller was created. It keeps track of each of the memory offsets at all time, being updated when the caches are read or written, when entering a new layer or when some of the outermost loops finish an iteration.

To get the maximum throughput from the core it has to be possible to read kernel weights, input and output values in batches of 9, 10 and 8 respectively. It is also important that the mapping between the caches and the memories is coherent and that the HLS tool is able to correctly analyze that behaviour. To make it possible and considering that the input, output and kernels will always be multiples of 8, 8 and 9 respectively, the memories are subdivided in 8 or 9 smaller dual port memories, which is possible creating two dimension arrays (originally having 3 or 4 dimensions per layer) and using the HLS pragma `array_partition`. To finish this improvement and to be able to read the 10 input pixels needed per iteration from 8 memories, some rewrite of the input cache load algorithm was needed, exploiting the two available ports.

4.4.3 Parameter configuration: kernel $3 \times 3$ stride 2 and kernel $1 \times 1$

This step would complete the dynamic reconfiguration that the project requests. Making use of the same caches created for the $3 \times 3$ kernel with stride 1, the other options will be considered as simplifications of the same model, with some complexity added to the caches load/store algorithms and to the memory controller but keeping the MAC implementation unmodified.

For the $3 \times 3$ kernel with stride 2, the same computations than the base implementation are done. However, only alternate output elements will be loaded or stored. In consequence the performance for this kind of layers is reduced in half compared to the base implementation. Any other solution considering less performance loss would need greater caches (as with stride 2 there is less input data reuse) and a more complex mapping between them and the MAC instances, which was discarded for complexity reasons.
For the 1×1 kernel the approach taken was the opposite, as there is more memory available than needed. Only the middle row of the input cache needs to be used for the MAC (meaning that the computation power is not fully exploited), achieving the same performance than the base implementation. It is also important to note that for this kind of layers and with the purpose of keeping the kernels alignment, if the total amount of kernels in the layer is not a multiple of 9, the memory will be filled with zeros until it is. Not doing so could generate a huge problem with the mapping due to the misalignment of the next layers’s kernels.

### 4.4.4 Fixed point implementation

With the fixed point analysis of the network finished with Ristretto, the bit width and fractional bits for each of the layers have been defined. Xilinx® Vivado HLS provides a type definition for fixed point arithmetics: `ap_fixed<bit width,frac bits>`. However, it is not possible to modify the fractional bits during runtime, as the tool needs that value to be defined at compile time. To be able to cope with this reconfiguration, integers and bit shifted were used to implement fixed-point operations.

Let’s take, as an example, the first convolutional layer from the modified model presented in Appendix B. Input, output and weights they all have 8 as bit width, with 0, -2 and 7 as fractional bits, respectively. The input and weights multiplication will lead to a 9.7 fixed point number. At the same time, the output value needs -2 fractional bits, which means that all fractional bits and two of the integers bits are discarded. In consequence, once the MAC operation has been finished, the value from the cache will be shifted right 9 positions and then stored into the memory. When that value has to be loaded again into the cache, it is shifted left 9 positions again so the next MAC result is coherent.

This solution needs caches bit width to be quite big in order to avoid overflows, but at the same time, keeps the overall system memory unmodified. However, there are two different details that influence the precision achieved. The first one raises when shifting first right and then left small negative numbers. While shifting a -1 right should actually divide the number by two, getting zero as the result, what actually happens is that the 2’s Complement codification keeps that value as -1. Afterwards, when shifting it left, it is actually multiplied by two and resulting on a more negative number.

Figure 4.5: Fire modules memory identification
than the one originally stored. In addition, storing intermediate values back to the memories has the consequence of discarding several amount of bits for the next MAC operation, as well as risking the possibility of overflows that are controlled in caches. This has to be taken into consideration and should lead to slightly increasing the amount of bits needed for the memories.

Although the above issues could be mitigated increasing the control over the shifts operations and the overflows, it is quite complex and can lead to big increases in the pipeline depth. For that reason, they are left unfixed. In case the results do not reach the accuracy goal, some solution will be implemented, to the cost of a potentially big performance penalty. In addition, as all the operations are computed using integers, the issues presented above, together with Ristretto simulating fixed-point, are the only elements responsible for the precision difference that is expected between the GPU and the hardware accelerator.

### 4.4.5 Fire modules: double memory scheme

Once all the computational improvements have been successfully implemented, the control module has to be finished. Several different ways were tried to successfully send the pipelined core alternated memories on each iteration. It was also tried to let the memory controller take care of it. However, HLS is still very restricted in terms of pointers management and there were some other issues trying to make the tool understand how the memory swapping was done. As a result, the loop iterating over all the layers was manually unrolled to execute sequentially each of the fire modules's layers before proceeding to the next module. Then, the memories were hardwired to each call of the pipelined core module, as it can be seen in Figure 4.5. This also allows to generate several instances to the pipelined core that might be optimized by the HLS tool.

### 4.5 Results and analysis

After all the improvements presented, the accelerator is ready to classify images using the network weights. For that purpose Xilinx® Vivado HLS Co-simulation was used, as it mimics the hardware behaviour. All the images from the validation dataset that achieved 86% accuracy with Ristretto were evaluated and after more than 185 hours simulation it resulted in an overall 80% accuracy requesting 26 million cycles per image. As the critical path is quite small, a 100MHz clock can be used, being able to process approximately 4 frames per second.

This results can be considered a big success, as the precision is just on the lower limit that was set for the project and the performance is nearly 4 times bigger than that lower limit. In consequence, there is not any other modification that needs to be performed and the accelerator is ready to be deployed.
4.5.1 Synthesis results

Although the model that has been described in Section 4.2 is achievable and can be implemented, the reduced amount of resources that the pipelined core needs (50 DSPs, 8,800 Flip-flops and 10,000 LUTs) allow some modification/duplication of the instances to reduce the pipeline depth. Indeed, this situation is one of those were HLS can outperform human designers. As a default and because of the different memory inputs, Vivado generates two instances of the core with a 4 steps pipeline, requiring 26,596,261 cycles. To improve this design, several tries were made using the function_instantiate pragma, creating 4 different instances of the core. With a great amount of resources shared between them, only 15% more DSPs, 27% more flip-flops and 33% more LUTs were used compared to implementing the double mode core. This allows two out of the four pipelines to be reduced in 1 step. However, this had only a ridiculous 0.2% performance improvement, being finally discarded. A comparison with Zynqnet resources and performance can be found in Table 4.2.

4.5.2 Limitations and possible improvements

Although the results of the accelerator are a success, there are still several improvements that can be done and limitations to its flexibility:

- **Convolutional layers and kernel types** The most clear limitation is that the pipelined core is only able to compute convolutional layers with kernels of sizes $3 \times 3$ (with stride 1 and 2) and $1 \times 1$, with the necessary padding. In consequence, any net that considers layers different to these, cannot be accelerated. However, some extra work on the accelerator should make it possible to, extending the caches, add extra kernel types in a similar way to the $3 \times 3$ stride 2 and $1 \times 1$ kernels were added.

- **Image sizes** Another important limitation is that the accelerator relies on input and output images being all of sizes multiples of 8. Not having such regularity would completely misalign the memories, meaning that providing flexibility for image sizes would, most likely, require a completely rewrite of the accelerator.

<table>
<thead>
<tr>
<th></th>
<th>Dual Instance</th>
<th>Zynqnet</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRAM 18Kb</td>
<td>2,000</td>
<td>996</td>
</tr>
<tr>
<td>DSP Slices</td>
<td>100</td>
<td>739</td>
</tr>
<tr>
<td>FF</td>
<td>19,965</td>
<td>137,000</td>
</tr>
<tr>
<td>LUT</td>
<td>24,349</td>
<td>154,000</td>
</tr>
<tr>
<td>Performance (FPS)</td>
<td>4</td>
<td>0.5</td>
</tr>
<tr>
<td>Operation frequency (MHz)</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>External RAM</td>
<td>No</td>
<td>Yes (1GB)</td>
</tr>
</tbody>
</table>

Table 4.2: Comparison resources Zynqnet
• **Bit width** It is also important to note that the accelerator considers input, output and weights to have 8 bits width, as all the layers of the models presented in Appendix B only have parameters of 8 bits width. However, it should be possible to both use several parameters and macros to customize them at compile time (for nets with constant bit width) or to dynamically reconfigure the accelerator to perform operations with different bit widths, to the cost of using all the time memory for the maximum bit width.

• **Precision** Comparing the results of Ristretto to the ones from the accelerator, a bit more than a 6% accuracy has been lost. As it has been explained in Subsection 4.4.4, the precision loss due to approximations and shifts can be evaluated. Some fix for this issues could be implemented to increase precision, with an unknown performance penalty.

• **Net structure** Due to the problems experienced with HLS and pointers, the network structure had to be partially unrolled, as it can be seen in Figure 4.5. This would avoid executing a net with a different structure and is one of the main drawbacks of not having any software involved. However, anybody with a little experience with HLS could easily adapt this definition to the structure needed.

• **ReLU** This project has only considered convolutional layers that might have a ReLU after it. Using any other non-linearity is currently not implemented.
As it has been stated in Sections 3.3 and 4.5, the results for both training and the accelerator can be considered a success as the thesis goals have been fulfilled. However, this project can be considered part of a bigger goal: to be able to translate the Swedish Sign Language using a deep neural network. For several different reasons that end up in not having the necessary hardware or not being able to interact with it, a demo could not be done. In addition, some little steps (like subtracting the mean value from the input pixels using floating point) would still be missing even if the hardware was available. It is also important to note that the Swedish alphabet has another 3 special characters that have not been considered for the project as movement is involved and, as it was stated in Section 1.2 that is out of the project’s scope.

So to conclude the report, I can consider that this was a good first step in order to be able to transcript one sign language and it could be complemented by future thesis, getting closer to the goal of giving back to society what it has invested in our education.
The final budget and an explanation about the time frame for the development are provided in this chapter. However, through all the steps followed in this report, the methodology has been detailed. So the only missing point is the time frame. Thus, a Gant chart in order to clarify how the different steps of the project have overlapped each other. This chart can be seen in Figure 6.1.

Regarding the budget, an estimation considering the infrastructure, software, hardware and time spent can be found in Table 6.1. The human resources have been evaluated on 50€ per hour for the engineering specific work and 20€ per hour for the work in which no qualification is needed, such as training data collection. In addition, the computation resources have been evaluated on 2€ per hour of CPU and an extra 8€ per hour of GPU.

Figure 6.1: Gant chart: project’s schedule
<table>
<thead>
<tr>
<th>Details</th>
<th>Amount</th>
<th>Cost(€)</th>
<th>Total(€)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Workstation</strong></td>
<td>1</td>
<td>1,500.00</td>
<td>1,500.00</td>
</tr>
<tr>
<td>i7 core</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32GB RAM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Motherboard</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1TB HDD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Specific hardware</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU</td>
<td>1</td>
<td>800.00</td>
<td>800.00</td>
</tr>
<tr>
<td>Testing FPGA + license</td>
<td>1</td>
<td>800.00</td>
<td>800.00</td>
</tr>
<tr>
<td><strong>Human resources</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data collection and manipulation</td>
<td>50 hours</td>
<td>20.00/hour</td>
<td>1,000.00</td>
</tr>
<tr>
<td>Training</td>
<td>50 hours</td>
<td>50.00/hour</td>
<td>2,500.00</td>
</tr>
<tr>
<td>Accelerator design</td>
<td>140 hours</td>
<td>50.00/hour</td>
<td>7,000.00</td>
</tr>
<tr>
<td>Testing</td>
<td>30 hours</td>
<td>50.00/hour</td>
<td>1,500.00</td>
</tr>
<tr>
<td><strong>Computation resources</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulation</td>
<td>185 hours</td>
<td>2.00/hour</td>
<td>370.00</td>
</tr>
<tr>
<td>GPU Training</td>
<td>20 hours</td>
<td>10.00/hour</td>
<td>200.00</td>
</tr>
<tr>
<td><strong>Total projected</strong></td>
<td></td>
<td>15,670.00</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Project budget
LIST OF FIGURES

0.1 Capa convolucional ........................................ 4
0.2 Influencia del aumento de la base de datos en el entrenamiento .......................... 7
0.4 Comparación entre los resultados del entrenamiento de dos modelos en punto fijo 9
0.5 Diferentes arquitecturas resultado de desenrollar distintos bucles .................. 12
0.3 Influencia del entrenamiento de distintas capa en la precisión ................. 15
2.1 AI classification .............................................. 21
2.2 Convolutional layer ........................................... 23
2.3 Non-linearities ............................................... 23
2.4 Difference between code in HLS and software ............................................... 26
3.1 The Swedish Sign Language alphabet .......................................................... 30
3.2 Training 1500 images; random split for validation; no augmentation ............... 33
3.3 Training 6500 images, random split for validation; augmentation done ............ 34
3.4 Influence of data augmentation using a correct validation dataset .................... 35
3.5 Beginning of Caffe training ........................................ 36
3.6 Influence of training several layers in validation accuracy ............................ 37
3.8 Comparison between training results of two fixed-point net definitions .......... 39
3.7 Influence of training different layers in validation accuracy ......................... 41
4.1 Top level block diagram ........................................... 47
4.2 Different hardware architectures consequence of loop unrolling ..................... 50
4.3 Pixels access pattern ............................................ 51
4.4 Input cache registers ............................................ 52
4.5 Fire modules memory identification ......................................................... 54
6.1 Gant chart: project’s schedule ......................................................... 61

FIGURE

Page

63
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Accelerator memory requirements</td>
<td>46</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparison resources Zynqnet</td>
<td>56</td>
</tr>
<tr>
<td>6.1</td>
<td>Project budget</td>
<td>62</td>
</tr>
</tbody>
</table>
# Get images with random names and rename them following a pattern

```bash
placeNames() {
    declare -a images
    declare -a names

    SETS=8
    PIC_SET=1

    for image in $(find $DIR -type f | sort)
    do
        images+=($image )
    done

    # Photos organized by alphabets
    sets=0
    while [ $sets -lt $SETS ]
    do
        # for dir in $(ls $DIR | grep VID) ; do
        #    cd $dir
        # for letter in {A..Z}
        #    do
        #        cnnSubdir=$letter
        #        number=$(ls $CNNDIR/$cnnSubdir | cut -d_ -f2 | cut -d. -f1 | sort -nr | head -n 1)
        #        # Photos per letter
        #        counter=0
        #        number=$(($(number)+$(sets)*$PIC_SET)+30))
        #        while [ $counter -lt $PIC_SET ]
        #            #
        #        done
        #    done
        number=$((($number)+$(sets)*$PIC_SET)+30))
        number=$(($number+sets*PIC_SET+30))
        while [ $counter -lt $PIC_SET ]
            #
        done
    done
}
```

# Photos organized by alphabets
sets=0
while [ $sets -lt $SETS ]
do
    # for dir in $(ls $DIR | grep VID) ; do
    #    cd $dir
    # for letter in {A..Z}
    #    do
    #        cnnSubdir=$letter
    #        number=$(ls $CNNDIR/$cnnSubdir | cut -d_ -f2 | cut -d. -f1 | sort -nr | head -n 1)
    #        # Photos per letter
    #        counter=0
    #        number=$(($number+sets*$PIC_SET)+30))
    #        while [ $counter -lt $PIC_SET ]
    #            #
    #        done
    #    done
    number=$(($number+sets*$PIC_SET+30))
    number=$(($number+sets*$PIC_SET+30))
    while [ $counter -lt $PIC_SET ]
        #
    done
```
do
    number=$(($number+1))
    names+=($DIR/`dir/${letter}_${number}.${EXT}`)
    counter=$(($counter+1))
done
cd ../
sets=$(($sets+1))

for i in $(seq 0 $(($sets*$counter*26-1)) )
do
    mv $images[$i] $names[$i]
done

unset images
unset names

rotateClock(){
    find $DIR -type f | grep $EXT | parallel mogrify -rotate 90 {}
}

rotateCounterClock(){
    find $DIR -type f | grep $EXT | parallel mogrify -rotate -90 {}
}

resize(){
    find $DIR -type f | grep $EXT | parallel mogrify -resize ${IMG_SIZE}x${IMG_SIZE}+0+0 {}
}

moveCnnDir(){
    for ext in $ALLEXT
do
        for image in $(ls $DIR | grep $ext)
do
            cnnSubdir=$(echo $image | cut -d_ -f1 | cut -d. -f1)
            if [ -f $DIR/$image ]; then
                mv $DIR/$image $CNNDIR/$cnnSubdir/$image
            fi
        done
    done
}

68
# Make sure that all images have the same orientation
# even if they have been taken in different positions
orientate() {
    find $DIR -type f | grep $EXT | parallel mogrify -auto-orient {} 
}

# Get every image that already has a letter assigned and
# move it to the right folder with the right name
send_boc() {
    resize
    orientate
    for image in $( ls $DIR | grep $EXT )
    do
        cnnSubdir=$( echo $image | cut -d_ -f1 )
        number=$( ls $CNNDIR/$(cnnSubdir) | cut -d_ -f2 | cut -d. -f1 | sort -nr | head -n 1 )
        number=$((($number+1)) )
        cp $DIR/$image $CNNDIR/$(cnnSubdir)/$(cnnSubdir)_${number}.${EXT}
    done
}

crop() {
    find $DIR -type f | grep $EXT | parallel " mogrify -gravity center -crop 535x535+0+0 {}"
}

flip() {
    find $DIR -type f | grep $EXT | parallel " mogrify -flop {}"
}

rotate10() {
    ROTATE_10="-alpha set \( +clone -background none -rotate -10 \) -gravity center - compose Src -composite"
    ROTATE10="-alpha set \( +clone -background none -rotate +10 \) -gravity center - compose Src -composite"

    find $DIR -type f | grep $EXT | parallel \
        "convert {} $ROTATE_10 {}-10.$EXT ; \
         convert {} $ROTATE10 {}+10.$EXT ; \
         rm {}"
    find $DIR -type f | grep $EXT | parallel " mogrify -gravity center -crop 461x461+0+0 {}"
}
APPENDIX A. DATA MANIPULATION AND AUGMENTATION

rotate20 () {  
  ROTATE_20="−alpha set \( +clone −background none −rotate −20 \) −gravity center −-compose Src −composite"
  ROTATE20="−alpha set \( +clone −background none −rotate +20 \) −gravity center −compose Src −composite"

  find $DIR –type f | grep $EXT | par allel “ convert {} $ROTATE_20 {.} ; \
  convert {} $ROTATE20 {.} ; \
  rm {} ” 
}

rotate30 () {  
  ROTATE_30="−alpha set \( +clone −background none −rotate −30 \) −gravity center −-compose Src −composite"
  ROTATE30="−alpha set \( +clone −background none −rotate +30 \) −gravity center −compose Src −composite"

  find $DIR –type f | grep $EXT | par allel “ convert {} $ROTATE_30 {.} ; \
  convert {} $ROTATE30 {.} ; \
  rm {} ” 
}

add_noise () {  
  find $DIR –type f | grep $EXT | par allel “ mogrify +noise Gaussian {}; mogrify +noise Gaussian {} ” 
}

modulate () {  
  find $DIR –type f | grep $EXT | par allel “ convert −modulate 50 {}; −dark.EXT ; \
  convert −modulate 130 {}; −bright.EXT; \
  convert −modulate 100,50 {}; −sat.EXT; \
  convert −modulate 100,200 {}; +sat.EXT; \
  rm {} ” 
}

help () {  
  echo “Usage: 
  editCNNImages [-src-dir source_dir] [-dst-dir destination_dir] functions

  Functions are: 
  placeNames 
  rotateClock 
  rotateCounterClock 
  resize 
  moveCnnDir” 
}
orientate
send_boc
crop
flip
rotate10
rotate20
rotate30
add_noise
modulate

}

LOG="/home/pablo/Pictures/Training/tmp/flipped_images"
EXT=jpg
ALLEXT="jpg png"
CNNDIR="/home/pablo/data/BoC/trainingDataSet"
DIR="/home/pablo/Pictures/processing/tmp"
AUX_DIR="/home/pablo/Pictures/processing/tmp"
IMG_SIZE=256

while [ "#$" -gt 0 ]; do
  case "$1" in
    -src-dir)
      DIR="$2"
      echo "DIR set to $DIR"
      shift 2
      ;;
    -dst-dir)
      CNNDIR="$2"
      echo "CNNDIR set to $CNNDIR"
      shift 2
      ;;
    *)
      echo "Option not found, try one of -src-dir,-dst-dir"
      help
      ;;
  esac
done
Appendix B

Ristretto Models

Original model

```plaintext
input: "data"
input_shape {
  dim: 1
  dim: 3
  dim: 256
  dim: 256
}
layer {
  name: "conv1"
  type: "ConvolutionRistretto"
  bottom: "data"
  top: "conv1"
  convolution_param {
    num_output: 64
    pad: 1
    kernel_size: 3
    stride: 2
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: 0
    fl_layer_out: -3
    fl_params: 7
  }
}
layer {
  name: "relu_conv1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
layer {
  name: "fire2/squeeze3x3"
  type: "ConvolutionRistretto"
  bottom: "conv1"
}
```

Modified model

```plaintext
input: "data"
input_shape {
  dim: 1
  dim: 3
  dim: 256
  dim: 256
}
layer {
  name: "conv1"
  type: "ConvolutionRistretto"
  bottom: "data"
  top: "conv1"
  convolution_param {
    num_output: 64
    pad: 1
    kernel_size: 3
    stride: 2
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: 0
    fl_layer_out: -2
    fl_params: 7
  }
}
layer {
  name: "relu_conv1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
layer {
  name: "fire2/squeeze3x3"
  type: "ConvolutionRistretto"
  bottom: "conv1"
```
APPENDIX B. RISTRETTO MODELS

```plaintext
top: "fire2/squeeze3x3"
convolution_param {
    num_output: 16
    pad: 1
    kernel_size: 3
    stride: 2
}
quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -3
    fl_layer_out: -5
    fl_params: 6
}
layer {
    name: "fire2/relu_squeeze3x3"
    type: "ReLU"
    bottom: "fire2/squeeze3x3"
    top: "fire2/squeeze3x3"
}
layer {
    name: "fire2/expand1x1"
    type: "ConvolutionRistretto"
    bottom: "fire2/squeeze3x3"
    top: "fire2/expand1x1"
    convolution_param {
        num_output: 64
        kernel_size: 1
    }
    quantization_param {
        bw_layer_in: 8
        bw_layer_out: 8
        bw_params: 8
        fl_layer_in: -5
        fl_layer_out: -5
        fl_params: 6
    }
}
layer {
    name: "fire2/relu_expand1x1"
    type: "ReLU"
    bottom: "fire2/expand1x1"
    top: "fire2/expand1x1"
}
layer {
    name: "fire2/expand3x3"
    type: "ConvolutionRistretto"
    bottom: "fire2/squeeze3x3"
    top: "fire2/expand3x3"
    convolution_param {
        num_output: 64
        pad: 1
        kernel_size: 3
    }
    quantization_param {
        bw_layer_in: 8
        bw_layer_out: 8
        bw_params: 8
        fl_layer_in: -5
        fl_layer_out: -5
        fl_params: 7
    }
    top: "fire2/squeeze3x3"
    convolution_param {
        num_output: 16
        pad: 1
        kernel_size: 3
        stride: 2
    }
    quantization_param {
        bw_layer_in: 8
        bw_layer_out: 8
        bw_params: 8
        fl_layer_in: -2
        fl_layer_out: -4
        fl_params: 6
    }
}
layer {
    name: "fire2/relu_squeeze3x3"
    type: "ReLU"
    bottom: "fire2/squeeze3x3"
    top: "fire2/squeeze3x3"
}
layer {
    name: "fire2/expand1x1"
    type: "ConvolutionRistretto"
    bottom: "fire2/squeeze3x3"
    top: "fire2/expand1x1"
    convolution_param {
        num_output: 64
        kernel_size: 1
    }
    quantization_param {
        bw_layer_in: 8
        bw_layer_out: 8
        bw_params: 8
        fl_layer_in: -4
        fl_layer_out: -4
        fl_params: 6
    }
}
layer {
    name: "fire2/relu_expand1x1"
    type: "ReLU"
    bottom: "fire2/expand1x1"
    top: "fire2/expand1x1"
}
layer {
    name: "fire2/expand3x3"
    type: "ConvolutionRistretto"
    bottom: "fire2/squeeze3x3"
    top: "fire2/expand3x3"
    convolution_param {
        num_output: 64
        pad: 1
        kernel_size: 3
    }
    quantization_param {
        bw_layer_in: 8
        bw_layer_out: 8
        bw_params: 8
        fl_layer_in: -4
        fl_layer_out: -4
        fl_params: 7
    }
```

74
layer {
  name: "fire2/relu_expand3x3"
  type: "ReLU"
  bottom: "fire2/expand3x3"
  top: "fire2/expand3x3"
}
layer {
  name: "fire2/concat"
  type: "Concat"
  bottom: "fire2/expand1x1"
  bottom: "fire2/expand3x3"
  top: "fire2/concat"
}
layer {
  name: "fire3/squeeze1x1"
  type: "ConvolutionRistretto"
  bottom: "fire2/concat"
  top: "fire3/squeeze1x1"
  convolution_param {
    num_output: 16
    kernel_size: 1
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -5
    fl_layer_out: -6
    fl_params: 6
  }
}
layer {
  name: "fire3/relu_squeeze1x1"
  type: "ReLU"
  bottom: "fire3/squeeze1x1"
  top: "fire3/squeeze1x1"
}
layer {
  name: "fire3/expand1x1"
  type: "ConvolutionRistretto"
  bottom: "fire3/squeeze1x1"
  top: "fire3/expand1x1"
  convolution_param {
    num_output: 64
    kernel_size: 1
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -6
    fl_layer_out: -5
    fl_params: 7
  }
}
layer {
  name: "fire3/relu_expand1x1"
  type: "ReLU"
  bottom: "fire3/expand1x1"
  top: "fire3/expand1x1"
}
APPENDIX B. RISTRETTO MODELS

layer {
  name: "fire3/expand3x3"
  type: "ConvolutionRistretto"
  bottom: "fire3/squeeze1x1"
  top: "fire3/expand3x3"
  convolution_param {
    num_output: 64
    pad: 1
    kernel_size: 3
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -6
    fl_layer_out: -6
    fl_params: 7
  }
}
layer {
  name: "fire3/relu_expand3x3"
  type: "ReLU"
  bottom: "fire3/expand3x3"
  top: "fire3/expand3x3"
}
layer {
  name: "fire3/concat"
  type: "Concat"
  bottom: "fire3/expand1x1"
  bottom: "fire3/expand3x3"
  top: "fire3/concat"
}
layer {
  name: "fire4/squeeze3x3"
  type: "ConvolutionRistretto"
  bottom: "fire3/concat"
  top: "fire4/squeeze3x3"
  convolution_param {
    num_output: 32
    pad: 1
    kernel_size: 3
    stride: 2
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -6
    fl_layer_out: -7
    fl_params: 8
  }
}
layer {
  name: "fire4/relu_squeeze3x3"
  type: "ReLU"
  bottom: "fire4/squeeze3x3"
  top: "fire4/squeeze3x3"
}
layer {
  name: "fire4/expand1x1"
  type: "ConvolutionRistretto"
  bottom: "fire4/squeeze3x3"
  top: "fire4/expand1x1"
}
convolution_param {
    num_output: 128
    kernel_size: 1
}
quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -7
    fl_layer_out: -6
    fl_params: 7
}
layer {
    name: "fire4/relu_expand1x1"
    type: "ReLU"
    bottom: "fire4/expand1x1"
    top: "fire4/expand1x1"
}
layer {
    name: "fire4/expand3x3"
    type: "ConvolutionRistretto"
    bottom: "fire4/squeeze3x3"
    top: "fire4/expand3x3"
    convolution_param {
        num_output: 128
        pad: 1
        kernel_size: 3
    }
    quantization_param {
        bw_layer_in: 8
        bw_layer_out: 8
        bw_params: 8
        fl_layer_in: -7
        fl_layer_out: -7
        fl_params: 8
    }
}
layer {
    name: "fire4/relu_expand3x3"
    type: "ReLU"
    bottom: "fire4/expand3x3"
    top: "fire4/expand3x3"
}
layer {
    name: "fire4/concat"
    type: "Concat"
    bottom: "fire4/expand1x1"
    bottom: "fire4/expand3x3"
    top: "fire4/concat"
}
layer {
    name: "fire5/squeeze1x1"
    type: "ConvolutionRistretto"
    bottom: "fire4/concat"
    top: "fire5/squeeze1x1"
    convolution_param {
        num_output: 32
        kernel_size: 1
    }
    quantization_param {
        bw_layer_in: 8
        bw_layer_out: 8
    }
}
layer {
    name: "fire4/relu_expand1x1"
    type: "ReLU"
    bottom: "fire4/expand1x1"
    top: "fire4/expand1x1"
}
layer {
    name: "fire4/expand3x3"
    type: "ConvolutionRistretto"
    bottom: "fire4/squeeze3x3"
    top: "fire4/expand3x3"
    convolution_param {
        num_output: 128
        pad: 1
        kernel_size: 3
    }
    quantization_param {
        bw_layer_in: 8
        bw_layer_out: 8
        bw_params: 8
        fl_layer_in: -7
        fl_layer_out: -6
        fl_params: 7
    }
}
layer {
    name: "fire4/relu_expand3x3"
    type: "ReLU"
    bottom: "fire4/expand3x3"
    top: "fire4/expand3x3"
}
layer {
    name: "fire4/concat"
    type: "Concat"
    bottom: "fire4/expand1x1"
    bottom: "fire4/expand3x3"
    top: "fire4/concat"
}
layer {
    name: "fire5/squeeze1x1"
    type: "ConvolutionRistretto"
    bottom: "fire4/concat"
    top: "fire5/squeeze1x1"
    convolution_param {
        num_output: 32
        kernel_size: 1
    }
    quantization_param {
        bw_layer_in: 8
        bw_layer_out: 8
    }
}
APPENDIX B. RISTRETTO MODELS

layer {
  name: "fire5/relu_squeeze1x1"
  type: "ReLU"
  bottom: "fire5/squeeze1x1"
  top: "fire5/squeeze1x1"
}
layer {
  name: "fire5/expand1x1"
  type: "ConvolutionRistretto"
  bottom: "fire5/squeeze1x1"
  top: "fire5/expand1x1"
  convolution_param {
    num_output: 128
    kernel_size: 1
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -7
    fl_layer_out: -6
    fl_params: 7
  }
}
layer {
  name: "fire5/relu_expand1x1"
  type: "ReLU"
  bottom: "fire5/expand1x1"
  top: "fire5/expand1x1"
}
layer {
  name: "fire5/expand3x3"
  type: "ConvolutionRistretto"
  bottom: "fire5/squeeze1x1"
  top: "fire5/expand3x3"
  convolution_param {
    num_output: 128
    pad: 1
    kernel_size: 3
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -7
    fl_layer_out: -7
    fl_params: 7
  }
}
layer {
  name: "fire5/relu_expand3x3"
  type: "ReLU"
  bottom: "fire5/expand3x3"
  top: "fire5/expand3x3"
}
layer {
  name: "fire5/concat"
  bw_params: 8
  fl_layer_in: -6
  fl_layer_out: -7
  fl_params: 7
}
layer {
  name: "fire5/relu_squeeze1x1"
  type: "ReLU"
  bottom: "fire5/squeeze1x1"
  top: "fire5/squeeze1x1"
}
layer {
  name: "fire5/expand1x1"
  type: "ConvolutionRistretto"
  bottom: "fire5/squeeze1x1"
  top: "fire5/expand1x1"
  convolution_param {
    num_output: 128
    kernel_size: 1
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -7
    fl_layer_out: -6
    fl_params: 7
  }
}
layer {
  name: "fire5/relu_expand1x1"
  type: "ReLU"
  bottom: "fire5/expand1x1"
  top: "fire5/expand1x1"
}
layer {
  name: "fire5/expand3x3"
  type: "ConvolutionRistretto"
  bottom: "fire5/squeeze1x1"
  top: "fire5/expand3x3"
  convolution_param {
    num_output: 128
    pad: 1
    kernel_size: 3
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -7
    fl_layer_out: -6
    fl_params: 7
  }
}
layer {
  name: "fire5/relu_expand3x3"
  type: "ReLU"
  bottom: "fire5/expand3x3"
  top: "fire5/expand3x3"
}
layer {
  name: "fire5/concat"
layer {
  name: "fire5/concat"
  type: "Concat"
  bottom: "fire5/expand1x1"
  bottom: "fire5/expand3x3"
  top: "fire5/concat"
}

layer {
  name: "fire6/squeeze3x3"
  type: "ConvolutionRistretto"
  bottom: "fire5/concat"
  top: "fire6/squeeze3x3"
  convolution_param {
    num_output: 64
    pad: 1
    kernel_size: 3
    stride: 2
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -7
    fl_layer_out: -7
    fl_params: 8
  }
}

layer {
  name: "fire6/relu_squeeze3x3"
  type: "ReLU"
  bottom: "fire6/squeeze3x3"
}

layer {
  name: "fire6/expand1x1"
  type: "ConvolutionRistretto"
  bottom: "fire6/squeeze3x3"
  top: "fire6/expand1x1"
  convolution_param {
    num_output: 256
    kernel_size: 1
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -7
    fl_layer_out: -7
    fl_params: 7
  }
}

layer {
  name: "fire6/relu_expand1x1"
  type: "ReLU"
  bottom: "fire6/expand1x1"
}

layer {
  name: "fire6/expand3x3"
  type: "ConvolutionRistretto"
  bottom: "fire6/squeeze3x3"
  top: "fire6/expand3x3"
  convolution_param {
    num_output: 256
    pad: 1
  }
}

layer {
  name: "fire6/relu_expand3x3"
  type: "ReLU"
  bottom: "fire6/expand3x3"
}

layer {
  name: "fire6/expand1x1"
  type: "ConvolutionRistretto"
  bottom: "fire6/expand3x3"
  top: "fire6/expand1x1"
  convolution_param {
    num_output: 256
    kernel_size: 1
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -7
    fl_layer_out: -7
    fl_params: 7
  }
}

layer {
  name: "fire6/relu_expand1x1"
  type: "ReLU"
  bottom: "fire6/expand1x1"
}

layer {
  name: "fire6/expand3x3"
  type: "ConvolutionRistretto"
  bottom: "fire6/squeeze3x3"
  top: "fire6/expand3x3"
  convolution_param {
    num_output: 256
    pad: 1
  }
}
APPENDIX B. RISTRETTO MODELS

```
kernel_size: 3
}
quantization_param {
  bw_layer_in: 8
  bw_layer_out: 8
  bw_params: 8
  fl_layer_in: -7
  fl_layer_out: -7
  fl_params: 8
}
}
layers {
  name: "fire6/relu_expand3x3"
  type: "ReLU"
  bottom: "fire6/expand3x3"
  top: "fire6/expand3x3"
}
layers {
  name: "fire6/concat"
  type: "Concat"
  bottom: "fire6/expand1x1"
  bottom: "fire6/expand3x3"
  top: "fire6/concat"
}
layer {
  name: "fire7/squeeze1x1"
  type: "ConvolutionRistretto"
  bottom: "fire6/concat"
  top: "fire7/squeeze1x1"
  convolution_param {
    num_output: 64
    kernel_size: 1
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -6
    fl_layer_out: -7
    fl_params: 7
  }
}
layers {
  name: "fire7/relu_squeeze1x1"
  type: "ReLU"
  bottom: "fire7/squeeze1x1"
  top: "fire7/squeeze1x1"
}
layers {
  name: "fire7/expand1x1"
  type: "ConvolutionRistretto"
  bottom: "fire7/squeeze1x1"
  top: "fire7/expand1x1"
  convolution_param {
    num_output: 192
    kernel_size: 1
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -5
    fl_layer_out: -5
  }
}
layers {
  name: "fire6/relu_expand3x3"
  type: "ReLU"
  bottom: "fire6/expand3x3"
  top: "fire6/expand3x3"
}
layers {
  name: "fire6/concat"
  type: "Concat"
  bottom: "fire6/expand1x1"
  bottom: "fire6/expand3x3"
  top: "fire6/concat"
}
layer {
  name: "fire7/squeeze1x1"
  type: "ConvolutionRistretto"
  bottom: "fire6/concat"
  top: "fire7/squeeze1x1"
  convolution_param {
    num_output: 64
    kernel_size: 1
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -6
    fl_layer_out: -7
    fl_params: 7
  }
}
layers {
  name: "fire7/relu_squeeze1x1"
  type: "ReLU"
  bottom: "fire7/squeeze1x1"
  top: "fire7/squeeze1x1"
}
layers {
  name: "fire7/expand1x1"
  type: "ConvolutionRistretto"
  bottom: "fire7/squeeze1x1"
  top: "fire7/expand1x1"
  convolution_param {
    num_output: 192
    kernel_size: 1
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -5
    fl_layer_out: -5
  }
```
layer {
  name: "fire7/relu_expand1x1"
  type: "ReLU"
  bottom: "fire7/expand1x1"
  top: "fire7/expand1x1"
}

layer {
  name: "fire7/expand3x3"
  type: "ConvolutionRistretto"
  bottom: "fire7/squeeze1x1"
  top: "fire7/expand3x3"
  convolution_param {
    num_output: 192
    pad: 1
    kernel_size: 3
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -7
    fl_layer_out: -6
    fl_params: 8
  }
}

layer {
  name: "fire7/relu_expand3x3"
  type: "ReLU"
  bottom: "fire7/expand3x3"
  top: "fire7/expand3x3"
}

layer {
  name: "fire7/concat"
  type: "Concat"
  bottom: "fire7/expand1x1"
  bottom: "fire7/expand3x3"
  top: "fire7/concat"
}

layer {
  name: "fire8/squeeze3x3"
  type: "ConvolutionRistretto"
  bottom: "fire7/concat"
  top: "fire8/squeeze3x3"
  convolution_param {
    num_output: 112
    pad: 1
    kernel_size: 3
    stride: 2
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -6
    fl_layer_out: -6
    fl_params: 8
  }
}

layer {
  name: "fire8/relu_squeeze3x3"
  type: "ReLU"
  bottom: "fire8/squeeze3x3"
  top: "fire8/squeeze3x3"
}

layer {
  name: "fire7/relu_expand1x1"
  type: "ReLU"
  bottom: "fire7/expand1x1"
  top: "fire7/expand1x1"
}

layer {
  name: "fire7/expand3x3"
  type: "ConvolutionRistretto"
  bottom: "fire7/squeeze1x1"
  top: "fire7/expand3x3"
  convolution_param {
    num_output: 192
    pad: 1
    kernel_size: 3
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -7
    fl_layer_out: -5
    fl_params: 8
  }
}

layer {
  name: "fire7/relu_expand3x3"
  type: "ReLU"
  bottom: "fire7/expand3x3"
  top: "fire7/expand3x3"
}

layer {
  name: "fire7/concat"
  type: "Concat"
  bottom: "fire7/expand1x1"
  bottom: "fire7/expand3x3"
  top: "fire7/concat"
}

layer {
  name: "fire8/squeeze3x3"
  type: "ConvolutionRistretto"
  bottom: "fire7/concat"
  top: "fire8/squeeze3x3"
  convolution_param {
    num_output: 112
    pad: 1
    kernel_size: 3
    stride: 2
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -5
    fl_layer_out: -6
    fl_params: 8
  }
}

layer {
  name: "fire8/relu_squeeze3x3"
APPENDIX B. RISTRETTO MODELS

type: "ReLU"
bottom: "fire8/squeeze3x3"
top: "fire8/squeeze3x3"
}
layer {
  name: "fire8/expand1x1"
type: "ConvolutionRistretto"
bottom: "fire8/squeeze3x3"
top: "fire8/expand1x1"
  convolution_param {
    num_output: 256
    kernel_size: 1
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -6
    fl_layer_out: -4
    fl_params: 8
  }
}
layer {
  name: "fire8/relu_expand1x1"
type: "ReLU"
bottom: "fire8/expand1x1"
top: "fire8/expand1x1"
}
layer {
  name: "fire8/expand3x3"
type: "ConvolutionRistretto"
bottom: "fire8/squeeze3x3"
top: "fire8/expand3x3"
  convolution_param {
    num_output: 256
    pad: 1
    kernel_size: 3
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -6
    fl_layer_out: -5
    fl_params: 8
  }
}
layer {
  name: "fire8/relu_expand3x3"
type: "ReLU"
bottom: "fire8/expand3x3"
top: "fire8/expand3x3"
}
layer {
  name: "fire8/concat"
type: "Concat"
bottom: "fire8/expand1x1"
bottom: "fire8/expand3x3"
top: "fire8/concat"
}
layer {
  name: "fire9/squeeze1x1"
type: "ConvolutionRistretto"
bottom: "fire8/squeeze3x3"
top: "fire8/squeeze3x3"
}
layer {
  name: "fire8/expand1x1"
type: "ConvolutionRistretto"
bottom: "fire8/squeeze3x3"
top: "fire8/expand1x1"
  convolution_param {
    num_output: 256
    kernel_size: 1
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -6
    fl_layer_out: -4
    fl_params: 8
  }
}
layer {
  name: "fire8/relu_expand1x1"
type: "ReLU"
bottom: "fire8/expand1x1"
top: "fire8/expand1x1"
}
layer {
  name: "fire8/expand3x3"
type: "ConvolutionRistretto"
bottom: "fire8/squeeze3x3"
top: "fire8/expand3x3"
  convolution_param {
    num_output: 256
    pad: 1
    kernel_size: 3
  }
  quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -6
    fl_layer_out: -4
    fl_params: 8
  }
}
layer {
  name: "fire8/relu_expand3x3"
type: "ReLU"
bottom: "fire8/expand3x3"
top: "fire8/expand3x3"
}
layer {
  name: "fire8/concat"
type: "Concat"
bottom: "fire8/expand1x1"
bottom: "fire8/expand3x3"
top: "fire8/concat"
}
layer {
  name: "fire9/squeeze1x1"
type: "ConvolutionRistretto"
bottom: "fire8/concat"
top: "fire9/squeeze1x1"

```
convolution_param {
    num_output: 112
    kernel_size: 1
}
quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -5
    fl_layer_out: -4
    fl_params: 8
}
```

layer {
    name: "fire9/relu_squeeze1x1"
type: "ReLU"
bottom: "fire9/squeeze1x1"
top: "fire9/squeeze1x1"
}

layer {
    name: "fire9/expand1x1"
type: "ConvolutionRistretto"
bottom: "fire9/squeeze1x1"
top: "fire9/expand1x1"
convolution_param {
    num_output: 368
    kernel_size: 1
}
quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -4
    fl_layer_out: -2
    fl_params: 8
}

layer {
    name: "fire9/relu_expand1x1"
type: "ReLU"
bottom: "fire9/expand1x1"
top: "fire9/expand1x1"
}

layer {
    name: "fire9/expand3x3"
type: "ConvolutionRistretto"
bottom: "fire9/expand1x1"
top: "fire9/expand3x3"
convolution_param {
    num_output: 368
    pad: 1
    kernel_size: 3
}
quantization_param {
    bw_layer_in: 8
    bw_layer_out: 8
    bw_params: 8
    fl_layer_in: -4
    fl_layer_out: -3
    fl_params: 9
}


[4] F. Chollet et al., Keras.
https://keras.io, 2015.


The MathWorks, Natick, MA, USA.


