UNIVERSIDAD DE VALLADOLID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE MÁSTER

MÁSTER UNIVERSITARIO EN INVESTIGACIÓN

EN TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES

Investigación de la viabilidad operativa del reconstructor tomográfico para óptica adaptativa astronómica CARMEN a escalas E-ELT en GPUs

Autor:

D. Carlos González Gutiérrez

Tutores:

Dr. D. Mario Martínez Zarzuela

Dr. D. Francisco Javier Díaz Pernas

Dr. D. Francisco Javier de Cos Juez

Valladolid, 7 de Septiembre de 2016

TÍTULO:	**Investigación de la viabilidad operativa del reconstructor tomográfico para óptica adaptativa astronómica CARMEN a escalas E-ELT en GPUs**
AUTOR:	**D. Carlos González Gutiérrez**
TUTOR:	**Dr. D. Mario Martínez Zarzuela** **Dr. D. Francisco Javier Díaz Pernas** **Dr. D. Francisco Javier de Cos Juez**
DEPARTAMENTO:	TEORÍA DE LA SEÑAL Y COMUNICACIONES E INGENIERÍA TELEMÁTICA

TRIBUNAL

PRESIDENTE:	**Dr. D. Miguel López Coronado**
VOCAL:	**Dr. D. José Fernando Díez Higuera**
SECRETARIO	**Dr. D. Carlos Gómez Peña**
FECHA:	**12 de Septiembre de 2016**
CALIFICACIÓN:	

Resumen de TFM

Muchos de los sistemas de óptica adaptativa de nueva generación, empleados en los telescopios extremadamente grandes, requieren de técnicas tomográficas para corregir la turbulencia atmosférica sobre un campo de visión amplio. Se van a presentar diferentes implementaciones del reconstructor tomográfico basado en aprendizaje maquina “CARMEN”. En primer lugar, se introducirán conceptos básicos de óptica adaptativa, además de una pequeña explicación sobre sistemas de control utilizados en telescopios reales. Además, se detallará el funcionamiento del reconstructor, junto a tres *frameworks* de redes neuronales, y un código desarrollado directamente en CUDA. Se estudiará la influencia de los cambios en el tamaño del reconstructor, a la hora de entrenar y ejecutar la red. El código nativo en CUDA ha resultado ser la mejor opción en todos los casos, aunque algunos de los *frameworks* ofrecen un buen rendimiento bajo determinadas circunstancias.

Palabras clave

Óptica Adaptativa, Redes Neuronales, GPUs.

Abstract

Many of the next generation of adaptive optics systems on large and extremely large telescopes require tomographic techniques in order to correct for atmospheric turbulence over a large field of view. Different implementations of the tomographic reconstructor based on machine learning “CARMEN” are presented. Basic concepts of adaptive optics will be introduced first, with a short explanation about three different control systems used on real telescopes. The operation of the reconstructor, along with the three neural network frameworks used, and the developed CUDA code will be detailed. Changes to the size of the reconstructor influence the training and execution time of the neural network. The native CUDA code turns out to be the best choice for all the systems, although some of the frameworks offers good performance under certain circumstances.

Keywords

Adaptive Optics, Neural Networks, GPUs

Agradecimientos

Quiero agradecer a mi familia y amigos todo su apoyo durante todos estos años, por estar ahí en los momentos complicados, y por haberme motivado para seguir incluso cuando las cosas estaban difíciles.

No me quiero olvidar de dar las gracias a todos los profesores que me han ayudado durante este máster, en especial mis tutores, por todo su apoyo para sacar este trabajo adelante, y por ofrecerme la posibilidad de seguir dedicándome a la investigación en un futuro.

A todos vosotros, muchas gracias.

Índice

Lista de Figuras

Lista de Tablas

Capítulo 1: Introducción

La observación de las estrellas que aparecen en el firmamento ha fascinado al ser humano desde hace miles de años. La aparición de los primeros telescopios a principios del siglo XVII, supuso una revolución dentro de las observaciones estelares, ya que los astrónomos pudieron encontrar más detalles que no eran observables a simple vista. Con el paso de los siglos, los telescopios siguieron evolucionando, mejorando la calidad de los materiales con los que se construían, desarrollando nuevos sistemas de observación, o aumentando su tamaño para capturar la mayor cantidad de luz posible.

En la actualidad nos encontramos con telescopios que poseen varios metros de diámetro, y que nos permiten observar objetos que se encuentran a millones de años luz de nuestro planeta. Sin embargo, las observaciones realizadas desde telescopios terrestres plantean un problema importante. La luz, que ha viajado durante billones de kilómetros sin sufrir interferencias ni deformación alguna, tiene que atravesar la atmósfera terrestre hasta llegar a la lente del telescopio. Durante estos últimos kilómetros de su recorrido, las nubes y las diferentes turbulencias que posee nuestra atmósfera, distorsionan la trayectoria de la luz, haciendo que los fotones que recibimos en el telescopio lleguen desordenados.

Este problema supone un gran reto a la hora de observar objetos muy lejanos o poco luminosos, ya que pequeñas distorsiones pueden provocar que el objeto que se esté observando se vea borroso o incluso no llegue a distinguirse. Para solucionar este inconveniente, existen dos alternativas. La primera de ellas consiste en enviar el telescopio fuera de la atmósfera terrestre. Con esto conseguiremos que la imagen recibida no tenga distorsión alguna provocada por las turbulencias atmosféricas. Aunque esta solución sea la ideal, tiene un problema muy importante, y es el elevadísimo coste que tiene crear y mantener un telescopio de este tipo orbitando alrededor de la tierra.

Dado que no es económicamente viable mandar al espacio tantos telescopios como nos gustaría, se optó por buscar una segunda solución que permitiese mejorar la imagen recibida en los telescopios terrestres. La óptica adaptativa [1], permite corregir las distorsiones producidas por la atmósfera mediante el uso de un espejo deformable, que se adaptará al frente de onda formado por la luz recibida. Sin embargo, esta corrección no es trivial, por lo que es necesario el desarrollo de algoritmos que permitan calcular la deformación introducida por la atmósfera, de tal forma que pueda ser corregida mediante el espejo deformable.

A lo largo de los años, se han propuesto diferentes soluciones para lidiar con este problema. Una de las más destacadas y comúnmente utilizada, es emplear un algoritmo basado en reducir el error mediante mínimos cuadrados [2][3]. Otro algoritmo desarrollado para lidiar con este problema, es el conocido como *Learn and Apply* (L&A), desarrollado por el Observatorio Lesia de París [4]. A ellos, se ha añadido recientemente CARMEN (*Complex Atmospheric Reconstructor based on Machine lEaRning*), desarrollado en la Universidad de Oviedo, cuyo funcionamiento ya ha sido testado en el *William Herschel Telescope* (WHT) del Observatorio del Roque de los Muchachos [5].

1.1. Motivación

El aumento del tamaño de los telescopios, y los sistemas de óptica adaptativa, ha sido considerable a lo largo de los últimos años. En en unos años [6], se ha pasado de utilizar unas pocas decenas de valores de entrada y procesado de la información, hasta multiplicar dicha cantidad casi por mil veces. Incluso, si se mira hacia el futuro, se espera que dicho número siga aumentando, especialmente si tenemos en cuenta las cifras que se manejan con el *European Extremely Large Telescope* (E-ELT) [7]. Este aumento de la cantidad de información, supone un enorme reto computacional, tanto a la hora de gestionar y manejar el telescopio, como en el momento de aplicar los algoritmos de óptica adaptativa deseados.

Una solución frecuentemente utilizada, es el empleo de *Graphics Proccessing Units* (GPU), para paralelizar el proceso de cálculo y poder acelerar de forma notable la obtención de resultados. Existen ya algunas aproximaciones con vistas a su uso en el E-ELT, como la mostrada en [8], donde se hace una comparativa sobre el uso de diferentes elementos *hardware* para la aceleración de los procesos, incluyendo el uso de GPUs.

Si tenemos en cuenta los algoritmos de óptica adaptativa, también se han encontrado algunos artículos que nos indican cómo se han empezado a utilizar tarjetas gráficas para acelerar estos procesos. Por un lado, tenemos [9], donde se utilizan tanto

GPUs como *Field Programmable Gate Arrays* (FPGA) para acelerar la obtención del frente de onda reconstruido. Se ha encontrado también como el algoritmo de *Learn and Apply* está siendo adaptado para su funcionamiento en GPU [10], que ha permitido acelerar enormemente las diferentes operaciones con matrices que se realizan.

En el caso de CARMEN, tenemos un reconstructor basado en redes neuronales. Estaba originalmente desarrollado en R, utilizando el paquete AMORE. Sin embargo, las redes neuronales son algoritmos con un altísimo grado de paralelismo, que pueden ser ejecutadas en GPU consiguiendo acelerar de forma notable los tiempos de cálculo. Hay que tener en cuenta, que este tipo de aprendizaje máquina, requiere dos fases. Por un lado, es necesario realizar una fase de entrenamiento, para que la red aprenda los valores que luego va a emplear. Una vez que se ha realizado dicho entrenamiento, ya se podrá ejecutar la red para obtener las salidas adecuadas. Este proceso hace que sea necesario conseguir acelerar tanto el entrenamiento, como la ejecución, ya que ambos procesos cumplen con los requisitos adecuados para ser acelerados mediante GPUs.

1.2. Objetivos

El principal objetivo del presente trabajo, es implementar el reconstructor CARMEN, para que funcione en GPUs, de tal forma que se consiga acelerar de forma notable, sus tiempos de entrenamiento y de ejecución. Para ello, se han marcado una serie de objetivos parciales, que será necesario ir consiguiendo antes de llegar al resultado final:

- Estudiar las posibles implementaciones de CARMEN en GPUs, ya sea mediante la utilización de *frameworks* de redes neuronales, que faciliten el trabajo de desarrollo, o utilizando directamente un lenguaje de programación nativa en GPU como CUDA.
- Desarrollar los códigos necesarios para poder entrenar y ejecutar CARMEN en las opciones elegidas previamente.
- Diseñar un experimento, que permita comparar bajo condiciones controladas, los resultados obtenidos por las diferentes implementaciones. En este caso, el objetivo de la comparación serán las velocidades de ejecución y entrenamiento que proporcionan los distintos códigos desarrollados.

Se espera que si se consiguen todos los objetivos parciales propuestos, pueda alcanzarse con éxito el objetivo principal. Sin embargo, se han propuesto también una serie de objetivos secundarios, que pueden tener una gran importancia de cara al futuro

desarrollo del reconstructor, y a la continuación del presente trabajo como una tesis doctoral.

- Adquirir una base de conocimientos sobre óptica adaptativa y redes neuronales que no se poseían previamente. Aunque dichos conocimientos no son de una importancia crucial para el presente trabajo, se consideran muy importantes de cara a desarrollar futuras mejoras en el reconstructor, o a su utilización en nuevas aplicaciones.
- Utilizar los diferentes códigos creados, como base para que otras personas del grupo de investigación, puedan iniciarse en el uso y el manejo de redes neuronales.
- Publicar el contenido de este trabajo de investigación, como parte de un artículo en una conferencia o una revista de impacto.

Capítulo 2: Conocimientos Previos

Para realizar el presente Trabajo Final de Máster, ha sido necesario adquirir una serie de conocimientos previos, que ayudasen a entender mejor el problema presentado. A lo largo del presente capítulo, se procederá a explicar algunos de ellos, poniendo especial énfasis en aquellos conceptos que son relevantes para plantear el experimento.

El primer concepto importante, es una explicación sobre el funcionamiento de los telescopios de gran tamaño, especialmente en sus sistemas de óptica adaptativa, su funcionamiento, y cómo afecta esto a la creación y el diseño de los reconstructores. A continuación, se hará una pequeña explicación práctica sobre las redes neuronales, poniéndolas un poco en su contexto histórico. Además, se detallará cómo entrena una red neuronal, utilizando un pequeño ejemplo práctico, que nos ayudará a entender mejor sus posibilidades de paralelización y su funcionamiento. Tras ello, entraremos a detallar cómo ambos conceptos se combinan para crear el reconstructor CARMEN, explicando su funcionamiento. Para cerrar el capítulo, se introducirá de forma breve la computación en tarjetas gráficas, y hablaremos sobre diferentes *frameworks* que las utilizan y serán objeto de comparación en posteriores capítulos.

2.1. Sistemas de Óptica Adaptativa

Existen dos clases fundamentales de telescopios desarrollados a lo largo de décadas, denominados telescopios reflectantes y telescopios refractantes. Los telescopios refractantes aparecieron inicialmente en 1550, gracias a Leonard Digges, pero las primeras descripciones reales datan de años posteriores, de inicios del siglo XVII. Este tipo de telescopio, capta los rayos de luz paralelos procedentes de un objeto a gran distancia, y haciendo uso de lentes convergentes logra que los rayos converjan

en un punto del plano focal. Son comunes en telescopios solares, pero su uso trae problemas relacionados con las lentes. El primero se debe a la elaboración de las propias lentes, ya que es difícil fabricarlas útiles y ligeras, especialmente cuando son de gran tamaño. Además, existe la posibilidad de que aparezcan pequeñas burbujas de aire alojadas en algunas de las cavidades de la lente principal. Por último, para ciertas longitudes de onda, los cristales de la lente son opacos, perdiendo sensibilidad en partes del espectro de luz. En la Figura 1, podemos observar el funcionamiento de un telescopio de tipo refractor, donde una lente concentra la luz recibida sobre el plano focal.

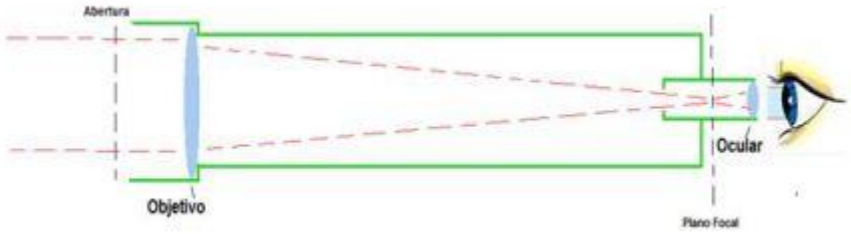


Figura 1: Telescopio Refractor

La mayoría de los problemas que conlleva el uso de telescopios refractantes, se solucionan haciendo uso de telescopios reflectantes, ya que este utiliza espejos en lugar de lentes. Algunas de sus ventajas son que solo es necesario garantizar la perfección de la superficie del espejo, en lugar del total de la lente, además de que no existe aberración cromática en un espejo, a diferencia de lo que ocurre en las lentes. Otra de las ventajas importantes, es que pueden ser completamente apoyados en una superficie, garantizando su estabilidad y evitando posibles deformaciones por culpa de la gravedad, a diferencia de las lentes, que únicamente pueden ser sujetadas por los extremos. Dentro de los telescopios de tipo de reflectante, se tienen diferentes variantes, siendo los más destacados los Newtonianos y los de tipo Cassegrain. En la Figura 2, podemos observar el funcionamiento de un telescopio catadióptrico o Cassegrain, en el que la luz es reflejada a través de dos espejos antes de ser observada. El funcionamiento de este telescopio, es muy similar al utilizado en todos los observatorios profesionales, y llevado con pequeñas modificaciones a los telescopios de gran tamaño o ELT.

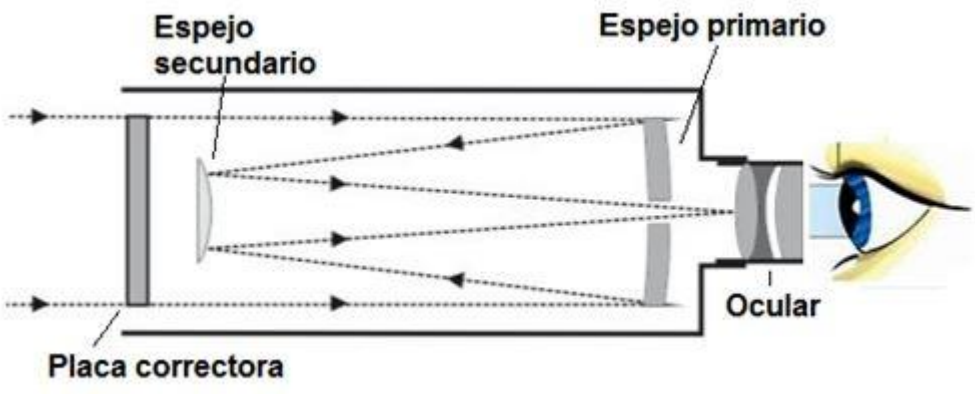


Figura 2: Telescopio Catadióptrico

Existen otras diferencias entre los distintos tipos de telescopios que no serán detalladas, por salirse demasiado del tema de este trabajo. Sin embargo, de aquí en adelante, cada vez que se hable de un telescopio, nos estaremos refiriendo a uno reflectante, de tipo catadióptrico.

2.1.1. Óptica Adaptativa

La óptica adaptativa (AO) es una técnica que permite corregir las perturbaciones más importantes que sufren los telescopios terrestres al realizar la reconstrucción de imágenes astronómicas [1]. Los sistemas de óptica adaptativa en un telescopio, permiten mejorar la calidad de una imagen astronómica, proporcionando las herramientas suficientes para reducir en tiempo real las perturbaciones que presenta un frente de onda al atravesar las diferentes capas atmosféricas. Estos sistemas están compuestos por varios elementos comunes, entre los que destacan el espejo deformable, el sensor de frente de onda y el sistema de procesamiento, que explicaremos a continuación.

Uno de los elementos más importantes que poseen los sistemas de AO, es el sensor de frente de onda (*wave-front sensor* WFS), que es el encargado de medir cuánto se desvía la luz recibida, respecto a su recepción en un frente de onda completamente plano. Existen diferentes tipos de WFS, siendo el más común, y sobre el que se hablará de aquí en adelante, el denominado de *Shack-Hartman* (SH-WFS) [11], aunque también tenemos los sensores de curvatura [12] y los sensores de tipo pirámide [13]. En el caso del SH-WFS, el frente de onda aberrado, atraviesa una matriz de lentes o subaperturas, lo que nos indica las desviaciones que posee la luz al atravesar la atmósfera. En la Figura 3 se puede observar un esquema simplificado, con las diferencias que se obtienen en las imágenes, entre recibir un frente de onda plano, y otro deformado.

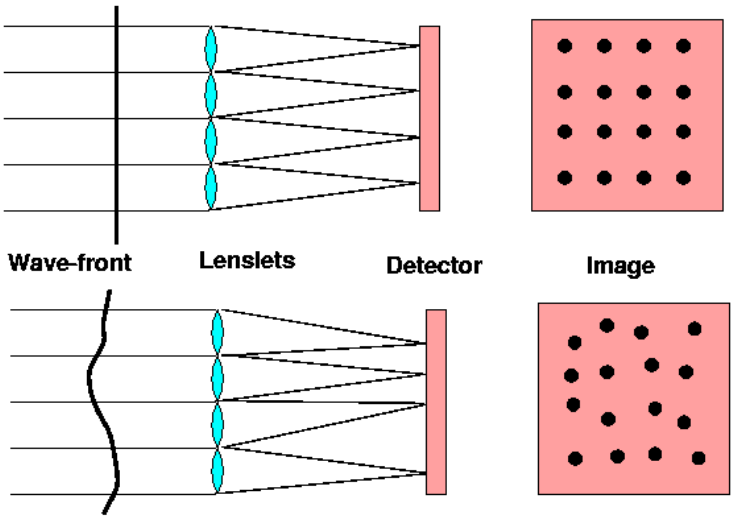


Figura 3: Sensor de Frente de Onda Shack-Hartman

Otro de los elementos claves en los telescopios de gran tamaño, es el espejo deformable. Los más comunes son los de tipo Piezo-Eléctrico [14]. Están compuestos por una membrana deformable muy fina, con capacidad de reflejar la luz, unida a un conjunto de actuadores piezo-eléctricos, cuya función es mover la membrana deformable al aplicarse un determinado voltaje a cada uno de ellos. En la Figura 4, se puede observar el *array* de actuadores, con la membrana deformable encima, además de una imagen de un conjunto de actuadores antes de que se les coloque la capa deformable encima.

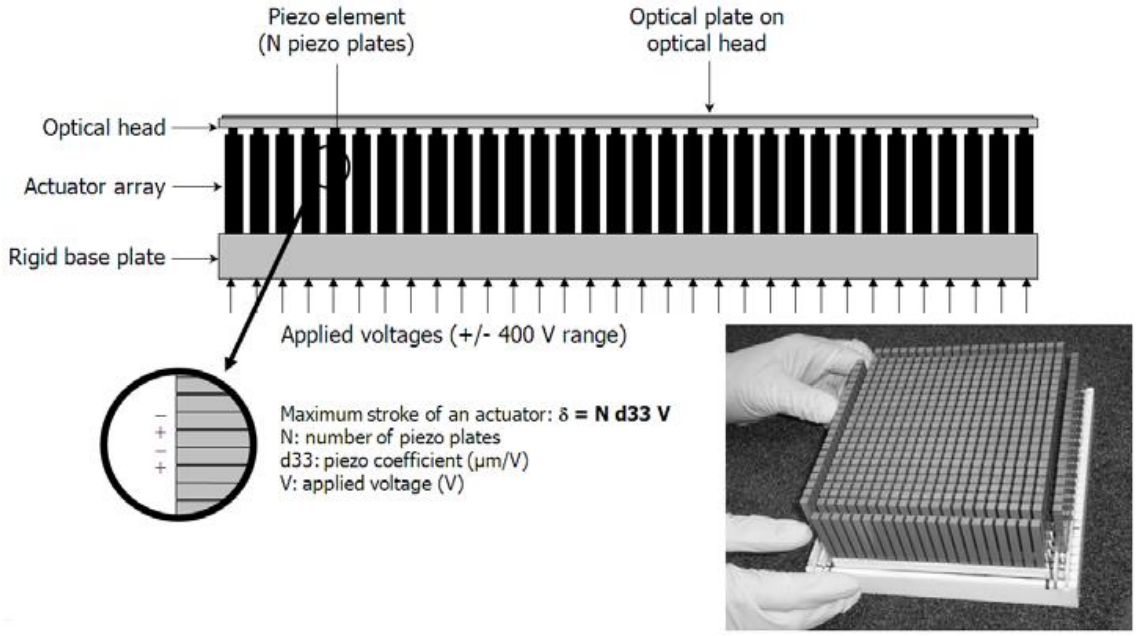


Figura 4: Espejo Deformable

Existen dos tipos de sistemas de AO muy utilizados, los de lazo abierto u *open-loop* y los de lazo cerrado o *closed-loop*. La diferencia entre ellos, es que en el caso de los sistemas de lazo cerrado, la información que se recibe en los sensores, llega después de haber sido reflejada por el espejo deformable, mientras que en el caso del lazo abierto dicha información llega de forma directa de la observación estelar. En la Figura 5, se puede observar un ejemplo de cómo un espejo divide la información entre la que se envía al espejo deformable, y la que es enviada al sensor de frente de onda. A lo largo del presente trabajo, siempre que se haga referencia a este tipo de sistemas, estaremos hablando de los de tipo *open-loop.*

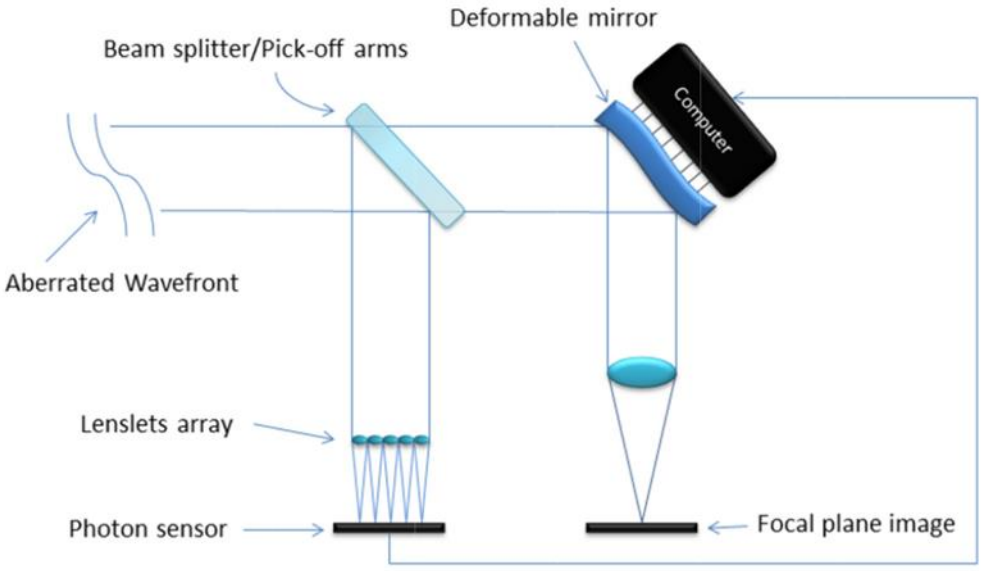


Figura 5: Ejemplo de sistema de Lazo Abierto

Existen numerosas técnicas empleadas en AO, como la Óptica Adaptativa Multi-Conjugada (*Multi-Conjugate Adaptive Optics, MCAO*), la Óptica Adaptativa de Capa Baja (*Ground Layer Adaptive Optics GLAO*) y la Óptica Adaptativa Multi-Objeto (*Multi-Object Adaptive Optics MOAO*), que será a la que se haga referencia de aquí en adelante.

En la Óptica Adaptativa Multi-Objeto, es posible observar diferentes estrellas guías de forma simultánea, tanto estrellas naturales (*Natural Guide Stars, NGS)*, como creadas por láser *(Laser Guide Stars, LGS*), que nos sirven de referencia. Cada una de estas estrellas, posee un WFS encargado de proporcionar la información de entrada, tal y como se puede observar en la Figura 6. Esta información, es utilizada posteriormente por el reconstructor para colocar el espejo deformable, de tal forma que las distintas turbulencias atmosféricas puedan ser compensadas.

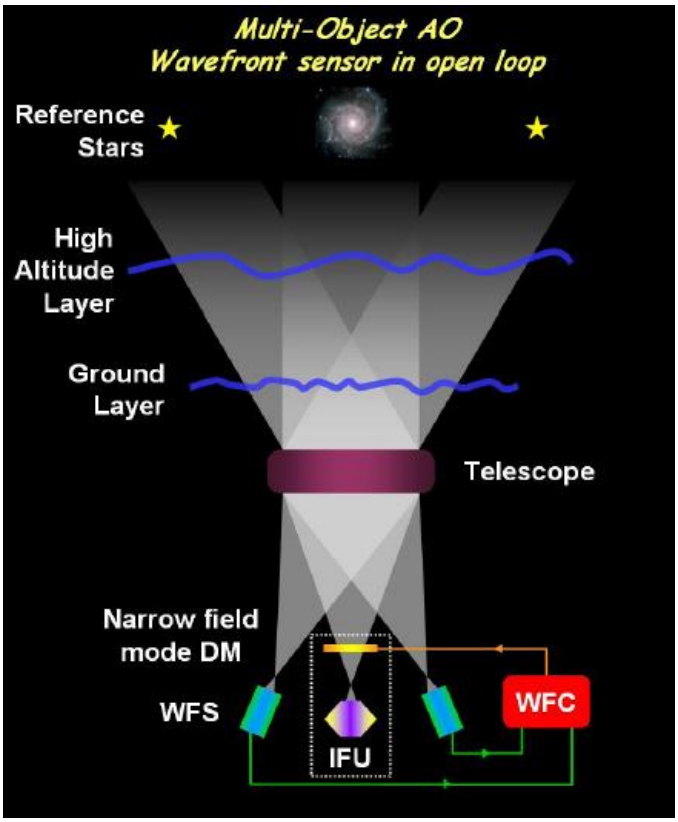


Figura 6: Óptica Adaptativa Multi-Objeto

2.1.2. Sistemas de Óptica Adaptativa

En este apartado, se darán los detalles sobre los sistemas de óptica adaptativa que serán objeto de estudio a lo largo del presente trabajo. Estos sistemas, pueden ser utilizados dentro del laboratorio, aunque su objetivo final es ser instalados en los telescopios de gran tamaño presentes en los observatorios más importantes.

CANARY [15] es un sistema de óptica adaptativa preparado para realizar pruebas en observaciones reales. Su principal objetivo, es probar conceptos de AO para el futuro E-ELT, que posee un diámetro de 39 metros. Actualmente, se encuentra funcionando en una plataforma *Nasmyth*, en el *William Heschel Telescope (WHT)*, de 4.2 metros de diámetro, que forma parte del grupo de telescopios Isaac Newton del Observatorio del Roque de los Muchachos de la Palma, en las Islas Canarias.

El sistema CANARY lleva años siendo desarrollado principalmente por el Departamento de Física de la Universidad de Durham. Durante este periodo, se han ido añadiendo y modificando algunos de sus elementos [6], por lo que nos encontramos con diferentes configuraciones para un mismo sistema. Para el presente trabajo, se van a analizar dos de sus configuraciones, que serán detalladas a continuación:

- CANARY FASE B1: Esta fase se corresponde con el momento en el que se realizaron los estudios de CARMEN en telescopios reales. Está diseñada para

realizar observaciones con una estrella guiada por láser, y hasta cuatro estrellas naturales. Posee un WFS con 7x7 subaperturas, aunque solo 36 de ellas son funcionales debido a la pupila circular del telescopio y a los oscurecimientos secundarios.

- CANARY FASE C2: En este caso, se corresponde con la fase más reciente desarrollada mientras se realizaron las pruebas de este trabajo. Posee 4 estrellas guiadas por láser, y un WFS con 14x14 subaperturas, aunque como en el caso anterior, solo ofrecen resultados funcionales 144 de ellas.

Además del sistema CANARY, que sigue siendo desarrollado, se ha considerado interesante realizar una aproximación al futuro sistema DRAGON. Este aún se encuentra en su fase inicial de desarrollo, por lo que no existe información precisa sobre el número exacto de entradas o estrellas que el sistema puede manejar. Sin embargo, ha sido posible hacer algunas estimaciones en función de la información proporcionada por el equipo de desarrollo de la Universidad de Durham.

- DRAGON: Sigue una filosofía similar a la de CANARY, aunque con el objetivo de ser un sistema bastante más moderno y considerablemente más grande. Se prevé que pueda funcionar utilizando 4 estrellas guiadas por láser, y hasta cuatro estrellas naturales como guía. El tamaño del WFS estimado es de 30x30, y dado que no tenemos información sobre la cantidad de subaperturas funcionales, las pruebas se van a realizar asumiendo que tendremos salidas en todas ellas, que sería el escenario más complejo.

Uno de los objetivos iniciales del presente TFM era poder realizar un análisis sobre un sistema con dimensiones similares al futuro E-ELT, que poseería alrededor de unos 100.000 entradas y aproximadamente unas 10.000 salidas, según los datos estimados [8]. Sin embargo, tras el estudio inicial, esta idea tuvo que descartarse, ya que una red de ese tamaño, requeriría una gigantesca cantidad de memoria para almacenar la información, por lo que no ha sido posible realizar dichas pruebas con los equipos de los que se ha dispuesto para este trabajo.

2.2. Redes Neuronales

2.2.1. Introducción Histórica

Corría el año 1943 cuando Warren McCulloch y Walter Pitts tuvieron una nueva idea. Había pasado casi medio siglo desde que Ramón y Cajal explicase el funcionamiento de las neuronas humanas, cuando estos dos científicos estadounidenses publicaron su artículo "*Cálculo lógico de Ideas inherentes en la*

actividad nerviosa" [16]. En él proponían la idea de crear una neurona artificial, que permitiese realizar operaciones lógicas.

Su propuesta, tal y como se puede ver en la Figura 7, consistía en el sumatorio de una serie de entradas multiplicadas por unos determinados pesos, seguidos por la activación de una función no lineal. Siguiendo la comparación con la neurona humana, se podría considerar que las entradas y sus pesos, son el equivalente a las dendritas. El soma o núcleo, se correspondería con el sumatorio de dichos pesos, mientras que el axón se correspondería con la función de salida.

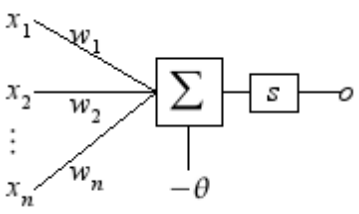


Figura 7: Neurona de McCulloch-Pitts

Pese a lo novedoso de la propuesta, esta neurona estaba pensada para calcular de forma manual los pesos correspondientes a las entradas, y estaba seriamente limitada por su salida, ya que era binaria. En 1949 el psicólogo Donald O. Hebb, introduce una importante novedad en este campo. En su libro *La Organización del Comportamiento*, Hebb propone un modelo para modificar los pesos de las neuronas en función de sus conexiones. Esta idea, aunque principalmente enfocada al comportamiento de una neurona biológica, permite establecer ciertas reglas que faciliten el aprendizaje de una neurona artificial.

Con estas dos ideas en mente, y tras la acuñación del término Inteligencia Artificial en el año 1956 en la "Conferencia Darmouth", Frank Rosenblatt inventa el perceptrón en el año 1957 [17]. En aquel momento, se encontraba trabajando en el Laboratorio Aeronáutico de Cornell, financiado por la oficina de Investigación Naval de los Estados Unidos. Esto provoca que no sea hasta el año siguiente, cuando su invento es anunciado al mundo, asegurando que la Marina posee un ordenador que se espera que sea capaz de hablar, ver o escribir, es decir, tener consciencia propia. Esto provoca un gran revuelo a la vez de generar una gran expectación.

El perceptrón propuesto, era una máquina diseñada con el objetivo de reconocer imágenes. Sin embargo, el algoritmo utilizado, aunque era semejante a lo visto anteriormente, aportaba nuevas ideas que hacían aumentar de forma notoria las posibilidades que este ofrecía. Por un lado, nos encontramos que aunque utiliza una neurona igual que la empleada por McCulloch y Pitts en la Figura 7, Rosenblatt propone añadir un umbral que se sumará al resultado de la multiplicación de las entradas por los pesos. Además, propone funciones de activación distintas a la función escalón propuesta originalmente, como la sigmoidea, la gaussiana o una lineal a trozos. Esto nos da como resultado que la salida Y de una neurona, se puede obtener calculando lo

expresado en la Ecuación 1, siendo x las entradas, w los pesos, f la función de activación y n el total de las entradas.

$$\mathrm{Y} = f\left(\sum_{i=0}^{n}(w_i \cdot x_i) + b\right)$$

Ecuación 1: Perceptrón Simple

Sin embargo, la mayor novedad introducida, es que el perceptrón posee un algoritmo que le permite aprender a calcular los pesos de forma autónoma. Debido a su formulación matemática, un perceptrón puede clasificar dos subconjuntos de elementos, que sean separables linealmente. Para ello, es necesario calcular los pesos de las entradas, y en este caso, se utilizarán datos conocidos para enseñar o entrenar a clasificar a la neurona. Para ello, será necesario inicializar los pesos de forma aleatoria, y a continuación utilizar un dato cuyas entradas y salidas sean conocidas, para calcular la salida Y de la Ecuación 1. A continuación, será necesario calcular la diferencia entre nuestra salida Y, y la salida esperada Y', obteniendo el error δ. Este error, multiplicado por el valor de la entrada x correspondiente, servirá para actualizar el peso w, tal como se puede ver en la Ecuación 2.

$$w_i' = w_i + (Y' - Y)x_i$$

Ecuación 2: Regla Aprendizaje

Este proceso es iterativo, y es necesario repetirlo con todos los pesos de entrada al perceptrón, y durante un número determinado de veces. En 1962, Novikoff demostró que el algoritmo de aprendizaje converge en un número finito de iteraciones, si los datos son separables linealmente, y el número de posibles errores está limitado. Aparecen también los primeros sistemas de interconexión de varios perceptrones, como ADALINE, que propone un sistema con varias neuronas interconectadas y varias salidas.

Estos descubrimientos generaron cierta euforia dentro de la comunidad científica. Se creía que esta nueva idea, permitiría a las máquinas aprender a hacer cualquier cosa, únicamente enseñándole la cantidad suficiente de información para que ellas solas entrenasen. Aparecen incluso pequeñas modificaciones respecto al perceptrón original, como el llamado *kernel perceptron*, que permitía clasificar elementos que no eran separables linealmente.

Dentro del gran optimismo generado, aparecen numerosos escépticos. Una de las figuras más importantes, y principal promotor del término inteligencia artifical es Marvin Minsky. Co-fundador del laboratorio de IA del *Massachusetts Institute of Technology*, y conocido desde la adolescencia de Rosenblatt, Minsky escribe en 1969 junto a Seymour Papert el libro "Perceptrones: Una introducción a la geometría

computacional” [18]. En él, se hace una crítica profunda al camino que había tomado la inteligencia artificial, al haberse centrado especialmente en el uso de perceptrones, y se prueban sus supuestas limitaciones, al demostrar matemáticamente que un perceptrón es incapaz de computar una operación XOR.

Pese al teórico potencial de este tipo de neuronas para computar todo tipo de operaciones binarias, esta limitación supone un jarro de agua fría. Gran parte de la lógica del momento, estaba basada en este tipo de operaciones, y una de las teóricas fortalezas iniciales de las redes, era su capacidad de poder aprender cualquier programa que pudiese simularse por ordenador. Sin embargo, las justificaciones de una eminencia como Minsky, suponen un enfriamiento importante en el uso de las redes neuronales, tanto a nivel de financiación como de investigación en ellas.

Tras muchos años de abandono, un nuevo empuje surge durante los años 80. Eran numerosas las personas que pensaban que el análisis de Minsky no había sido del todo correcto, y que existían formas de implementar una puerta XOR utilizando perceptrones. Se populariza entonces una nueva forma de entrenamiento, conocida como retropropagación, que permitía aprender a redes compuestas por más de una capa de neuronas. Esto permite crear redes compuestas por varios perceptrones, agrupados en una o más capas ocultas, que poseen además una capa de entrada y otra de salida, tal y como se puede ver en la Figura 8. Sin embargo, hay que tener en cuenta que es una simplificación, ya que una red de este tipo, conocida como perceptrón multicapa, puede poseer más de una capa oculta, y tener más de una salida.

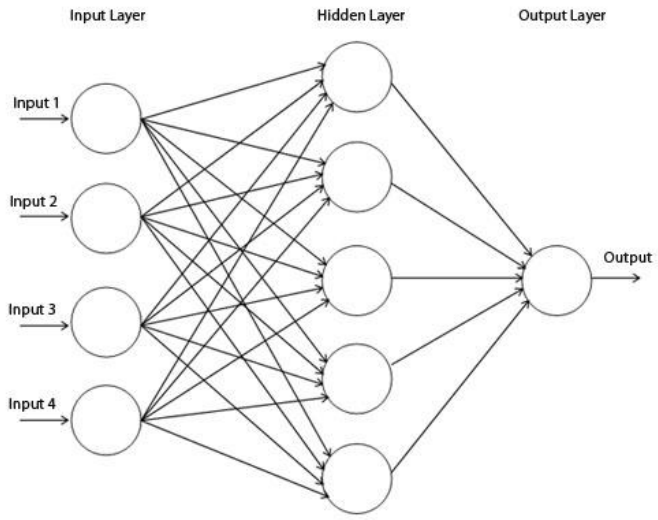


Figura 8 : Perceptrón Multicapa

El algoritmo de retropropagación es desarrollado por Paul Werbos [19] en 1975, aunque su popularización no llegaría hasta más tarde con la publicación del libro “Procesamiento Distribuido Paralelo” [20], de Rumelhart y McClelland. La utilización de

la retropropagación en un perceptrón multicapa, permitía crear y entrenar una red que resolviese con facilidad el problema de la puerta XOR, eliminando así el principal problema propuesto por Minsky en su libro. Sin embargo, este descubrimiento no consiguió que las redes neuronales retomasen la gran popularidad que habían tenido durante la década de los 60. El algoritmo de retropropagación era considerablemente lento, necesitando en ocasiones miles de iteraciones para converger a unos pesos adecuados, problema que se multiplicaba especialmente en redes con muchas neuronas o muchas capas ocultas, aunque no impidió seguir investigando en la mejora de este tipo de redes.

Durante los siguientes años empiezan a surgir numerosas técnicas que optimizan y aceleran el entrenamiento de las redes. Por un lado, empiezan a emplearse no solo como clasificadores, sino también como sustituto de los regresores lineales y logísticos. Aparecen mejoras del algoritmo de retropropagación, como el uso del algoritmo de descenso del gradiente y descenso del gradiente estocástico [21], que consisten en minimizar el valor de una función de error, en lugar de minimizar de forma directa el error que se tiene a la salida de la red, como se hacía en la Ecuación 2. Se retoman y desarrollan ideas surgidas en los 80, como el concepto de red neuronal recurrente desarrollado por Hopfield en 1982, y que pretendía dotar de memoria temporal a las redes. Esta idea pretendía que la salida de una red, no dependiese solo de sus entradas, como se puede ver en la Figura 9, sino que también estuviese afectada por las entradas de los instantes anteriores. Entre sus nuevas ideas destacan las redes con memoria a corto plazo [22], que son bastante fáciles de entrenar, debido a su relativa simplicidad.

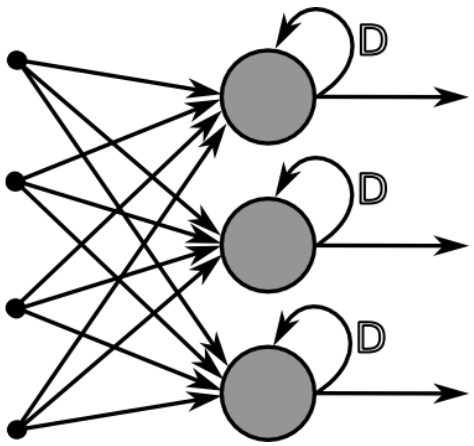


Figura 9: Red Neuronal Recurrente

Surgen también nuevos conceptos, como el de aprendizaje no supervisado [23] en el que se pretende entrenar una red, utilizando para ello muestras de entrenamiento de las cuales se desconoce su salida. Teniendo en mente el funcionamiento del algoritmo de retropropagación, podemos deducir cómo esto supone un pequeño reto y cambio en

la forma tradicional de entrenar una red, ya que no tenemos ahora una señal de salida que nos permita generar un error que retropropagar para corregir los pesos. Éstos son tan solo algunos de los ejemplos de distintas ideas que surgieron dentro del campo de las redes neuronales durante aquellos años, y que aparecieron en paralelo con las primeras aplicaciones de las redes neuronales más tradicionales a usos en la vida cotidiana. Durante esta época, tienen especial éxito el uso de las redes neuronales en el reconocimiento de voz, la identificación de la escritura manuscrita o el reconocimiento de objetos.

Con la entrada del nuevo milenio, y la constante mejora de la capacidad de procesado de los ordenadores, empieza a surgir y extenderse un nuevo término: el *"Deep Learning*" o "Aprendizaje Profundo" [24]. Esta idea en realidad, no aporta nada novedoso, sino que simplemente se introduce para diferenciar de forma más clara, el avance que habían ido teniendo las redes durante los últimos años. Todas las mejoras en los algoritmos, la potencia de procesamiento, etc, habían permitido que las redes estuviesen formadas por varias capas ocultas. Además, empiezan a cobrar cada vez más fuerza el uso de las capas convolucionales, que consistían en aplicar un filtro a cada una de las entradas, además de añadir una teórica tercera dimensión a las redes.

Durante este tiempo, aparecen numerosas topologías de red, que combinan todos los conceptos anteriores, con nuevas capas técnicas, adaptándose su arquitectura, al tipo de problema que se quiere resolver. Uno de los test más habituales para evaluar la calidad de cualquier red, o de cualquier sistema de inteligencia artificial, es el de reconocimiento de números escritos por humanos. Para ello, se utiliza la base de datos MNIST [25], que recoge 70.000 imágenes de números manuscritos, agrupadas en 60.000 imágenes para entrenamiento, y 10.000 que sirven para evaluar el rendimiento o la fiabilidad del sistema utilizado. Tal y como se puede ver en [25], las redes neuronales convolucionales han llegado a alcanzar un porcentaje de error en el test del 0.23%, es decir, fallando únicamente en 23 imágenes de las 10.000.

Uno de los cambios más recientes, y que ha permitido la definitiva explosión del aprendizaje profundo, es el cambio en la forma de programar los algoritmos de aprendizaje y ejecución de las redes. Tradicionalmente, en un entrenamiento de una red, todas las operaciones eran ejecutadas de forma secuencial en la CPU principal. Esto podía llegar a suponer tiempos de entrenamientos muy altos, especialmente en redes muy grandes o con enormes cantidades de datos para entrenar. Sin embargo, a principios de la presente década, numerosos investigadores se dan cuenta de que gran parte de las operaciones que se ejecutan en la red, son independientes entre sí, y que pueden ejecutarse de forma paralela. Esto coincide con el auge de la programación paralela de propósito general en unidades gráficas (*GPGPU* por sus siglas en inglés), que

permite acelerar enormemente este tipo de cálculos. En muy poco tiempo, se produce un gran cambio en el que casi todas las redes, pasan a desarrollarse bajo este nuevo paradigma, ya que es posible entrenar redes mucho más complejas y con mucha más cantidad de datos, en periodos mucho más cortos de tiempo.

Es muy abundante la literatura relacionada con las mejoras en los tiempos de entrenamiento y de ejecución de las redes, en una GPU en lugar de una CPU. Quizás, una forma más intuitiva de ver este cambio, es la que podemos observar en la Figura 10. En ella se recogen estadísticas del reto ILSVRC (*ImageNet Large Scale Visual Recognition Challenge*), consistente en que distintos equipos ponen a prueba sus arquitecturas, tratando de reconocer e identificar patrones en imágenes. Podemos ver como desde el año 2012 en adelante, el aumento de equipos que utilizan GPUs para sus desarrollos llega a alcanzar el 90%, además de cómo el porcentaje de error en el reconocimiento, es cada vez más pequeño.

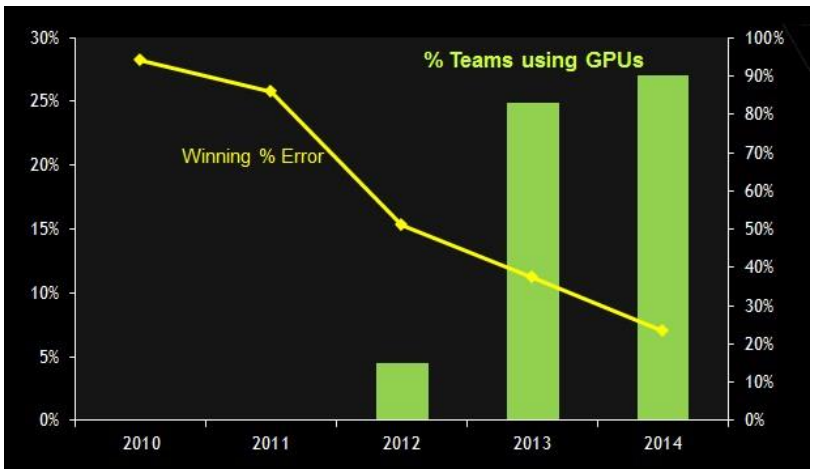


Figura 10: Utilización GPUs

A día de hoy, prácticamente todos los gigantes de la industria tecnológica, están invirtiendo miles de millones de euros en el desarrollo de sus propias redes. Desde Facebook con su reconocimiento de imagen, a Google con su coche autónomo, pasando por Apple o Microsoft con sus asistentes de voz, demuestran que ahora mismo, las redes neuronales están más vivas que nunca.

2.2.2. Funcionamiento de una red neuronal

Durante el apartado anterior hemos dado un pequeño repaso a la historia de las redes. Se ha hablado del enorme potencial que tienen, pero tan solo hemos dado unas mínimas nociones matemáticas sobre el funcionamiento de una red. A lo largo de este capítulo, vamos a intentar abordar de la manera más sencilla posible, cómo podemos desarrollar, ejecutar y entrenar nuestro propio perceptrón multicapa con una sola capa

oculta. Este tipo de red es uno de los ejemplos más sencillos que se pueden encontrar, lo cual no está reñido con su posible potencial en aplicaciones reales.

El primer paso a la hora de crear una red, es decidir su topología. Para ello, contamos con dos parámetros externos, que son el número de entradas de nuestros datos, y el número de salidas. Por ejemplo, si queremos crear una red que nos simulase una puerta XOR, sabríamos que el número de entradas de nuestra red va a ser 2, y que el número de salidas será 1. La siguiente decisión, es pensar en cuántas capas ocultas queremos que posea nuestra red. Este tipo de decisiones, como casi todas las que tomemos, son experimentales, y apenas existen reglas que siempre se cumplan. En nuestro caso y por simplicidad, vamos a optar por que nuestra red posea una única capa oculta. Una vez tenemos este dato, podemos decidir el número de neuronas que poseerá dicha capa, que van a ser 2. A continuación además, también será necesario decidir la función de activación de nuestra neurona, que será una sigmoide, aunque también podría ser la tangente hiperbólica, la función escalón o una función lineal a trozos. A excepción del número de entradas y salidas, cualquiera de estas decisiones pueden ser modificadas a posteriori, buscando que la red que creemos, obtenga los mejores resultados posibles.

Con todos estos datos, ya tenemos una topología de red como la que puede ver en la Figura 11, con dos neuronas de entrada, dos neuronas en la capa oculta, una neurona de salida, y las correspondientes conexiones entre las neuronas. Este tipo de redes, en las que todas las neuronas de cada capa, están conectadas con todas las neuronas de la siguiente, se conocen como capas “totalmente conectadas”.

Además de la topología de la red, necesitamos también conocer nuestros datos de entrenamiento. Si tenemos en cuenta que queremos crear una puerta XOR, nuestros datos se corresponderán con los que aparecen en la Tabla 1. En ella podemos observar los valores de nuestras dos entradas, y el valor que se correspondería con nuestra salida. Por último, necesitamos asignar unos valores iniciales a nuestros pesos (W) y al umbral (b), también conocido como bias. Habitualmente, esta inicialización es aleatoria, utilizando distribuciones lineales o gaussianas, pero por simplicidad en nuestro caso, haremos que todos los valores sean 0.

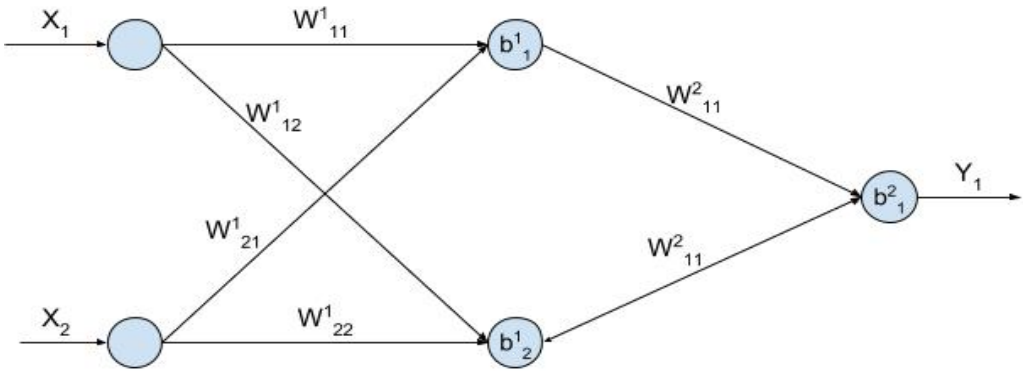


Figura 11: Red Neuronal XOR

Entradas		Salidas
X1	X2	Y1
0	0	0
0	1	1
1	0	1
1	1	0

Tabla 1: Datos de Entrenamiento XOR

Con todo esto, ya podemos calcular cuáles son los valores de salida de nuestra red, si las entradas fuesen algunas de las que aparecen en la Tabla 1. Para ello, será necesario utilizar la Ecuación 3, muy similar a la vista en el capítulo anterior, y que nos permite calcular el valor de salida de cualquiera de nuestras neuronas. En este caso, vamos a optar por obtener la salida que nos devuelve la red, cuando a su entrada tenemos los valores (0,0).

$$\mathrm{Y} = Sigm\left(\sum_{i=0}^{n}(w_i \cdot x_i) + b\right)$$

Ecuación 3: Activación Neurona

Teniendo en cuenta que todos los pesos y bias son 0, nos encontramos con que las salidas de las dos neuronas de la capa oculta, son 0.5, ya que al sustituir en la Ecuación 3, tenemos que $Sigm((0 \cdot 0) + (0 \cdot 0) + 0)$ cuyo valor es 0.5. Esto hace, que el valor de nuestra neurona de salida sea $Sigm((0.5 \cdot 0) + (0.5 \cdot 0) + 0)$, cuyo valor, es de nuevo 0.5. Éste valor, está considerablemente alejado del valor que queremos obtener, que es un valor de 0. Es aquí cuando entra en acción, una de las principales fortalezas de las redes neuronales, que es su capacidad de autoaprendizaje con datos conocidos. Para

ello, vamos a utilizar el algoritmo de retropropagación, combinado con un descenso de gradiente.

El objetivo de esta idea es sencillo. Vamos a definir una función de error, que en nuestro caso puede ser el error cuadrático medio, aunque existen gran cantidad de funciones disponibles. Esta función, computará todos los errores de todas las neuronas de salida, respecto a los valores teóricos que debería dar según nuestros valores de entrenamiento. Intuitivamente, podemos darnos cuenta de que si nuestra función de error alcanza un valor mínimo, esto quiere decir que nuestra red está cada vez más cerca de tener a su salida el resultado correcto, ya que si el error es muy pequeño, quiere decir que la salida de la red, es muy parecida a salida que deseamos obtener. A la hora de obtener el valor mínimo de esta función, es cuando se utiliza el descenso del gradiente.

Para explicar el descenso del gradiente, es muy habitual utilizar una analogía de una bola deslizándose por una cuesta. Imaginemos una función $f = (Y - a)^2$, que se correspondería con el cálculo del error cuadrático medio de nuestra red, siendo Y la salida de nuestra red, y a el valor de la salida de entrenamiento. Es relativamente fácil de imaginar que si colocásemos una bola en la Figura 12, esta caería hasta alcanzar el valor mínimo, que en este caso sería el 0. Una representación matemática de dicha caída, sería calcular el gradiente que tiene la bola en un determinado punto. Para ello, habría que calcular las derivadas parciales de la función anterior, que en este caso nos darían $\nabla f = 2Y - 2a$, al tener una sola variable. Esto nos indicaría la dirección que tiene que tomar la bola para continuar cayendo. Llegados a este punto, es trivial de ver que cuando nuestra salida Y, sea igual al valor previsto a, el gradiente, se anula, por lo que nuestra bola no se movería más, y habríamos alcanzado la convergencia.

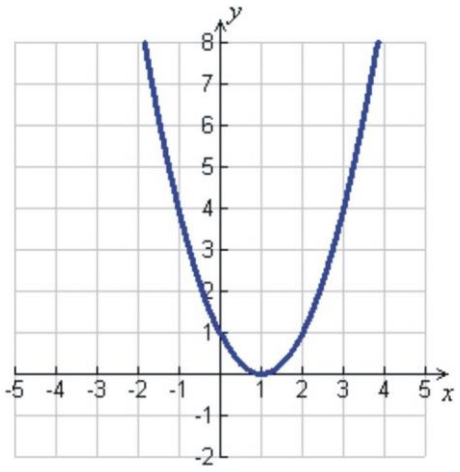


Figura 12: Parábola Desviada

El ejemplo utilizado, es una simplificación importante respecto a lo que realmente pasa en cualquier red neuronal mínimamente compleja. Las redes pueden tener cientos o miles de variables de salida, complicando infinitamente la representación de la función. Sin embargo, el concepto del gradiente, puede aplicarse a un número infinito de dimensiones, ya que siempre buscará una forma de alcanzar un mínimo de la función. Este valor que obtenemos, es tan solo una pista sobre la dirección que debemos tomar para minimizar el valor de nuestra función de error, y será utilizado para corregir los pesos y los bias de nuestra red.

Es en este punto, donde es necesario comenzar a utilizar el algoritmo de retropropagación propiamente dicho, compuesto principalmente por cuatro ecuaciones que pueden consultarse en la Tabla 2 obtenida de [13]. En primer lugar, necesitamos calcular el error de salida, que es la multiplicación del gradiente de la función de error, por la derivada de la función de activación. Esto nos queda $\delta^L = 0.5 \cdot (0.5 \cdot (1 - 0.5)$ que es igual a 0.25.

Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \qquad \text{(BP1)}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \qquad \text{(BP2)}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \qquad \text{(BP3)}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \qquad \text{(BP4)}$$

Tabla 2: Ecuaciones de Retropropagación

Una vez que hemos obtenido el error a la salida, es el momento de retropropagar el error a través de la red. Para ello, es necesario multiplicar el error, por el peso asignado a la conexión entre dos neuronas. Esto hace que el error se pondere, en función de la importancia o peso que tiene esa neurona en la capa siguiente. Debido a nuestra inicialización de pesos nula, nos encontramos con que los errores que hay a la salida de la capa oculta, son nulos en ambos casos, ya que ninguna de las dos neuronas tienen a priori influencia en la salida.

Por último, una vez que tenemos los errores retropropagados en cada capa, es necesario actualizar los pesos y los bias de la red. En el caso de los pesos, se cumple que

$w' = w - (\lambda \cdot \delta \cdot a)$, siendo w' el nuevo peso, w el peso que se quiere actualizar, λ una constante conocida como ratio de aprendizaje, δ el error en dicha neurona y a el valor de salida de la neurona. Si aplicamos dicha ecuación para actualizar el peso w^2_{11}, con una constante de aprendizaje de 1, tendremos que $w_{11}^{2\,\prime} = 0 - (1 \cdot 0.25 \cdot 0.5)$ que resulta en un valor de -0.125. Por simetría, el valor obtenido es el mismo para el caso de w_{12}^2. El siguiente paso, sería actualizar los pesos que conectan las neuronas de entrada, con las neuronas de la capa oculta. Sin embargo es fácil ver que al poseer ambas neuronas un δ = 0, no se produciría actualización alguna de los pesos. No hay que olvidarse de actualizar también el bias, que cumple que $b' = b - (\lambda \cdot \delta)$, por lo que en la neurona de salida, tendremos $b_1^{2\,\prime} = 0 - (1 \cdot 0.25)$, que nos da un resultado de -0.25.

Es el momento de comprobar si con nuestros nuevos pesos, la red ha aprendido algo, o sigue sin conseguir nada. Al no haber modificado los pesos de la primera capa, seguimos teniendo que las salidas de las neuronas de la capa oculta, son ambas 0.5. Es ahora el momento de calcular el valor de la neurona de salida con la Ecuación 3. Tenemos que $Y = Sigm\left(\left(0.5 \cdot (-0.125)\right) + \left(0.5 \cdot (-0.125)\right) + (-0.25)\right)$ que nos da 0.407. Esto supone una pequeña mejora respecto al 0.5 inicial, lo que nos hace ver que nuestra red, ha conseguido aprender.

No hay que olvidarse, que este proceso es iterativo, y que es necesario repetirlo en varias ocasiones, para conseguir que el error de nuestra red sea lo más pequeño posible. En este caso, después de utilizar la primera entrada de la Tabla 1, sería necesario repetir este mismo proceso con la segunda entrada, luego con la tercera y la cuarta. El realizar una actualización de nuestra red, utilizando todos los datos de entrenamiento, es conocido como un *epoch*. Habitualmente, es habitual tener que realizar varios *epochs*, hasta que nuestra red converja, o bien nos de unos valores de error suficientemente pequeños. La cantidad a realizar, depende de muchos factores. Uno de ellos es el ratio de aprendizaje λ, que nos indica la rapidez con la que aprende nuestra red, ya que hace más grande el "paso" que daría nuestra bola al bajar por la función. También depende enormemente de la cantidad de datos de entrenamiento que poseamos, de la topología de la red o las funciones de activación empleadas.

Para terminar este apartado, creo que es interesante hacer unas pequeñas reflexiones sobre el ejemplo analizado aquí. Para empezar, no es difícil darse cuenta de que nuestra red nunca podrá tener un error 0. La función de activación sigmoidea solo alcanza estos valores en $\pm\infty$, por lo que necesitaríamos un número infinito de iteraciones para conseguir un error nulo. Esto nos hace pensar sobre la importancia que tiene la elección de la función de activación, en consecuencia con los datos con los que se esté trabajando. También puede resultar llamativo que debido a nuestra inicialización de los bias y los pesos, no hayamos podido retropropagar el error desde la capa oculta

hacia la capa de salida. Este caso, también ilustra que una inicialización de pesos correcta, es bastante importante a la hora de trabajar con una red. Es bastante habitual el uso de distribuciones aleatorias, tanto normales como gaussianas, que nos ayuden a dar valores distintos a 0 todos los pesos. Dicha elección depende en gran medida de la red que queramos crear.

Otro de los aspectos que también pueden llamar la atención, es el problema de que en redes más complejas, nuestro gradiente se quede estancado en un mínimo local en lugar de en un mínimo global. Esto haría que nuestra red dejase de aprender antes de llegar a un valor óptimo. Esto es un problema bastante habitual en todas las redes neuronales, y existen algunas ideas que pueden ayudar a solucionarlo. La más extendida es el momentum, que sigue también la idea de una bola que desciende hacia un valle, pero que además de seguir la dirección indicada por el gradiente en un momento determinado, posee una cierta "velocidad". Esta idea permite que nuestra bola, cuando alcanza un determinado valle, pueda seguir subiendo un poco antes de estabilizarse, ya que posee cierta inercia. Si el mínimo alcanzado es un mínimo local, es posible que debido a la "velocidad" adquirida, pueda salir y continuar hacia un mínimo absoluto.

Estas son solo algunas de las ideas más básicas relacionadas con las redes neuronales, aunque a día de hoy existen infinidad de mejoras, técnicas y nuevos conceptos, evolucionando todo a velocidades de vértigo. Actualmente, existen redes con más de 150 capas intermedias, que combinan numerosas y novedosas técnicas, pero su manejo y dominio requiere gran trabajo y práctica.

2.3. Introducción a CARMEN

A lo largo de este apartado, se van a dar unas pequeñas explicaciones sobre el *Complex Atmospheric Reconstructor based on Machine lEarNing*, o CARMEN, relacionando su funcionamiento con los conceptos explicados anteriormente, y mostrando algunos de los resultados obtenidos en pruebas con datos reales, tanto dentro del laboratorio como en observaciones en telescopios.

La reconstrucción tomográfica en AO es un proceso matemático que permite estimar el volumen de turbulencia en la atmósfera a partir de la medición de porciones de esta [26]. En algunas posiciones del campo del telescopio, la estimación se hace utilizando la luz de estrellas guías láser o artificiales, mientras que en otras zonas, es posible la utilización de estrellas naturales que aparecen en el firmamento.

Uno de los problemas más relevantes que se presenta en la reconstrucción tomográfica, está relacionado con la turbulencia atmosférica, pues dicho fenómeno es

aleatorio y surge como consecuencia del calentamiento de las moléculas de gas que conforman la atmósfera. Cuando la luz atraviesa dicha atmósfera, se ve afectada por las diferentes capas formadas por la interacción entre moléculas, ocasionando distorsión en la luz captada por el telescopio.

En el caso de CARMEN, tenemos un reconstructor que pretende corregir la distorsión introducida por la atmósfera, mediante la utilización de redes neuronales. Para ello, se hará uso de la información proporcionada por los sensores de frente de onda, con el fin de modificar los espejos deformables que poseen los sistemas en tiempo real explicados en los apartados previos.

Este reconstructor lleva años siendo desarrollado en la Universidad de Oviedo, en colaboración con la Universidad Católica de Chile, y el Departamento de Física de la Universidad de Durham. Fue originalmente desarrollado utilizando un análisis regresivo *MARS (Multivariate Adaptive Regression Splines)*, con el fin de modificar de forma directa los voltajes que actuaban sobre el espejo deformable [14]. Sin embargo, estudios posteriores demostraron que el uso de redes neuronales, ofrecía unos resultados mejores, además de un mayor margen de mejora [27].

Una vez se empezó a utilizar un reconstructor basado en redes neuronales, fue necesario analizar cuáles eran las entradas y salidas más adecuadas para dicha red. Originalmente, se hicieron pruebas utilizando los valores proporcionados por los diferentes WFS como entradas, intentando estimar los coeficientes de Zernike como salidas, lo que permitió definir una topología adecuada para la red [28]. En posteriores artículos, se demostró cómo su rendimiento podía ser superior al de diferentes reconstructores también utilizados en el campo de la óptica adaptativa, utilizando para ello datos simulados en el laboratorio [29] [30].

Tras los buenos resultados obtenidos en las pruebas, el reconstructor fue probado en condiciones reales, en el *William Herschel Telescope (WHT)*, ubicado en el Roque de los Muchachos, en la Palma de Gran Canaria. Durante dos noches de observación, pudieron analizarse los resultados obtenidos, y compararlos con los proporcionados por los reconstructores basados en mínimos cuadrados y *Learn and Apply* (*L+A*). Durante dichas observaciones, pudo comprobarse como el rendimiento ofrecido por CARMEN era muy similar al arrojado por L+A, siendo los dos mejores reconstructores [5], e incluso se demostró que el rendimiento de CARMEN podía ser bastante superior si las condiciones eran muy cambiantes [31].

Estos estudios fueron agrupados en [32], donde se expone el potencial de la aplicación de las redes neuronales dentro del campo de la óptica adaptativa. Además, deja la puerta abierta a posibles cambios y mejoras que deben ser abordados en el

futuro, como el uso de redes neuronales más grandes y complejas, o su implementación en GPUs que permitan un procesado más rápido de la información.

2.3.1. Funcionamiento de CARMEN

Como se ha explicado anteriormente, CARMEN es un reconstructor basado en redes neuronales. Actualmente, la red utilizada se corresponde con un perceptrón multi-capa, de una sola capa oculta. El número de entradas de la red neuronal, guarda una relación directa con el número de estrellas de referencia que se estén observando, multiplicado por el número de subaperturas que posea el sensor de frente de onda de cada sistema.

En la Figura 13, podemos observar una imagen de lo que recibiríamos en un WFS, al observar una estrella guía. En dicha imagen, se pueden ver algunas de las características mencionadas anteriormente, como la forma circular de la pupila, y el oscurecimiento central que se produce, inutilizando una parte de las entradas que posee el sensor de frente de onda. También se puede ver, cómo en cada subapertura, se tiene una serie de píxeles dispersos, que se corresponden con la luz recibida. En el caso de CARMEN, se obtendrá el centroide de cada una de las aperturas, con sus correspondientes coordenadas *x* e *y*, que serán los valores de entrada que se proporcionarán a la red.

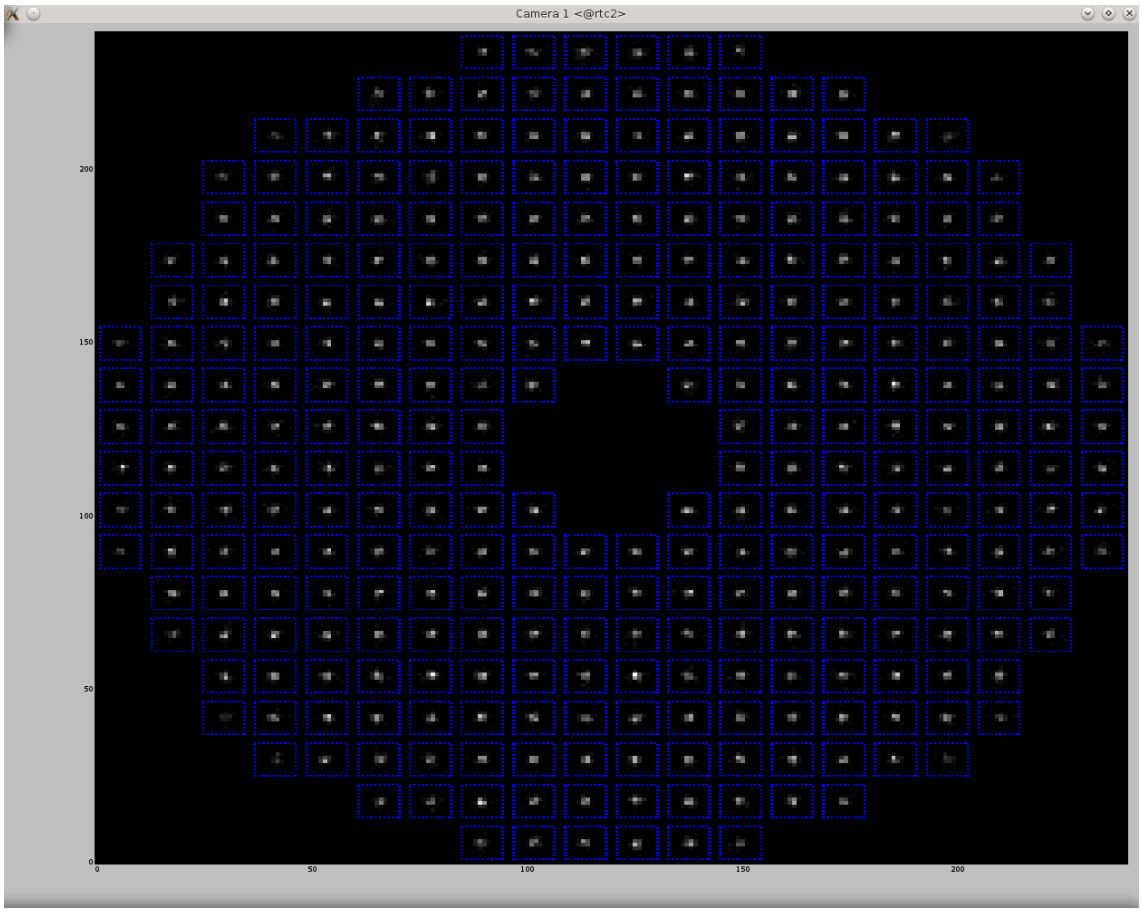


Figura 13: Imagen recibida en un WFS

Originalmente, a su salida CARMEN intentaba estimar los coeficientes de los polinomios de Zernike, que permitirían reconstruir el frente de onda deformado, y adaptar el espejo deformable para compensar esta distorsión. Sin embargo, actualmente CARMEN proporciona en su salida, una estimación de los centroides del asterismo que se esté observando. Con esta información, es posible también reconstruir el frente de onda, y modificar el espejo deformable. Esto hace, que el número de salidas de CARMEN, sea igual al número de subaperturas funcionales del WFS, multiplicado por dos, debido a las dos dimensiones que posee cada centroide.

En el caso de la capa oculta, posee el mismo número de neuronas que la capa de entrada. Este dato ha sido tomado de los estudios previos realizados, donde se analizaban diferentes topologías de la red [28], y se concluía que los mejores resultados se obtenían haciendo coincidir el número de neuronas de la capa oculta con la capa de entrada. Aunque este dato puede ser objeto de futuros estudios, especialmente cuando el tamaño de las redes aumente, y las matrices de pesos crezcan de forma exponencial, se ha optado por dejar fuera del objetivo de estudio de este trabajo. En la Figura 14 se puede observar un resumen de la topología de CARMEN, donde se indican las diferentes entradas y salidas

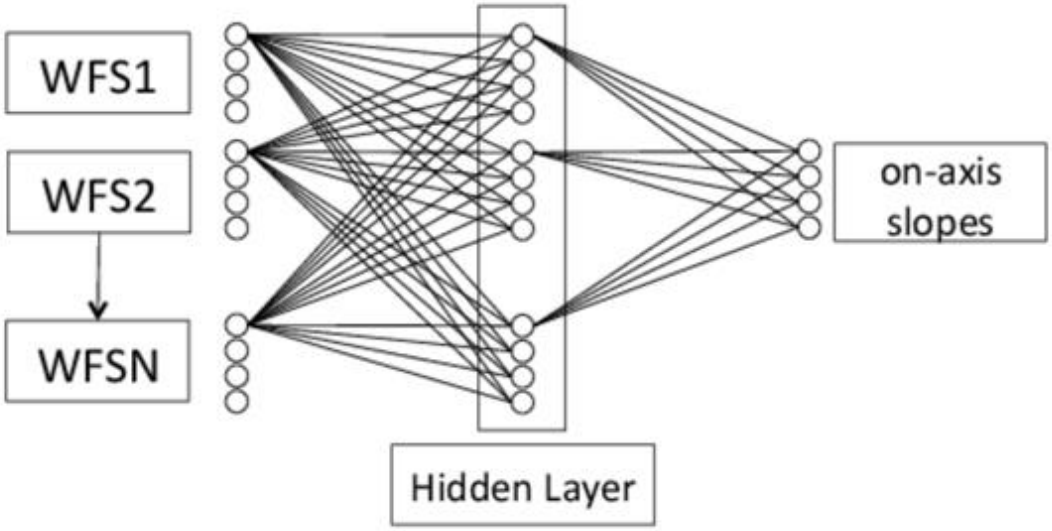


Figura 14: Topología de CARMEN

Otro de los aspectos importantes, es conocer el proceso de entrenamiento utilizado con la red. La idea detrás de este reconstructor, es la de poder ser entrenado *off-sky*, es decir, antes de ser llevado al telescopio, y una vez se tienen las matrices de pesos adecuadas, poder ser implementado en el sistema de tiempo real del telescopio.

Para ello, es necesario definir el asterismo que se quiere observar, que en los grandes telescopios suele conocerse con semanas de antelación. Una vez conocido este parámetro, hay que preparar la red en función del sistema que se utilice, ya que es necesario conocer el número de subaperturas que posee el sistema, además de las estrellas guiadas por láser disponibles. Por último, hay que estudiar si existen estrellas naturales de referencia en el campo de visión, que puedan proporcionar información extra y complementarse con las estrellas guiadas por láser. Conociendo todos estos parámetros, ya podremos saber el número de entradas y salidas que deberá poseer CARMEN.

La red necesita ser entrenada enseñándole el mayor número de combinaciones de entrada-salida posibles. Es imposible entrenar la red con el total de los escenarios a los que se va a enfrentar, ya que las combinaciones de las diferentes capas de turbulencias que pueden observar son infinitas. Sin embargo, es posible utilizar un set básico con turbulencias, que permita a la red aprender a combinarlas y realizar su propia aproximación a casos más complejos.

Para ello, se va a definir una capa de turbulencia, que va a poseer unas determinadas características, y se van a ir variando de forma aleatoria. Dicha capa, estará originalmente colocada a nivel de tierra. Una vez se ha generado suficiente

variabilidad, se irá incrementando progresivamente la altura de la turbulencia, variando en cada nueva altura los diferentes parámetros. Esto al final nos generará un set de entrenamiento, en el que tenemos una sola capa de turbulencia, pero que estará ubicada a diferentes alturas y bajo diferentes condiciones. La cantidad de datos de entrenamiento generados puede ser altamente variable, ya que cuanto más grande sea la red, mayor cantidad de datos se necesitará para entrenar. Una de las características del reconstructor, es que aunque es muy robusto ante condiciones climáticas cambiantes, necesita ser reentrenado en el caso de que se quiera observar un nuevo asterismo [5].

Uno de los cambios introducidos en el entrenamiento de CARMEN, es el método de entrenamiento utilizado. Originalmente, era un entrenamiento con un descenso de gradiente clásico, en el que los pesos de la red se actualizaban después de ejecutar cada muestra. Sin embargo, con el fin de acelerar el proceso de entrenamiento, se ha optado por un sistema de entrenamiento basado en mini-lotes o *mini-batches*, donde un conjunto de entradas se ejecutan utilizando los mismos pesos, y luego estos se actualizan de forma acumulativa [33]. Este tipo de entrenamiento no tiene una gran influencia en la calidad del entrenamiento de la red neuronal. Sin embargo, proporciona una gran mejora en los tiempos de entrenamiento, consiguiendo obtener los resultados en un tiempo mucho menor.

2.4. Introducción a la computación en GPU

Durante los últimos años, una de las soluciones más frecuentes para acelerar los cálculos de diferentes problemas, es el empleo de las unidades de procesamiento gráfico (*Graphic Processing Units o GPU*). Este tipo de tarjetas, tradicionalmente enfocadas en los videojuegos y la generación de gráficos 3D, ha supuesto un pequeño cambio en el paradigma tradicional de computación paralela.

En los siguientes apartados, se describirá brevemente el funcionamiento interno de una tarjeta gráfica reciente, y de cómo su arquitectura interna puede aprovecharse para paralelizar y acelerar enormemente ciertos cálculos. Además, se hará una breve introducción al lenguaje de programación *CUDA*, desarrollado por *Nvidia*, y diseñado para poder realizar computación de propósito general dentro de una tarjeta gráfica.

2.4.1. Modelo de Arquitectura Maxwell

Existen diferentes compañías que desarrollan procesadores gráficos (*AMD, Intel, Imagination Technologies…*), pero en el presente trabajo se va a poner especial énfasis en la compañía *Nvidia*, ya que ha desarrollado el lenguaje propietario *CUDA*, que será

parte fundamental de las explicaciones de posteriores capítulos. En este apartado, se va a hablar sobre la arquitectura *Maxwell* [34], desarrollada por *Nvidia* en el año 2014. Aunque recientemente han aparecido nuevos modelos desarrollados bajo una nueva arquitectura denominada *Pascal* [35] [36].

Un elemento clave en las tarjetas gráficas de *Nvidia*, son los denominados *CUDA Cores*. Éstos son los encargados de realizar todas las operaciones de propósito general que se envíen a la GPU. Estos núcleos guardan cierta similitud con los que podemos encontrar en cualquier CPU tradicional, aunque son considerablemente más sencillos y carecen de capacidad de hacer operaciones especialmente complejas.

Para entender el funcionamiento de cualquier GPU moderna de propósito general, hay que empezar explicando el concepto de multiprocesador adaptado al contexto en el que se está trabajando. En el esquema de la Figura 15, se pueden observar los distintos elementos que componen un *stream multiprocessor* (SM), correspondientes a la arquitectura *Maxwell*.

El elemento más abundante de la figura, son los *CUDA Cores.* En el caso de *Maxwell*, posee 128 por cada SM, que vienen agrupados en cuatro bloques de 32. Éstos, serán los principales encargados de efectuar las operaciones de propósito general que se encarguen a la tarjeta gráfica. En el caso de *Maxwell*, se han reducido prácticamente al mínimo las unidades encargadas de procesar datos en 64 bits, por lo que para realizar operaciones, estaremos limitados a trabajar con números en precisión simple. Para completar el resumen de las distintas unidades dedicadas al cálculo de datos, hay que mencionar las SFU (*Special Function Units*), que son las encargadas de efectuar cálculos especiales, como senos, logaritmos, etc. Los bloques LD/ST (*Load/Store*), se encargan de ayudar en el cálculo de las direcciones de carga y almacenamiento, aunque no efectúan ninguna operación de forma directa. Dentro de las tarjetas gráficas, existen más elementos especialmente pensados para el procesamiento de gráficos. Sin embargo, como a lo largo de este trabajo, las tarjetas son utilizadas como coprocesadores para el cálculo, dichas unidades no van a ser explicadas. Es decir, tanto las TMUs (*texture mapping units*), como las ROPs (*render output pipelines*), además de los detalles del *polymorph engine*, quedan fuera del estudio de este texto por carecer de interés.

Además de los núcleos encargados de efectuar los cálculos, son bastante importantes los llamados *Warp Scheduler*. Estos módulos, se encargan de recibir las instrucciones escritas por el programador, y repartirlas entre los distintos núcleos mediante el uso de hilos o *threads*. Estos repartidores de trabajo, cobrarán especial importancia en el siguiente apartado, cuando se pase a explicar el empleo de CUDA, y cómo se relacionan las instrucciones dadas en ese lenguaje, con la arquitectura aquí explicada.

Se puede observar también, que dentro de cada multiprocesador aparecen distintas cachés de almacenamiento. Entre ellas, destaca la memoria compartida (*Shared* Memory) o caché de nivel 1 (*L1 Cache*). Este tipo de memoria puede ser accedida de forma explícita por el programador, como se verá más adelante, pudiendo almacenar en ella datos con los que realizar cálculos posteriores. Sin embargo, otras como la caché de solo lectura (*Read-Only Data Cache*), o la de textura (*Tex*), son automáticamente gestionadas por la GPU, sin que se pueda realizar un acceso mediante código a ellas.

Una vez comprendidos los funcionamientos de los *shaders* o *CUDA Cores*, y la forma en la que se agrupan mediante multiprocesadores, es el momento de subir un nivel de abstracción, y pasar a detallar el funcionamiento de un chip completo de una tarjeta gráfica cualquiera, diseñada mediante arquitectura *Maxwell*.

Como se puede ver en la Figura 16, un chip completo GM 204 está formado por un total de 2048 *CUDA Cores*, agrupados en 16 multiprocesadores. Éstos, pese a ser relativamente independientes entre sí a la hora de realizar los cálculos, comparten una caché de nivel 2 (*L2 Cache*), a través de la cual pueden compartir datos en caso de ser necesario.

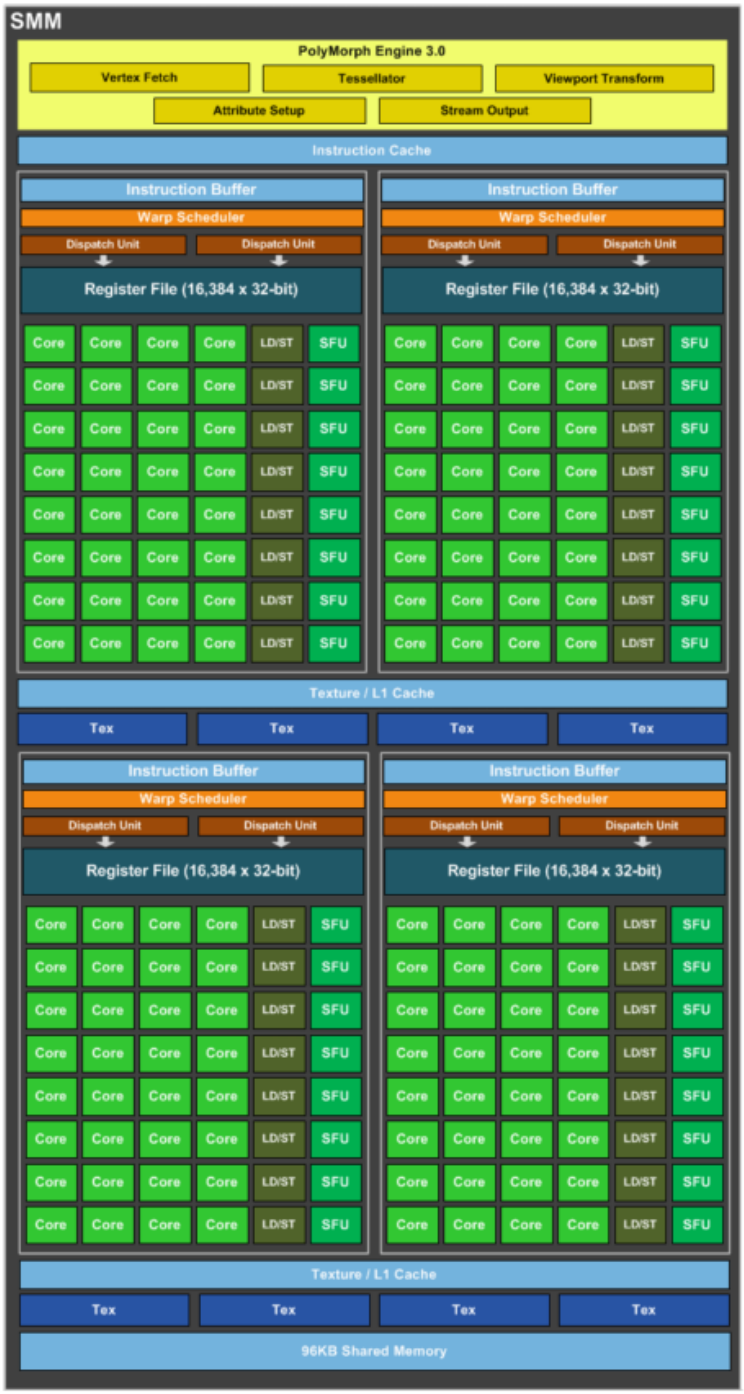


Figura 15: Esquema de multiprocesador Maxwell obtenido de [34]

Figura 16: Chip GM 204 Completo obtenido de [34]

Además de los multiprocesadores, hay que destacar las dos conexiones que comunican al chip con el exterior. Por un lado, aparece la conexión *PCI Express*, que se encarga de conectar la tarjeta gráfica, a la placa base del PC. Por otro lado aparece un controlador de memoria (*Memory Controller*), que es el encargado de gestionar las conexiones con la VRAM integrada en la propia GPU.

2.4.2. Lenguaje de Programación CUDA

El objetivo de este apartado, es introducir al lector en el lenguaje CUDA. Para ello, se darán una serie de nociones básicas sobre programación en paralelo, aprovechando los conocimientos adquiridos en el apartado anterior sobre las arquitecturas de las GPUs. A continuación, se introducirán una serie de conceptos básicos para este lenguaje, y que facilitarán la comprensión de posteriores capítulos. Por último, se realizará una breve introducción a las librerías *cuBLAS* y *cuDNN*, que serán posteriormente empleadas.

Para un programador habituado a manejar otros lenguajes como *C, C++* o *Java*, el concepto de programación en paralelo puede resultar un tanto novedoso. Los lenguajes de programación, están preparados para desarrollar las tareas de forma secuencial. Es decir, no se ejecutan dos instrucciones a la vez, sino que se espera a terminar con la instrucción actual, antes de ejecutar la siguiente. Esta idea tiene bastante sentido en los procesadores clásicos, ya que al disponer de un solo núcleo, no había forma posible de ejecutar dos instrucciones a la vez. Sin embargo, con la aparición de los primeros procesadores multi-núcleo, comienzan a surgir nuevos lenguajes que permiten ejecutar de forma simultánea varias instrucciones, enviando una a cada núcleo disponible.

Evidentemente, no todas las situaciones son susceptibles de ser programadas a través de varios núcleos. Cualquier instrucción cuya ejecución dependa del resultado obtenido en el paso anterior, no podrá ser programada en paralelo, ya que no podrá ejecutarse hasta que se tengan los datos que devuelve la instrucción anterior.

Sin embargo, existen infinidad de situaciones donde todas las operaciones son completamente independientes, como ocurre con las redes neuronales, y pueden ejecutarse completamente en paralelo, aprovechando al máximo todos los núcleos disponibles. Mientras que las CPUs actuales más potentes no superan los 36 núcleos, se puede encontrar en el mercado tarjetas gráficas con más de 3000 núcleos. Gracias a toda esta potencia, es posible conseguir que cada *core* ejecute una operación, obteniendo enormes ganancias en tiempo, si se compara con la tradicional programación secuencial.

El primer concepto importante que hay que entender dentro de la programación en paralelo, es el de *thread* o hilo. Se denomina como *thread*, al conjunto de instrucciones que son enviadas a un determinado *core* en un momento determinado. En el caso de CUDA, cada hilo se encargará de gestionar las operaciones que realiza cada *CUDA Core*, encargándose de devolver los correspondientes resultados al finalizar las operaciones. Con el fin de simplificar su manejo, los hilos se agrupan en bloques, dentro de los cuales se asigna un identificador a cada *thread*, como se puede ver en la Figura 17. Además, los bloques son agrupados de forma similar dentro de un elemento denominado *grid*, que estará compuesto por el número total de hilos que se pretenden ejecutar de forma simultánea en la GPU. Aunque en la Figura 17 los índices sugieran que los distintos elementos están ordenados siguiendo una matriz de dos dimensiones, este orden puede ser cambiado a elección del programador, pudiendo agruparse tanto en 1, 2 o 3 dimensiones.

Otro elemento destacable dentro de CUDA, son los denominados *kernels*. Éstos, son los encargados de ejecutar los cálculos en paralelo, encargándose de gestionar un *grid*, con el tamaño y las dimensiones que le hayan sido indicadas.

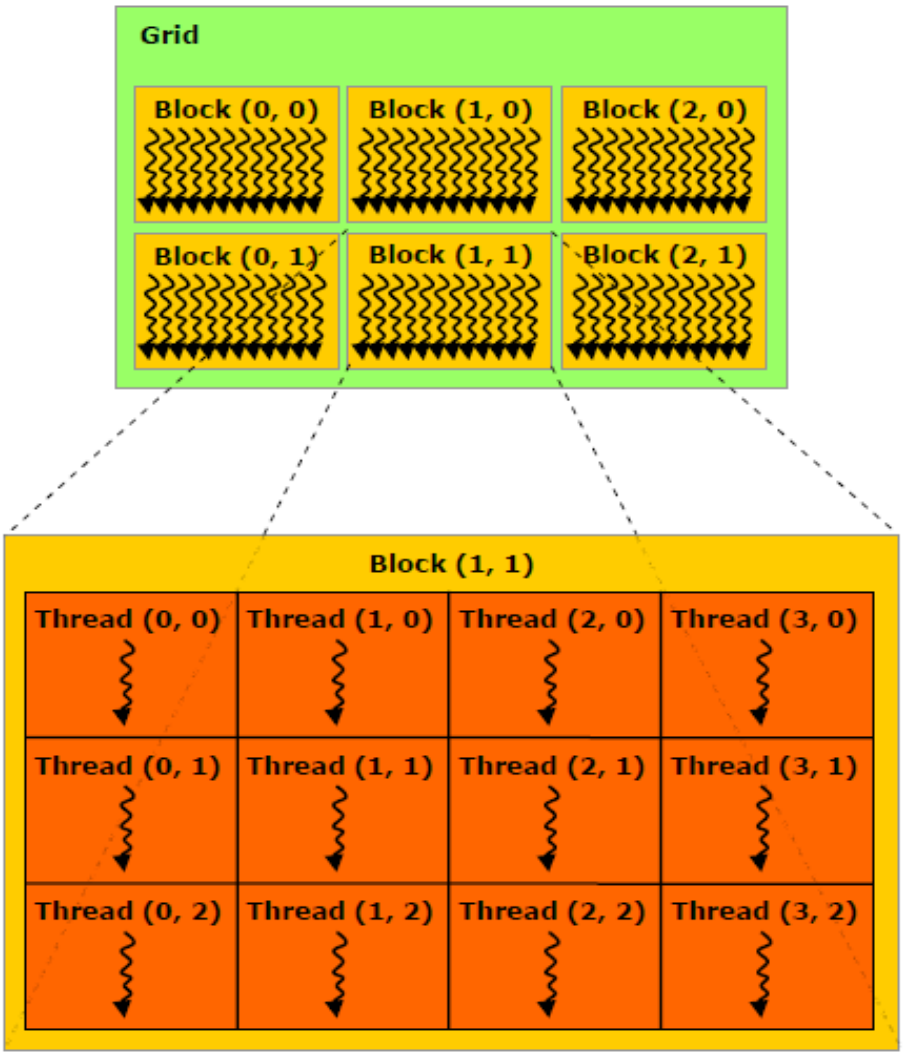


Figura 17: Orden de hilos en GPU obtenido de [37]

Como se puede ver en la Figura 18, la ejecución de un *kernel* cambia del modelo de un hilo único a través del cual se desarrolla el código en la CPU, por la ejecución en paralelo de múltiples hilos sobre la GPU. La llamada a estos *kernels*, se realiza cuando es necesario descargar de tareas pesadas a la CPU, pudiendo utilizar la GPU como procesador secundario para realizar distintos cálculos. Como se ha explicado anteriormente, estas operaciones necesitan ser paralelizables, es decir, que no exista una fuerte dependencia entre ellas, y puedan ser realizadas de forma simultánea. Es importante mencionar, que el empleo de los *kernels*, se limita principalmente, al cálculo de operaciones. Una particularidad dentro de estas operaciones, es que no se garantiza el orden en el que son ejecutadas, sino que solamente se sabe que al finalizar su ejecución, todos los cálculos habrán sido realizados. Todas las tareas relacionadas con las reservas o copias a memoria o la preparación de los datos, se realizan fuera del propio *kernel*.

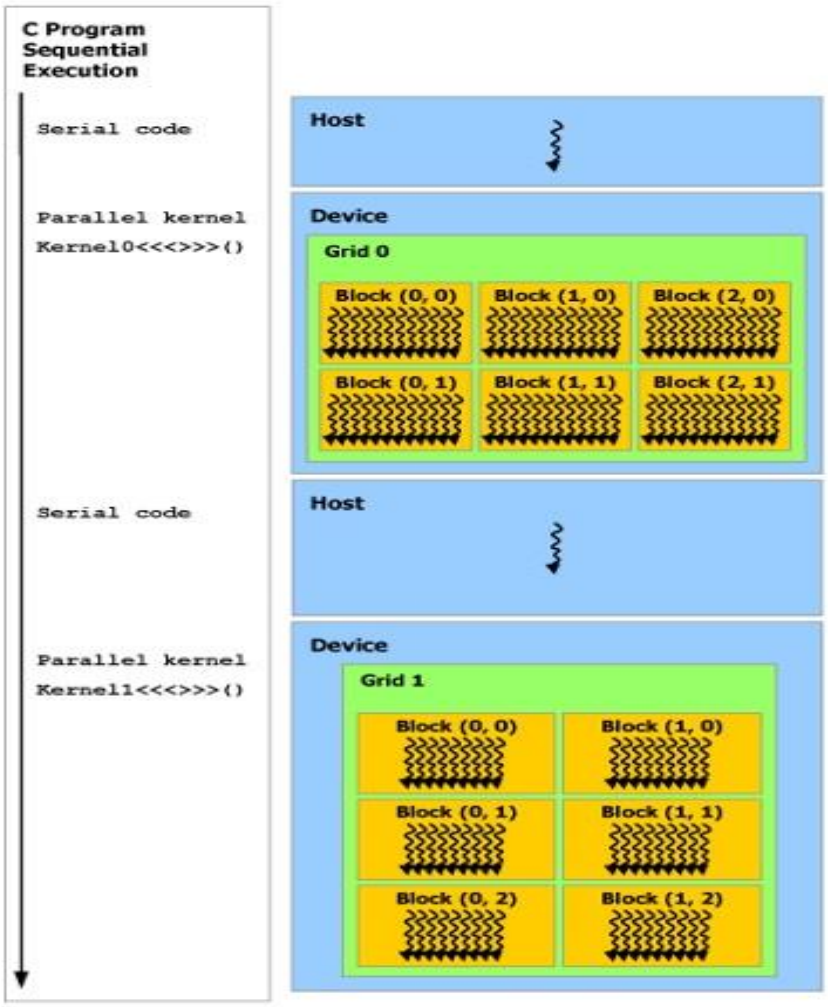


Figura 18: Empleo de Kernels obtenido de [37]

Tras conocer las ideas básicas sobre el concepto de *kernel*, es necesario familiarizarse ahora con la forma en la que se reparte el trabajo entre los distintos multiprocesadores. Una simplificación de este reparto, aparece en la Figura 19, en la que los bloques se van repartiendo de forma ordenada entre los distintos multiprocesadores.

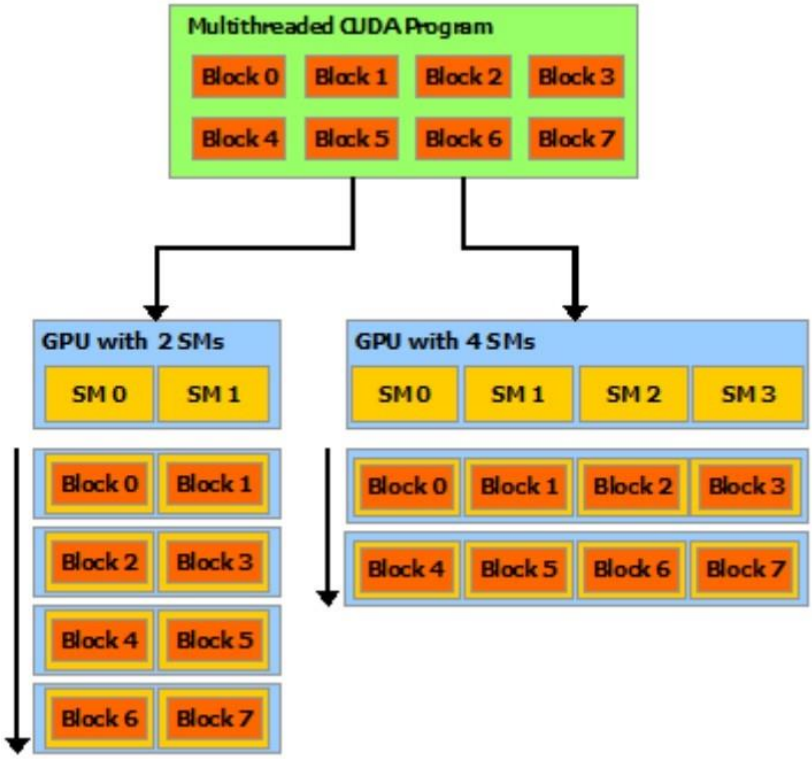


Figura 19: Reparto de trabajo entre SMs obtenido de [37]

Además de todas las herramientas necesarias para programar los cálculos a bajo nivel, existen numerosas librerías preparadas por *Nvidia*, que sirven para acelerar cálculos comunes, sin necesidad de que el programador se encargue de paralelizar de forma directa las tareas. La lista de librerías es considerablemente extensa, y puede ser consultada en [38], aunque para este trabajo se han utilizado únicamente dos de ellas.

Debido a la forma en la que las redes neuronales pueden ser ejecutadas y entrenadas, es fácil encontrar cómo es necesario realizar una gran cantidad de multiplicaciones de matrices. Por definición, esta es una de las operaciones que más alto grado de paralelismo posee, y puede ser acelerada enormemente mediante el uso de una GPU. Para ello, se hará uso de la librería *cuBLAS*, que permite realizar de forma sencilla operaciones con matrices y vectores [39]. El empleo de esta solución, permite acelerar enormemente el cálculo de las distintas multiplicaciones, ya que posee optimizaciones de bajo nivel, que requerirían una gran cantidad de trabajo si tuviesen que ser programadas de nuevo.

La otra librería empleada es notablemente más reciente, y ha sido creada aprovechando el gran impacto que están teniendo las redes neuronales durante los últimos años. Con *cuDNN* [40] se tiene una librería que facilita el cálculo de los gradientes para el entrenamiento de las redes, las funciones de activación, etc. Además, está pensada para ser complementada con otras librerías como *cuBLAS* para acelerar de forma completa el proceso de ejecución y entrenamiento de cada red.

2.5. Frameworks de Redes Neuronales

Como se ha explicado en apartados anteriores, durante los últimos años las redes neuronales han tenido un gran impacto en diferentes campos. Además, debido a la aparición de redes cada vez más grandes y más complejas, se ha necesitado un esfuerzo cada vez mayor para poder ejecutar y entrenar dichas redes, tanto a nivel de programación, como en cuanto al hardware requerido para ello.

Para solucionar estos problemas, son numerosas las universidades y empresas que han compartido sus soluciones, de forma que otros grupos puedan aprovechar parte de su trabajo para crear una nueva red que se adapte a su investigación. En [41] podemos observar una extensa lista con diferentes marcos o *frameworks* empleados actualmente en el campo de las redes neuronales.

Todas estas soluciones, buscan facilitar a los usuarios, la creación de su propia red, intentando reducir al mínimo el esfuerzo necesario para programarla. Además, es muy habitual que los diferentes *frameworks* ofrezcan la posibilidad de acelerar los diferentes cálculos de las redes, utilizando lenguajes de programación paralela, como *OpenMP* o *CUDA*.

El primer reto que se plantea, es elegir la solución más adecuada para nuestro problema. Todos los sistemas tienen sus ventajas e inconvenientes, y continuamente se están añadiendo nuevas características, o incluso *frameworks* nuevos. En una búsqueda inicial, encontramos que desde la página de *Nvidia*, se hacía especial énfasis en tres de los sistemas de la lista: *Caffe, Torch* y *Theano*. Los desarrolladores de estos tres *frameworks* habían colaborado estrechamente con *Nvidia* en el desarrollo de la librería *cuDNN*, además, de ser tres de las soluciones más populares entre la comunidad [42].

2.5.1. Caffe

Caffe es un *framework* de aprendizaje profundo, desarrollado por el Centro de Visión y Aprendizaje de Berkeley, que ha sido lanzado bajo una licencia de tipo BSD 2-Clause [43]. Está principalmente desarrollado bajo C/C++, y usa CUDA para acelerar la ejecución en la GPU, además de poseer soporte para la librería *cuDNN.* Posee interfaces para *Matlab*, *Python*, e intérprete en línea de comandos, que será el que se vaya a utilizar en el presente trabajo.

Una de las principales características de *Caffe*, es su facilidad para diseñar, entrenar y ejecutar una red neuronal. No es necesario poseer conocimiento de programación alguno, ya que el diseño de la topología de la red y el ajuste de los parámetros, se realiza a través de un fichero de texto. Únicamente es utilizar una de las

plantillas que proporciona, para indicar al *framework* el número de capas que se desean, el tipo, el número de neuronas que va a tener, o el resto de características que se quieran. Además, en otro fichero, se le pasará información sobre los valores de entrenamiento, como la tasa de aprendizaje, el algoritmo que se quiera utilizar, o el número de *epochs* que se desea realizar. Una vez se ha configurado todo, basta con hacer una llamada al programa desde línea de comandos para que la red empiece su entrenamiento.

Pese a ser el sistema más sencillo de utilizar de los diferentes *frameworks* analizados, posee el problema de que no es fácilmente integrable con otros programas más grandes. Aunque el código de *caffe* está disponible para poder trabajar de forma directa con él, la interfaz que proporciona no es la más adecuada para que pueda interactuar de forma directa con otros lenguajes.

2.5.2. Torch

Torch es un *framework* de computación científica con soporte para algoritmos de aprendizaje máquina. Está escrito en *Lua* y C/CUDA, para proporcionar una rápida ejecución, además de que posee la posibilidad de importar módulos externos para completar y acelerar el sistema. Es principalmente utilizado y mantenido por importantes compañías como *Google, Facebook, Twitter*, etc.

A diferencia de *Caffe*, aquí si es necesario tener una serie de conocimientos básicos de programación, aunque pueden ser aprendidos con facilidad. Al estar basado en *Lua*, será necesario crear una serie de pequeños *scripts*, donde se vaya indicando el proceso que se quiere seguir. En este caso es necesario gestionar la carga de datos, dónde queremos que se ejecuten (si en la CPU principal o en la GPU), o de cuándo y cómo guardar el modelo que se ha entrenado. Sin embargo, la creación, el entrenamiento y la ejecución de las redes es notablemente simple, ya que gracias a los diferentes paquetes existentes, bastará con indicar el tipo de capa que se quiere emplear, irlas anidando, para posteriormente poder ejecutarlas, y si se quiere también calcular el error, retropropagarlo y actualizar los pesos.

En este caso, nos encontramos con que *Torch* proporciona un equilibrio interesante entre dificultad de programación y flexibilidad. Es cierto que las capas que se pueden utilizar, o los algoritmos de retropropagación empleados vienen encapsulados dentro de un paquete (aunque disponemos del código fuente y podríamos modificarlo), pero al utilizar *scripts* y disponer de un lenguaje de programación completo, podemos realizar tareas más complejas que en el caso de *Caffe.*

2.5.3. Theano

Theano es una librería para *Python* desarrollada para ayudar a los investigadores en sus investigaciones sobre aprendizaje profundo [44]. Está siendo desarrollado en la Universidad de Montreal, y utiliza algunas de las librerías más comunes de *Python*, como *numpy* o *scipy*. Proporciona también la posibilidad de utilizar la GPU para acelerar los procesos de entrenamiento y ejecución.

A diferencia de los dos anteriores, aquí es altamente recomendable tener unos buenos conocimientos de *Python*, ya que será necesario programar gran parte del funcionamiento de la red en este lenguaje. Con la importación de *Theano* tendremos facilidad a la hora de computar el gradiente o de retropropagar el error a través de las diferentes capas, pero será necesario que nosotros las diseñemos y las programemos.

La ventaja que esto proporciona, es que es posible modificar a nuestro antojo las diferentes capas, no estando limitados a los tipos más habituales. Además, es posible diseñar nuestras propias funciones de error, o incluso realizar ajustes de más bajo nivel en la red, que no son posibles en el resto de *frameworks*.

2.5.4. Implementación en CUDA

En este caso, no vamos a disponer de un *framework* que nos proporcione todas las características necesarias para crear una red neuronal, sino que se va a crear un pequeño código, donde se programe dicha red utilizando directamente CUDA, y algunas de las librerías proporcionadas por *Nvidia*. El objetivo de esta implementación, es ser lo más ligero y rápido posible, evitando cualquier tipo de operación innecesaria, y adaptando los diferentes valores para que proporcionen un rendimiento óptimo con un perceptrón multicapa como CARMEN.

Para ello, ha sido necesario realizar un análisis sobre el funcionamiento de una red neuronal, que ha sido explicado en 2.2.2, y posteriormente decidir cuál sería la forma más adecuada de paralelizarla y acelerarla lo máximo posible. Una de las primeras observaciones, es que tanto la ejecución como el entrenamiento de las redes, puede agruparse en forma de operaciones con vectores y matrices. La forma más rápida de realizar dichas operaciones, es hacer uso de la librería *cuBLAS*, que proporciona un rendimiento muy elevado, superior incluso al que se podría obtener creando nuestro propio código para trabajar con matrices. También tenemos que *Nvidia* proporciona la librería *cuDNN*, que facilita el cálculo de las funciones de activación, y que puede ahorrar una parte del trabajo de desarrollo.

Con estas ideas en mente, se ha desarrollado un código en el que se realiza una copia asíncrona de los datos de entrenamiento de la RAM a la VRAM. A continuación, se

obtiene las salidas, utilizando *cuBLAS* para multiplicar las entradas por las matrices de pesos, y *cuDNN* para obtener los resultados de las funciones de activación. Con este código, ya tendríamos la parte de ejecución de la red, que nos permitirá comparar la velocidad con la que podemos obtener cada una de las salidas.

Para entrenar la red, ha sido necesario crear una serie de *kernels*, que calculen el error obtenido. Estos valores se retropropagan a lo largo de la red, utilizando diferentes llamadas tanto a la librería *cuBLAS* para realizar las operaciones con matrices, como a los distintos *kernels* que ajustan los errores a lo largo de las distintas capas. En el caso del entrenamiento, mientras se actualizan los pesos de las matrices, se empieza a realizar la copia del siguiente lote que se utilizará para entrenar, consiguiendo así que el sistema no tenga que esperar a copiar los datos nuevos antes de empezar a entrenar con un nuevo lote.

Capítulo 3: Descripción del Problema

En el capítulo anterior, se ha hecho una introducción sobre una serie de aspectos que van a ser cruciales para este trabajo final de máster. Se ha realizado una introducción al funcionamiento de los telescopios de gran tamaño, poniendo especial énfasis en la óptica adaptativa. Además, se ha explicado el funcionamiento de las redes neuronales, poniéndolas en su contexto histórico, y detallando de forma matemática su funcionamiento.

La unión de estos dos elementos, dio lugar al reconstructor tomográfico CARMEN, que se encarga de mejorar la calidad de las imágenes que se obtienen en cualquier ELT. Tras diferentes pruebas, tanto en laboratorio como en telescopios reales, se comprobó que era necesario acelerar el funcionamiento de dicho sistema, ya que su implementación original no estaba pensada para funcionar en telescopios con enormes cantidades de entradas.

Para realizar esta mejora, se ha optado por utilizar las unidades de procesamiento gráfico para acelerar el proceso. En los apartados anteriores, se ha comprobado cómo el empleo de GPUs en la aceleración del entrenamiento y la ejecución de redes neuronales, es una de las soluciones más comunes.

Sin embargo, tras optar por una solución basada en GPUs, surge un nuevo problema. ¿Cuál es el *framework* que más se adecua a nuestras necesidades? Tras una búsqueda inicial, se optó por emplear los tres sistemas explicados anteriormente. Además, se consideró que debido a que la red empleada en CARMEN no era especialmente compleja, sería especialmente interesante desarrollar un código escrito directamente en CUDA, y comparar su rendimiento con el de los diferentes *frameworks*.

3.1. Aproximación Metodológica

Uno de los parámetros claves en la definición del problema, es decidir cómo se van a comparar los diferentes sistemas entre sí. Existen una gran cantidad de variables que podrían ser tenidas en cuenta a la hora de elegir un *framework* u otro, como su complejidad, su flexibilidad o lo integrables que pudieran ser en un sistema más grande. Aunque todos estos elementos son importantes, y podrán ser tenidos en cuenta en el futuro, no serán objeto de evaluación y comparación en este trabajo.

Para el presente trabajo, se va a utilizar una aproximación puramente cuantitativa. Para ello, los valores que se van a comparar, son los tiempos de entrenamiento y de ejecución que cada *framework* nos ofrece. Este valor es completamente objetivo, ya que únicamente depende de las condiciones utilizadas, que pueden ser perfectamente reproducidas en un entorno de condiciones controladas.

El objetivo de esta aproximación, es saber cuál es el sistema que con más rapidez nos podrá devolver los resultados, tanto en el caso de entrenamiento, como en el de la ejecución. Por un lado, es muy importante conocer cuál es el *framework* que más rápido entrena, ya que cuando las redes crezcan de forma sustancial, el ahorro que pueda suponer utilizar una propuesta u otra, puede ser muy sustancial. Además, conseguir entrenar lo más rápido posible, puede acercar la posibilidad de hacer un entrenamiento *on-line*, una vez que la red esté implementada en el telescopio, y esta pueda ser adaptada a las condiciones de la observación, en lugar de ser entrenada únicamente con un set de datos predefinido.

Es importante también, que a la hora de llevar el reconstructor a un sistema de tiempo real controlar los tiempos de ejecución. Los sistemas de tiempo real ofrecen una imagen nueva en márgenes de unidades de milisegundos, ya que las turbulencias atmosféricas cambian a grandes velocidades. Por ello, es necesario corregir la deformación del espejo a la mayor velocidad posible, para que éste se adapte a la deformación que se está observando. Cualquier mínima mejora en los tiempos de ejecución puede suponer una gran diferencia, ya que el espejo se estará adaptando mejor a la información que le ha proporcionado el sistema en tiempo real, en lugar de estar corrigiendo con información previa.

Para poder medir tanto los tiempos de entrenamiento y ejecución, se va a diseñar una serie de experimentos, en los que se comprobará bajo condiciones controladas, los tiempos de entrenamiento y de ejecución que ofrecen los diferentes *frameworks*, además del código desarrollado directamente en *CUDA*. Para ello, se empleará información de los sistemas de tiempo real explicados en capítulos anteriores, y se

realizará un análisis de cómo influye la variación del tamaño de la red en los diferentes *frameworks*. Esto permitirá elegir la solución más rápida en función del tamaño del telescopio, además de poder entrenar con el más rápido, lo que nos permitirá obtener y probar nuevos experimentos con mayor velocidad

3.2. Descripción del Experimento

Se van a diseñar dos experimentos distintos para medir tanto los tiempos de ejecución, como los tiempos de entrenamiento. Cada uno de ellos tiene características propias, por lo que hace que distintos parámetros tengan que ser tenidos en cuenta a la hora de realizar el diseño.

3.2.1. Tiempos de Entrenamiento

Existe una gran cantidad de parámetros que es necesario ajustar cuando se entrena una red neuronal, aunque solo unos pocos de ellos tienen una influencia directa en los tiempos de entrenamiento. Con el fin de simplificar las comparaciones, únicamente se variarán los parámetros que se consideren relevantes, que serán detallados y justificados a continuación.

Uno de las características más comunes en el entrenamiento de las redes neuronales, es el uso del algoritmo de retropropagación como método de entrenamiento. En este caso, como se ha mencionado anteriormente, se va a utilizar una variante del descenso de gradiente tradicional, que sería el descenso de gradiente estocástico con *momentum* mediante mini-lotes. Este método, se encuentra implementado en los tres *frameworks* explicados anteriormente, además de en el código *CUDA* desarrollado. En este tipo de entrenamiento, existen dos variables que es necesario ajustar para obtener los mejores resultados posibles: los valores del *momentum* y la tasa de aprendizaje. Aunque estas variables son cruciales para un correcto entrenamiento de la red, no tienen influencia alguna en el tiempo de entrenamiento. Dado que todos los elementos que vamos a comparar emplean el mismo método, se puede asumir que si todos ellos usan los mismos valores de tasa de aprendizaje y *momentum,* obtendremos en todos ellos los mismos valores para las matrices de pesos. Por ello, se va a asumir que ambos valores han sido perfectamente optimizados, y no van a variarse a lo largo de todo el experimento, ya que no van a tener influencia en la calidad o los tiempos del entrenamiento.

Tampoco es objetivo de este trabajo, analizar la calidad de los resultados obtenidos por las distintas redes. Como se ha mencionado anteriormente, se puede suponer que al utilizar el mismo método de entrenamiento con los mismos parámetros,

y los mismos valores de entrenamiento, los diferentes *frameworks* van a obtener los mismos resultados. Esto hace que tampoco se vaya a incluir en el experimento, información sobre los errores que obtienen las redes después de entrenarlas, ya que se puede asumir que serán prácticamente iguales.

Otro parámetro crítico, es el tamaño del conjunto de datos de entrenamiento. Sin embargo, es fácil de comprobar que en el método utilizado basado en mini-lotes, el tiempo crece de forma proporcional al número de datos utilizado. Esto hace, que sea fácil de calcular el tiempo de entrenamiento de la red aunque cambie el tamaño del set de datos, ya que ambos cambiarán de la misma forma. Esto hace que la cantidad de datos empleados en el entrenamiento, sea irrelevante a la hora de comparar los resultados, aunque dicha cantidad es extremadamente importante a la hora de conseguir que la red entrene correctamente. En este caso, la cantidad de datos empleada puede verse en la Tabla 3.

Para los diferentes experimentos, se van a emplear tres tamaños diferentes de redes neuronales, una por cada uno de los sistemas de tiempo real explicados previamente.

En el caso de CANARY Fase-B1, los datos con los que se ha realizado el experimento, se corresponden con una observación en la que había una estrella guiada por láser, y dos estrellas naturales, lo que nos deja un total de 216 valores para la entrada de la red. Para CANARY Fase-C2, los datos se corresponden a una observación con 4 estrellas guiadas por láser, lo que da un total de 1152 entradas. En ambos casos, los datos utilizados para entrenar, se corresponden con los valores obtenidos en el simulador CANARY.

La situación de DRAGON es ligeramente diferente. Como se ha explicado anteriormente, este es aún un sistema en desarrollo, por lo que no se puede obtener datos de simulación de una situación real. Es por ello, que se va a asumir que los datos de entrada se corresponden con la observación de un asterismo que no posee estrellas naturales en el campo de visión, pero que sí tendremos 4 estrellas guiadas por láser como información de entrada. Esto hace, que tengamos 7200 valores de entrada para este sistema. En la Tabla 3 se puede ver un resumen con el tamaño de las redes en función del sistema de tiempo real empleado.

Nombre	Tamaño de Red	Número de muestras de entrenamiento
CANARY-B1	216-216-72	350.000
CANARY-C2	1152-1152-288	1.500.000
DRAGON	7200-7200-1800	1.000.000

Tabla 3: Resumen de los sistemas de AO y tamaño de las redes

Además del tamaño de la red, se va a variar también el tamaño de los mini-lotes, para estudiar cómo afecta el cambio a los tiempos de entrenamiento de la red. El tamaño de los lotes tiene un alto impacto en la posibilidad de paralelizar las operaciones, por lo que se ha considerado especialmente relevante conocer cómo afecta a las diferentes redes mencionadas. Se ha optado por estudiar los tamaños más comunes, que han sido empleados en otros estudios [42]. Estos valores serían de 16, 32, 64, 128 y 256.

Una vez definidos todos los parámetros, es el momento de detallar cómo se realizará el proceso de medición de los tiempos. En primer lugar, se van a inicializar los pesos, quedando almacenados en la memoria VRAM de la GPU[1]. También se cargarán desde el disco duro, todo el set de datos de entrenamiento, que se almacenará en la RAM principal del sistema. Una vez estos datos estén cargados, se arrancará un temporizador, y se iniciará un bucle. En él, se realizará la copia de los mini-lotes desde la RAM hacia la VRAM, se obtendrá la salida de la red, se calculará el error y se actualizarán los pesos. Este bucle recorrerá el set de datos de entrenamiento completo. Una vez haya terminado, se detendrá el temporizador, por lo que obtendremos el tiempo que cada *framework* tarda en entrenar durante un *epoch*. Esta operación se volverá a repetir durante 20 veces. Esto nos permitirá promediar los tiempos de entrenamiento entre los diferentes *epochs*, haciendo el resultado más fiable. Además, se podrá comprobar si existe una gran diferencia entre los diferentes tiempos de entrenamiento, o estos permanecen relativamente constantes con el tiempo.

3.2.2. Tiempos de Ejecución

En el caso de los tiempos de ejecución, se van a emplear las mismas redes definidas en el apartado anterior, y resumidas en la Tabla 3. Sin embargo, existen una serie de diferencias con el proceso de medición de los tiempos de entrenamiento, que necesitan ser detalladas.

En primer lugar, aquí la red es alimentada utilizando una única entrada, en lugar de emplear mini-lotes. Esto pretende simular el comportamiento que debería tener la red cuando se llevase a un sistema de tiempo real de un telescopio, donde las entradas se van proporcionando de forma secuencial, y espaciadas entre cortos periodos de tiempo. Al simular esta configuración, se puede asumir que todas las variables ya han sido inicializadas, y que los pesos están ya cargados en la VRAM, por lo que únicamente estaremos esperando a tener una salida para poder ejecutar la red, sin necesidad de realizar más operaciones.

[1] De aquí en adelante, nos referiremos a VRAM como la RAM que posee la GPU, y utilizaremos el acrónimo RAM para referirnos a la RAM principal del sistema.

Sin embargo, no podemos estar completamente seguros de que la entrada que proporcione el telescopio, vaya a ser cargada a la memoria RAM del sistema, de tal forma que podamos acceder a ella desde el código de nuestro programa. Por ello, será necesario diferenciar dos casos: Por un lado, se va a asumir el escenario ideal, donde la entrada se encuentra ya cargada en la RAM, y desde cualquiera de los *frameworks* podemos acceder directamente a ella, mientras que también se estudiará el caso, en el que tengamos que cargar dicha entrada desde el disco duro. En el primer escenario, se empezará a contabilizar el tiempo una vez que la entrada ya se encuentre en la RAM, y se detendrá una vez que volvamos a copiar la salida desde la VRAM. En el segundo caso, se tendrá también en cuenta el tiempo necesario para copiar los datos desde el disco duro hasta la RAM en el caso de la entrada, y la copia inversa en el caso de la salida.

Puesto que las pruebas no han podido realizarse en un telescopio real, se ha optado por almacenar cada una de las entradas en un fichero h5 [45], que es compatible con todos los *frameworks* utilizados, con el fin de simular lo que nos encontraremos en un sistema de control de tiempo real. Esto permite que al tratar cada una de las entradas por separado, se pueda medir con facilidad tanto el proceso completo, como únicamente el tiempo ejecución una vez que la entrada se ha cargado en la RAM.

Al igual que en el caso del entrenamiento, se va a alimentar a la red con una gran cantidad de entradas -10.000 en este caso-, que se irán ejecutando de una en una. Esto permitirá promediar los tiempos obtenidos, además de evitar fallos en las mediciones, ya que es habitual que las primeras ejecuciones sean más lentas de lo esperado, puesto que no todas las instrucciones están aún almacenadas en la memoria caché de GPU. Como se ha mencionado anteriormente, esto no supondrá un problema cuando se lleve el sistema a los telescopios, ya que la cantidad de entradas que se ejecutarán será considerablemente más grande.

A la hora de realizar las mediciones, nos hemos encontrado con dos problemas. En el caso de *Theano*, no ha sido posible medir el caso en el que los datos se encuentran ya cargados en la RAM, ya que el *framework* realiza la copia de forma interna, y no ha sido posible diferenciar en qué momento del proceso la realiza. Además, como se mencionó en los apartados introductorios, *Caffe* no permite una fácil integración de su código con otros sistemas. Por ello, ha sido imposible realizar las mediciones utilizando un conjunto de ficheros separados, ya que no permite dejar todos los datos inicializados, e ir alimentando a la red con entradas de diferentes archivos sin volver a cargar toda la información de nuevo.

3.2.3. Equipamiento empleado

Para realizar las diferentes mediciones, se ha utilizado un servidor que posee un procesador *Intel Xeon CPU* E5-1650 v3 @3.50 *Ghz*, con 128*Gb* de memoria RAM DDR4. La tarjeta gráfica empleada ha sido una *Nvidia Geforce GTX TitanX*, además de poseer un SSD como sistema de almacenamiento principal.

En la parte *software*, el equipo funcionaba bajo el sistema operativo Ubuntu 14.04.3 LTS, con la versión 7.5 de CUDA, junto con la librería cuDNN v3. En el caso de los diferentes *frameworks,* la versión de *Theano* utilizada ha sido la v0.7, y en los casos de *Caffe* y *Torch* se ha empleado la versión disponible en sus repositorios a fecha de 28 de febrero de 2016.

Capítulo 4: Análisis de Resultados

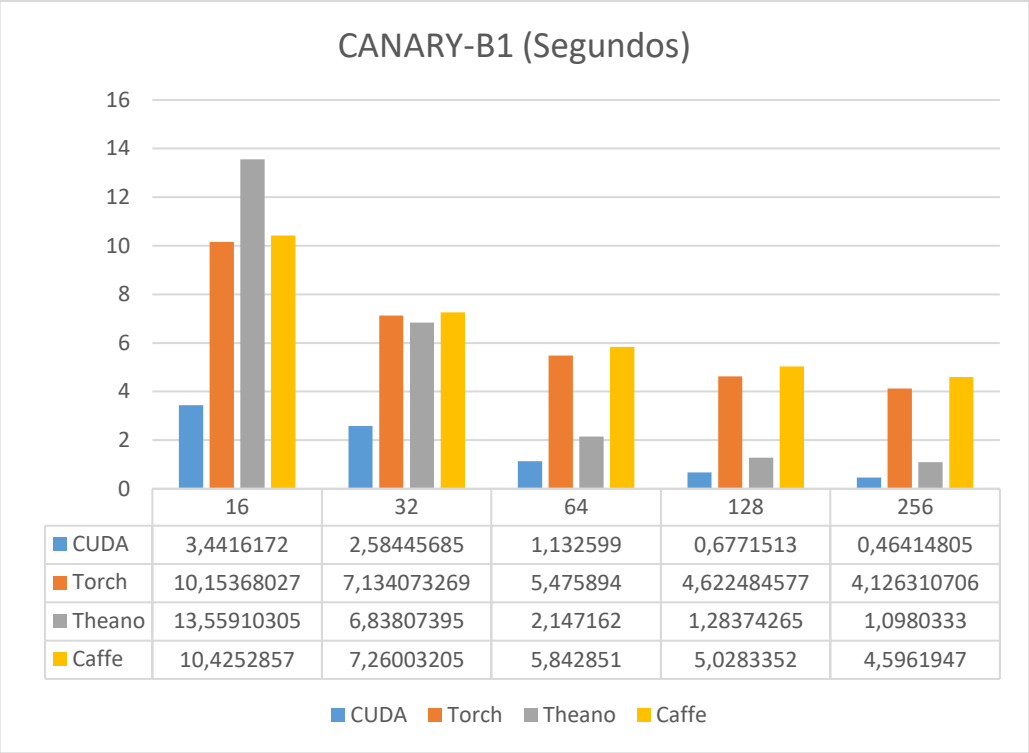
A lo largo del presente capítulo, se va a proceder a presentar y analizar los resultados obtenidos en los diferentes experimentos. Para ello, se van a dividir el capítulo entre los diferentes sistemas de tiempo real utilizados, donde se expondrá y se analizará de forma separada los diferentes resultados obtenidos.

4.1. CANARY Fase B1

Es el sistema más pequeño de todos los presentados. Como se ha detallado anteriormente, se ha utilizado una red neuronal de 216-216-72 neuronas, y en este caso se han utilizado un total de 350.000 valores para el entrenamiento.

Como se puede ver en la Figura 20, aumentar el tamaño del lote, proporciona una clara mejoría en los tiempos de entrenamiento. Sin embargo, esto afecta de diferentes formas a cada uno de los diferentes sistemas.

Para todos los tamaños de lote utilizados, el código escrito directamente en C/CUDA es el que mejores resultados proporciona, siendo incluso en el peor de los casos dos veces más rápido que la siguiente opción. En el caso de *Theano,* se tiene que para valores de lotes grandes, obtiene grandes resultados, siendo el segundo más rápido con diferencia, aunque al disminuir el tamaño del lote, hace que sus valores se disparen. Los casos de *Torch* y *Caffe* son bastante similares. Aunque para tamaños de lote pequeños obtienen resultados similares a *Theano*, luego no tienen la capacidad de reducir los tiempos de entrenamiento cuando se aumenta el tamaño del lote, siendo mucho peores que CUDA para lotes especialmente grandes.



	16	32	64	128	256
CUDA	3,4416172	2,58445685	1,132599	0,6771513	0,46414805
Torch	10,15368027	7,134073269	5,475894	4,622484577	4,126310706
Theano	13,55910305	6,83807395	2,147162	1,28374265	1,0980333
Caffe	10,4252857	7,26003205	5,842851	5,0283352	4,5961947

Figura 20: Tiempos de entrenamiento por epoch para CANARY Fase B1

En el caso de los tiempos de ejecución, los resultados obtenidos son ligeramente diferentes, tal y como se puede ver en la Figura 21. En este caso, se vuelve a tener que el código escrito en CUDA obtiene los mejores resultados de nuevo, siendo más de dos veces más rápido que la siguiente alternativa. Sin embargo, en el caso de *Torch*, ahora es bastante más rápido que *Theano*, aunque aún se queda lejos de los mejores resultados obtenidos.

Es interesante observar cómo en este caso, la diferencia entre cargar los datos desde el disco duro, o tenerlos ya cargados en la RAM, supone un cambio muy importante. Podemos comprobar cómo la diferencia entre un caso y otro, hace que se tarde hasta cinco veces más en obtener una salida desde que se ha recibido la entrada.

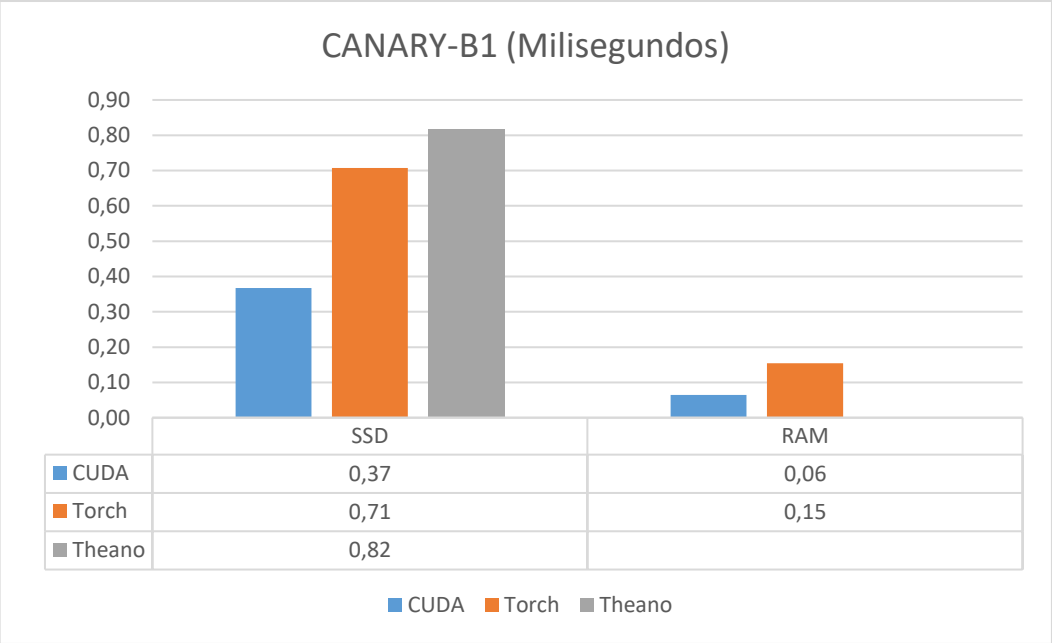


Figura 21: Tiempos de Ejecución para CANARY Fase B1

4.2. CANARY Fase – C2

En el caso de CANARY Fase – C2, se tiene una red con 1.152-1.152-288 neuronas, y se van a emplear 1.500.000 de datos para el entrenamiento. En esta ocasión, la reducción del tamaño de los lotes, tiene un impacto aún mayor aumentando el tiempo de entrenamiento, debido a que la red es bastante más grande.

Para este sistema, el código escrito en CUDA vuelve a ser de nuevo el más rápido en todos los casos, tal y como se puede ver en la Figura 22. En este caso, es hasta dos veces más rápido que el siguiente *framework*, aunque las diferencias no son tan grandes como en el caso anterior. Para este tamaño de red, *Caffe* demuestra un buen rendimiento, especialmente en el caso de los lotes pequeños, aunque no es capaz de reducir su tiempo de entrenamiento al crecer el tamaño del lote. En el caso de *Theano*, ocurre lo mismo que con CANARY Fase B1, donde mostraba un gran rendimiento para lotes grandes, pero sus tiempos crecían de forma notable en los lotes más pequeños. Por último, *Torch* se muestra como la peor solución de todas las estudiadas, aunque en casi todos los casos se muestra en valores bastante cercanos a los proporcionados con *Caffe*.

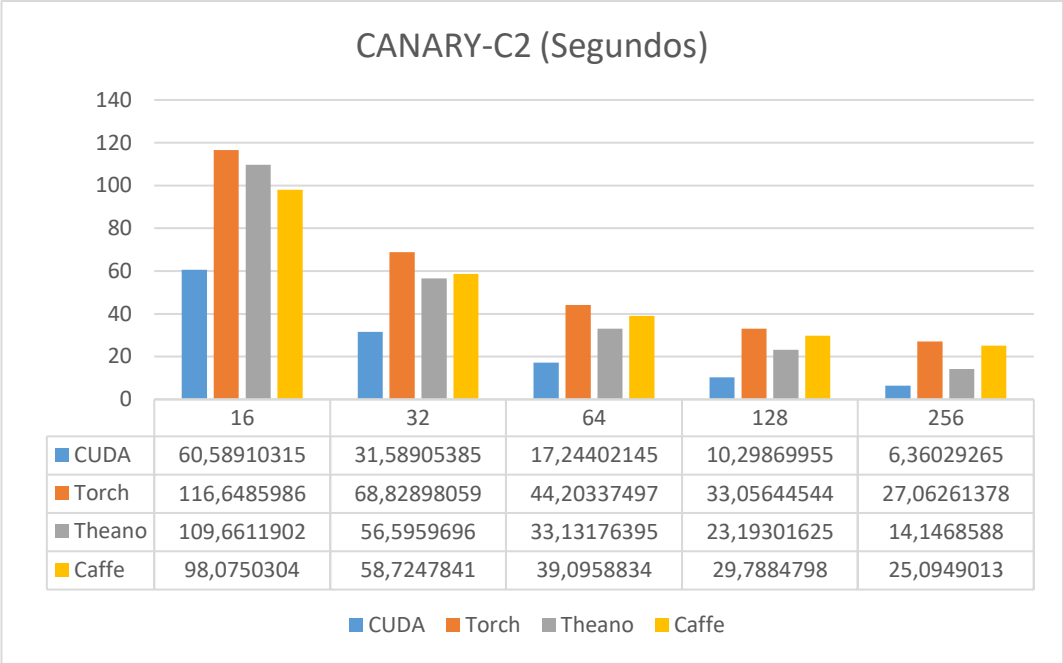


Figura 22: Tiempos de entrenamiento por epoch para CANARY - Fase C2

Con los tiempos de ejecución, la situación es muy similar a la vista en la Fase B1, tal y como se puede comprobar en la Figura 23. El código en CUDA vuelve a lograr los mejores tiempos de ejecución en ambos casos, seguido por *Torch*, teniendo *Theano* los peores resultados. Aquí también aparece una diferencia significativa entre tener los datos ya cargados en la RAM, frente a tener que cargarlos desde el disco duro, aunque la diferencia es ligeramente menor que en el caso previo.

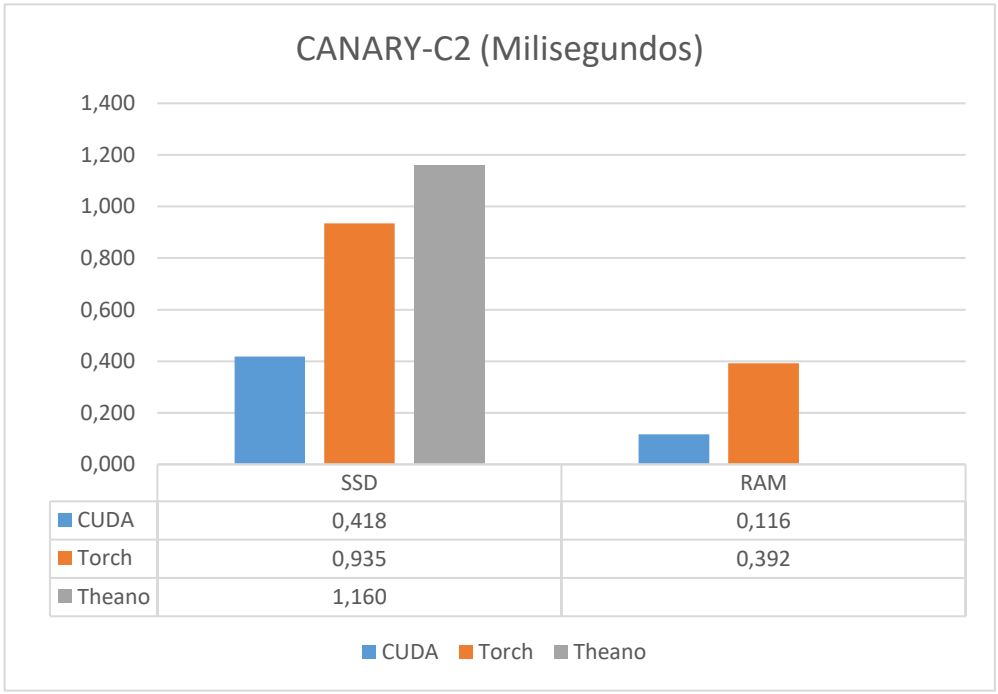
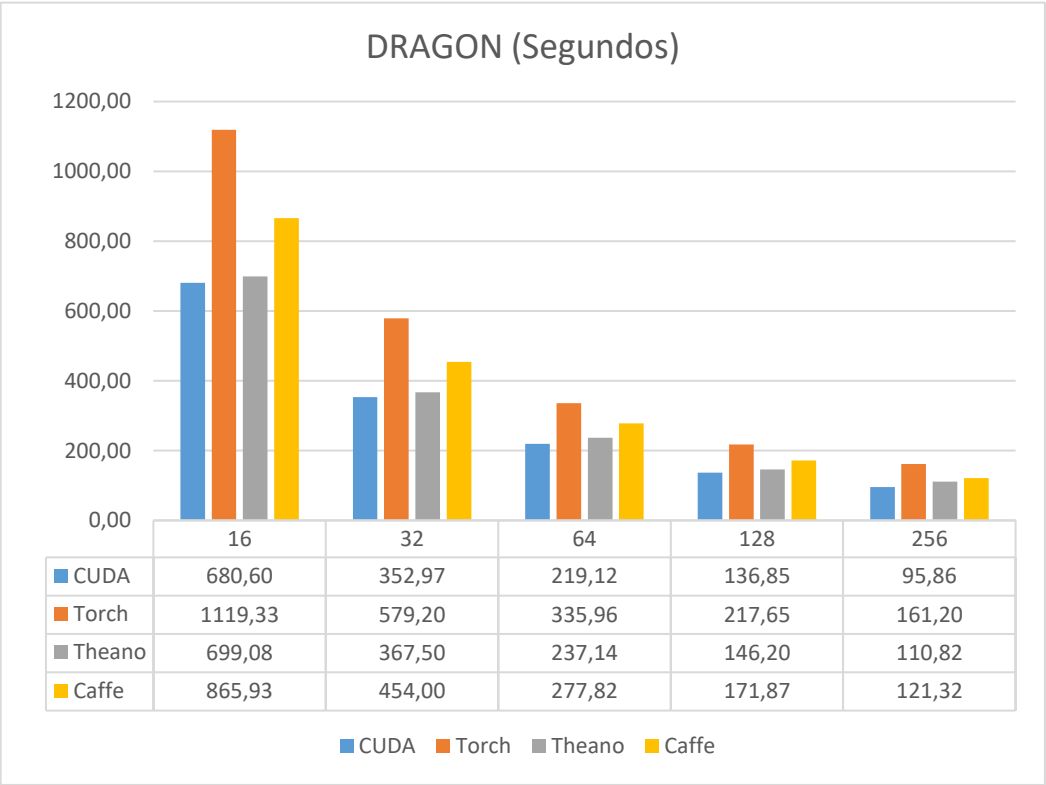


Figura 23: Tiempos de Ejecución para CANARY Fase C2

4.3. DRAGON

La red más grande de las analizadas, va a contar con 7.200-7.200-1.800 neuronas en cada capa, y se utilizarán un millón de muestras para su entrenamiento. En este caso, los tiempos de entrenamiento son mucho más similares entre los diferentes *frameworks*, y la reducción de tiempo debida al aumento del tamaño de los lotes es aún más grande, tal y como se puede comprobar en la Figura 24.

Como en todos los casos anteriores, el código en CUDA es el que consigue los mejores resultados. Sin embargo, esta vez el resto de *frameworks* obtienen unos valores mucho más cercanos, especialmente en el caso de *Theano*, que prácticamente llega a igualar los tiempos para los lotes más pequeños. Tanto *Caffe* como *Torch* muestran una gran mejor al aumentar el tamaño del lote, llegando a quedarse bastante cerca de los tiempos de CUDA para los lotes más grandes.



	16	32	64	128	256
CUDA	680,60	352,97	219,12	136,85	95,86
Torch	1119,33	579,20	335,96	217,65	161,20
Theano	699,08	367,50	237,14	146,20	110,82
Caffe	865,93	454,00	277,82	171,87	121,32

Figura 24: Tiempos de entrenamiento por epoch para DRAGON

Los resultados obtenidos para las ejecuciones varían poco respecto a los sistemas previos, tal y como se puede ver en la Figura 25. De nuevo, el código en CUDA es el que más rápido consigue obtener una salida, seguido por *Torch* y *Theano*. Al igual que en el caso del entrenamiento, las diferencias entre los diferentes *frameworks* se reducen, debido al mayor tamaño de la red. Además, es interesante destacar como ahora la diferencia que existe entre cargar los datos directamente desde el disco duro, o tenerlos

almacenados ya en la RAM, es proporcionalmente más pequeña que para todos los sistemas anteriores.

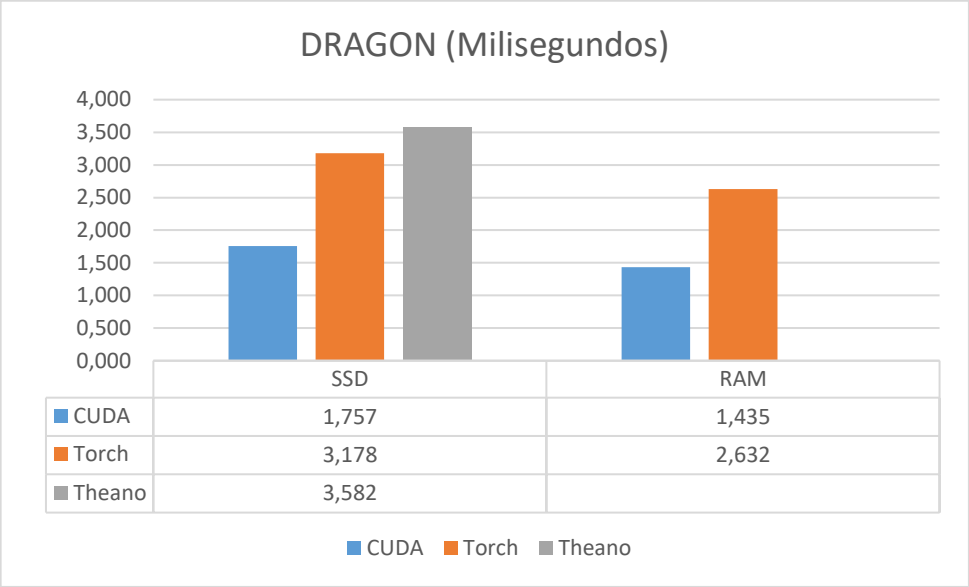


Figura 25: Tiempos de Ejecución para DRAGON

4.4. Análisis de Resultados

Podemos ver cómo especialmente en las redes pequeñas, las diferencias entre los distintos *frameworks* pueden ser enormes. A priori, entrenar una red pequeña como CANARY-B1 llevaría tan solo unos minutos con cualquiera de ellos, lo cual es un tiempo bastante razonable. Sin embargo, es interesante reducir ese tiempo lo máximo posible, ya que puede la posibilidad de realizar algún tipo de entrenamiento *on-line*, puede proporcionar una mejora en los resultados. Como se ha explicado anteriormente, en los grandes telescopios, las condiciones atmosféricas son enormemente cambiantes, además de que varían a gran velocidad, en unos pocos milisegundos. Si se pudiese re-entrenar la red en esos márgenes de tiempo, con información que procediese de la observación actual, es posible que la calidad de la reconstrucción fuese aún mayor, lo que permitiría obtener imágenes más nítidas.

En los casos de las redes más grandes, las diferencias van haciéndose cada vez más pequeñas. Especialmente en el caso de DRAGON, la diferencia entre CUDA y *Theano*, llega a ser casi de cero. Este comportamiento, puede ser explicado por la forma en que se emplean los diferentes tiempos de cálculo. En las redes pequeñas, cualquier cálculo extra introducido por cualquiera de los *frameworks*, puede suponer una gran diferencia. En el caso de CUDA, al ser un código especialmente diseñado para tratar con una red del tipo perceptrón multicapa, no se introduce ningún tipo de operación que no vaya a ser

utilizada por la propia red. Sin embargo, cuando el tamaño de la red crece, el tiempo que se emplea calculando las salidas y retropropagando el error, va siendo cada vez mayor, por lo que las pequeñas operaciones extra que pudieran introducirse, van teniendo un impacto cada vez menor.

Además, es importante mencionar que los diferentes *frameworks* utilizan diferentes librerías para computar las operaciones con matrices necesarias para entrenar la red. Aunque no ha sido posible encontrar información precisa al respecto, en el código fuente de los tres *frameworks*, se han encontrado referencias a dos de las librerías de computación de matrices más utilizadas en GPU: *cuBLAS y Thrust*. Aunque ambas están optimizadas para funcionar sobre tarjetas gráficas, su rendimiento puede ser notablemente distinto, especialmente cuando la carga de trabajo es pequeña, como ocurre en algunas de las redes presentadas previamente. Esta es otra de las posibles razones para que existan diferencias en los tiempos de cálculo de las diferentes soluciones presentadas.

En el caso de la ejecución de la red, de nuevo el código en CUDA ha sido el más rápido para los diferentes sistemas de óptica adaptativa propuestos. En los sistemas de tiempo real de los grandes telescopios, se suele recibir entradas nuevas cada 2 milisegundos aproximadamente. Tradicionalmente, este tiempo se ha tomado como referencia para que el reconstructor proporcione una salida. Se considera que si se tarda más de 2 milisegundos en recolocar el espejo deformable, la corrección aplicada no será lo suficientemente correcta, ya que la turbulencia atmosférica puede haber variado bastante durante ese tiempo.

Para los sistemas más pequeños, todas las soluciones propuestas logran estar por debajo de ese límite. Sin embargo, en el caso de DRAGON, solo el código en CUDA puede quedarse por debajo, llegando a rozarlo cuando hay que cargar la imagen desde la memoria RAM. Como se ha explicado anteriormente, aunque exista un límite teórico a partir del cual una salida no se considera buena, es crucial bajar los tiempos de ejecución lo máximo posible, para poder adaptar la corrección a la observación que se está realizando en ese momento.

Es fácil comprobar que en todos los escenarios presentados, la solución desarrollada en CUDA es la opción más rápida. Se puede decir que es un resultado esperable, ya que el resto de *frameworks* están diseñados para trabajar con redes neuronales mucho más complejas, en lugar de un simple perceptrón multicapa. Sin embargo, cuando las redes crecen, esta diferencia disminuye, por lo que es de esperar que cuando el número de neuronas se dispare, las diferencias sean notablemente más pequeñas.

Sin embargo, en los tiempos de ejecución, al ser una operación que requiere mucha menos cantidad de cálculo, las diferencias pueden ser notables, y se puede aprovechar la nula cantidad de trabajo extra que introduce CUDA respecto al resto de *frameworks*, para optimizar al máximo posible dichos tiempos, aunque la red pueda seguir creciendo.

Capítulo 5: Conclusiones y Líneas Futuras

A lo largo del presente trabajo final de máster, se ha realizado un análisis sobre el funcionamiento de diferentes *frameworks* de redes neuronales, cuando empleamos un reconstructor de óptica adaptativa. Para ello, se ha empezado por estudiar el funcionamiento de los telescopios de gran tamaño, poniendo especial énfasis en los sistemas de óptica adaptativa, que serán cruciales para comprender el resto del trabajo. También se ha detallado el funcionamiento de una red neuronal, proceso que ha servido para facilitar su implementación en los diferentes *frameworks*. Además, se ha realizado una breve introducción a la computación en GPU, exponiendo sus beneficios a la hora de paralelizar las operaciones.

Una vez adquiridos todos los conocimientos necesarios, se ha propuesto un experimento donde se ha comparado el rendimiento de los diferentes *frameworks*, cuando son empleados en sistemas de tiempo real de varios tamaños. La ejecución de este experimento, permite extraer una serie de conclusiones en cuanto al rendimiento cuantitativo obtenido. Sin embargo, es interesante destacar que todos los conocimientos adquiridos durante la realización de este trabajo, proporcionan una información más amplia en cuanto a las posibles conclusiones, y a la hora de plantear las futuras líneas de trabajo.

5.1. Conclusiones

Si se tiene solo en cuenta los resultados numéricos obtenidos, es evidente que programar la red neuronal directamente en CUDA, ofrece los mejores resultados posibles. En segundo lugar se puede considerar que *Theano* es la segunda solución más rápida, aunque con ciertas excepciones, como en el caso de redes pequeñas con tamaños de lote pequeños. Por último, las diferencias entre *Caffe* y *Torch* son

considerablemente pequeñas, por lo que serían las dos opciones más lentas, al menos para los sistemas en tiempo real estudiados.

Es interesante en este punto, hablar sobre la complejidad de la programación, y la ganancia que se consigue con ello. En el caso de CUDA, es necesario que el programador tenga conocimientos tanto de C, como de CUDA, y sepa manejar algunas de las librerías que *Nvidia* proporciona. Esto hace que si la red que se quiera estudiar es más compleja que la presentada aquí, pueda necesitarse un esfuerzo a nivel de programación considerable, ya que cualquier pequeño cambio en la arquitectura de la red, puede implicar grandes cambios en el código. Es por ello que hay que analizar si el beneficio en cuanto a los tiempos de entrenamiento o ejecución, compensaría los esfuerzos hechos en el desarrollo.

En el caso del resto de sistemas, especialmente con *Torch* y *Caffe*, se requiere mucho menos esfuerzo para cambiar la red, ya que están pensados para realizar estos cambios con facilidad. Sin embargo, estos sistemas son los más lentos de los estudiados. Esto hace pensar en la posibilidad de utilizar alguno de ellos para analizar nuevas redes, con distinto número de capas, o incluso capas distintas, donde es mucho más importante la flexibilidad a la hora de cambiar las características de la red, que la velocidad del entrenamiento. Una vez decidida la topología definitiva de la red, sobre la que los cambios que se realizarán serán prácticamente nulos, puede ser el momento de plantearse el esfuerzo de programar el sistema en CUDA, con el fin de optimizar los tiempos de entrenamiento de cara a las pruebas en telescopios reales.

Es importante destacar en este punto, que este trabajo final de máster es una primera etapa de un proyecto notablemente más grande, y que gran parte del trabajo realizado, ha consistido en comprender y familiarizarse con muchos de los conceptos que se han introducido en los primeros capítulos. Todos estos conocimientos, facilitarán la resolución de los nuevos retos que se plantean en el futuro, y que serán presentados más adelante.

Por último, es importante mencionar que los diferentes *frameworks* analizados están en constante cambio. Los diferentes equipos responsables, además de la comunidad de desarrolladores buscan mejorar cada día el rendimiento de cada uno de ellos, por lo que puede ser interesante volver a realizar las mismas pruebas de forma periódica, comprobando si los resultados obtenidos han variado, o se mantienen constantes. El hecho de haberse familiarizado con los diferentes lenguajes, de haber creado los códigos y tener una serie de experimentos claramente definidos, ayudará también a facilitar dichas comparaciones en el futuro, además de que siempre existe la posibilidad de añadir nuevos *frameworks* a la comparativa.

5.2. Líneas Futuras

Uno de los aspectos más importantes que se buscaba conseguir a la hora de realizar este trabajo, era abrir nuevas puertas en el futuro de la aplicación de redes neuronales en los sistemas de óptica adaptativa. La implementación original de CARMEN era considerablemente limitada, y no permitía la posibilidad de probar nuevos tipos de redes neuronales, o incluso ejecutarse y entrenarse en tiempos razonables en sistemas de cierto tamaño.

Los códigos creados, aún optimizados para ser lo más rápidos posibles, poseen márgenes de mejora en otros campos. En primer lugar, sería muy importante mejorar la gestión del uso de la RAM y la VRAM, especialmente cuando el tamaño de las redes se dispare. Es de esperar que las matrices de pesos de una red diseñada para funcionar en el futuro E-ELT, puedan alcanzar las decenas o incluso centenas de *gigabytes*, lo cual llenaría completamente la memoria de cualquier GPU, además de que no se estaría haciendo un uso óptimo de la memoria. También en el caso de la RAM principal, sería importante hacer una gestión correcta de la cantidad de datos usada para entrenar, ya que se prevé que puedan llegar a alcanzar varios *terabytes* de información.

Una de las futuras mejoras que se realizarán también en la implementación de CARMEN, es la de utilizar sistemas multi-GPU. Esto proporcionará principalmente dos mejoras sobre lo ya existente. Por un lado, al tener varias tarjetas trabajando juntas, se dispondrá de mucha más VRAM para repartir las matrices de pesos, o los datos de entrenamiento con los que se esté trabajando. Por otro lado, al repartir el trabajo entre tarjetas, es esperable que los tiempos de entrenamiento y de ejecución se reduzcan también de forma notable.

Algunos de los *frameworks* estudiados, ofrecen la posibilidad de realizar un entrenamiento utilizando varias GPUs de forma simultánea, repartiendo el lote entre las distintas tarjetas, y posteriormente compartiendo entre ellas la actualización de los pesos. Sin embargo, esta opción es bastante trivial, y aunque proporciona cierta mejora en los tiempos de entrenamiento, no sirve para optimizar el uso de la memoria VRAM de las tarjetas, ni permite utilizar varias GPUs para ejecutar una única entrada. Por ello, sería interesante estudiar las posibles implementaciones en multi-GPU de la red, y analizar cuáles son sus ventajas y los tiempos de entrenamiento y ejecución que pueden ofrecer.

Otro aspecto muy interesante a la hora de disponer de diferentes *frameworks* para crear redes, es que se pueden analizar nuevas arquitecturas o tipos de redes para mejorar el funcionamiento de CARMEN. A lo largo del presente trabajo, se ha optado

por dejar fuera cualquier tipo de comparativa respecto a la calidad de los resultados que ofrece el reconstructor, ya que se han proporcionado una serie de referencias donde se probaba su buen funcionamiento. Sin embargo, los resultados obtenidos por CARMEN tienen aún bastante margen de mejora, por lo que será especialmente interesante, analizar en un futuro su rendimiento, cuando se trabaje con distintos tipos de redes neuronales.

Una de las arquitecturas más comunes, y que se encuentra implementada en todos los *frameworks* analizados, son las redes neuronales convolucionales [46], que han proporcionado una importante mejora a la hora de resolver algunos de los problemas más habituales en el procesado de imagen, como en el caso de MNIST [25]. Este tipo de red, podría procesar la imagen obtenida directamente del sensor de frente de onda, en lugar de convertir dicha imagen a un conjunto de centroides, lo que podría proporcionar una mejora importante en el rendimiento del reconstructor.

También sería muy interesante, estudiar las posibilidades de realizar un entrenamiento *on-sky*, que permitiese a la red adaptarse a las condiciones de la observación, en lugar de ser entrenada antes de llevarla al telescopio. Esto podría hacer al reconstructor aún más robusto, ya que la red se estaría adaptando a las condiciones ambientales de cada momento, lo que debería mejorar su rendimiento. Otra idea interesante, sería la utilización de redes neuronales recurrentes [47], que pudiesen aprovechar la continuidad de las observaciones, para mejorar la respuesta que proporciona el reconstructor.

Bibliografía

[1] J. M. Beckers, “ADAPTIVE OPTICS FOR ASTRONOMY: Principles, Perfomance, and Applications,” *Annu. Rev. Astron. Astrophys.*, vol. 31, pp. 13–62, 1993.

[2] B. L. Ellerbroek, “First-order performance evaluation of adaptive-optics systems for atmospheric-turbulence compensation in extended-field-of-view astronomical telescopes,” *J. Opt. Soc. Am. A*, vol. 11, no. 2, p. 783, Feb. 1994.

[3] T. Fusco, J.-M. Conan, G. Rousset, L. M. Mugnier, and V. Michau, “Optimal wave-front reconstruction strategies for multiconjugate adaptive optics,” *J. Opt. Soc. Am. A*, vol. 18, no. 10, p. 2527, Oct. 2001.

[4] F. Vidal, E. Gendron, and G. Rousset, “Tomography approach for multi-object adaptive optics.,” *J. Opt. Soc. Am. A. Opt. Image Sci. Vis.*, vol. 27, no. 11, pp. A253-64, Nov. 2010.

[5] J. Osborn, D. Guzman, F. J. D. C. Juez, A. G. Basden, T. J. Morris, E. Gendron, T. Butterley, R. M. Myers, A. Guesalaga, F. S. Lasheras, M. G. Victoria, M. L. Sánchez Rodríguez, D. Gratadour, and G. Rousset, “Open-loop tomography with artificial neural networks on CANARY: On-sky results,” *Mon. Not. R. Astron. Soc.*, vol. 441, no. 3, pp. 2508–2514, 2014.

[6] A. G. Basden, D. Atkinson, N. A. Bharmal, U. Bitenc, M. Brangier, T. Buey, T. Butterley, D. Cano, F. Chemla, P. Clark, M. Cohen, J.-M. Conan, F. J. De Cos, C. Dickson, N. A. Dipper, C. N. Dunlop, P. Feautrier, T. Fusco, J. L. Gach, E. Gendron, D. Geng, S. J. Goodsell, D. Gratadour, A. H. Greenaway, A. Guesalaga, C. D. Guzman, D. Henry, D. Holck, Z. Hubert, J. M. Huet, A. Kellerer, C. Kulcsar, P. Laporte, B. Le Roux, N. Looker, A. J. Longmore, M. Marteaud, O. Martin, S. Meimon, C. Morel, T. J. Morris, R. M. Myers, J. Osborn, D. Perret, C. Petit, H. Raynaud, A. P. Reeves, G. Rousset, F. Sanchez Lasheras, M. Sanchez Rodriguez, J. D. Santos, A. Sevin, G. Sivo, E. Stadler, B. Stobie, G. Talbot, S. Todd, F. Vidal, and E. J. Younger, “Experience with wavefront sensor and deformable mirror interfaces for wide-field adaptive optics systems,” *Mon. Not. R. Astron. Soc*, vol. 459, no. 2, pp. 1350–1359, 2016.

[7] S. K. Ramsay, M. M. Casali, J. C. González, and N. Hubin, “The E-ELT instrument

roadmap: a status report," p. 91471Z, 2014.

[8] N. A. Dipper, A. Basden, U. Bitenc, R. M. Myers, A. Richards, and E. J. Younger, "Adaptive Optics for Extremely Large Telescopes III ADAPTIVE OPTICS REAL-TIME CONTROL SYSTEMS FOR THE E-ELT," in *Adaptive Optics for Extremely Large Telescopes III*, 2013.

[9] J. G. Marichal-Hernández, L. F. Rodríguez-Ramos, F. Rosa, and J. M. Rodríguez-Ramos, "Atmospheric wavefront phase recovery by use of specialized hardware: graphical processing units and field-programmable gate arrays.," *Appl. Opt.*, vol. 44, no. 35, pp. 7587–94, Dec. 2005.

[10] H. Ltaief and D. Gratadour, "Shooting for the Stars with GPUs," *GPU Technology Conference*, 2015. [Online]. Available: http://on-demand.gputechconf.com/gtc/2015/video/S5122.html. [Accessed: 14-Mar-2016].

[11] B. C. Platt and R. Shack, "History and Principles of Shack-Hartmann Wavefront Sensing," *J. Refract. Surg.*, vol. 17, no. 5, pp. S573–S577, 2001.

[12] F. Roddier, "Curvature sensing and compensation: a new concept in adaptive optics," *Appl. Opt.*, vol. 27, no. 7, pp. 1223–1225, 1988.

[13] S. Esposito and A. Riccardi, "Pyramid Wavefront Sensor behavior in partial correction Adaptive Optic systems," *Astron. Astrophys.*, vol. 369, no. 2, pp. L9–L12, Apr. 2001.

[14] D. Guzmán, F. J. de C. Juez, F. S. Lasheras, R. Myers, and L. Young, "Deformable mirror model for open-loop adaptive optics using multivariate adaptive regression splines," *Opt. Express*, vol. 18, no. 7, pp. 6492–6505, 2010.

[15] R. M. Myers, Z. Hubert, T. J. Morris, E. Gendron, N. A. Dipper, A. Kellerer, S. J. Goodsell, G. Rousset, E. Younger, M. Marteaud, and others, "CANARY: the on-sky NGS/LGS MOAO demonstrator for EAGLE," in *SPIE Astronomical Telescopes+ Instrumentation*, 2008, p. 70150E--70150E.

[16] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bull. Math. Biophys.*, vol. 5, no. 4, pp. 115–133, Dec. 1943.

[17] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in …," *Psychol. Rev.*, vol. 65, no. 6, pp. 386–408, 1958.

[18] M. Minsky and S. Papert, "Perceptron: an introduction to computational geometry," *MIT Press*, vol. 19, no. 88, p. 2, 1969.

[19] P. Werbos, "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences," phdthesis, Harvard University, Cambridge, MA, 1974.

[20] D. E. Rumelhart, J. L. McClelland, P. D. P. R. Group, and others, *Parallel distributed processing*, vol. 1. IEEE, 1988.

[21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[22] S. Hochreiter, S. Hochreiter, J. Schmidhuber, and J. Schmidhuber, "Long short-term memory.," *Neural Comput.*, vol. 9, no. 8, pp. 1735–80, Nov. 1997.

[23] D. Wang and G. Hinton, “Unsupervised learning: Foundations of neural computation,” *Comput. Math. with Appl.*, vol. 38, no. 5–6, p. 256, 1999.

[24] J. Schmidhuber, “Deep Learning in neural networks: An overview,” *Neural Networks*, vol. 61. pp. 85–117, 2015.

[25] Y. LeCunn, C. Cortes, and C. Burges, “MNIST Database.” [Online]. Available: http://yann.lecun.com/exdb/mnist/. [Accessed: 14-Jun-2016].

[26] S. Bendersky, N. S. Kopeika, and N. Blaunstein, “Atmospheric optical turbulence over land in middle east coastal environments: prediction modeling and measurements,” *Appl. Opt.*, vol. 43, no. 20, pp. 4070–4079, Jul. 2004.

[27] D. Guzmán, F. J. D. C. Juez, R. Myers, A. Guesalaga, and F. S. Lasheras, “Modeling a MEMS deformable mirror using non-parametric estimation techniques.,” *Opt. Express*, vol. 18, no. 20, pp. 21356–21369, 2010.

[28] J. Osborn, F. J. De Cos Juez, D. Guzman, T. Butterley, R. Myers, A. Guesalaga, and J. Laine, “Open-loop tomography using artificial nueral networks,” *Adapt. Opt. Extrem. Large Telesc. II*, 2011.

[29] J. Osborn, F. J. D. C. Juez, D. Guzman, T. Butterley, R. Myers, A. Guesalaga, and J. Laine, “Using artificial neural networks for open-loop tomography,” *Opt. Express*, vol. 20, no. 3, pp. 2420–2434, 2012.

[30] F. J. de Cos Juez, F. Sánchez Lasheras, N. Roqueñí, and J. Osborn, “An ANN-based smart tomographic reconstructor in a dynamic environment.,” *Sensors (Basel).*, vol. 12, no. 7, pp. 8895–911, 2012.

[31] J. Osborn, D. Guzman, F. J. de Cos Juez, A. G. Basden, T. J. Morris, É. Gendron, T. Butterley, R. M. Myers, A. Guesalaga, F. Sanchez Lasheras, M. Gomez Victoria, M. L. Sánchez Rodríguez, D. Gratadour, and G. Rousset, “First on-sky results of a neural network based tomographic reconstructor: Carmen on Canary,” in *SPIE Astronomical Telescopes + Instrumentation*, 2014, p. 91484M.

[32] M. Gómez Victoria, “Investigación del problema inverso de reconstrucción tomográfica en óptica adaptativa para astronomía a través de técnicas de minería de datos e inteligencia artificial,” Universidad de Oviedo, 2014.

[33] M. Nielsen, “Neural Networks and Deep Learning,” 2016. [Online]. Available: http://neuralnetworksanddeeplearning.com/index.html. [Accessed: 26-Aug-2016].

[34] NVIDIA Corporation, “NVIDIA GeForce GTX 980 Featuring Maxwell, The Most Advanced GPU Ever Made,” 2014.

[35] NVIDIA Corporation, “NVIDIA Tesla P100,” 2016.

[36] NVIDIA Corporation, “NVIDIA GeForce GTX 1080,” 2016.

[37] NVIDIA Corporation, *CUDA C Programming Guide*. 2015.

[38] NVIDIA Corporation, “GPU-Accelerated Libraries.” [Online]. Available: https://developer.nvidia.com/gpu-accelerated-libraries. [Accessed: 28-Aug-2016].

[39] A. Chrzszczyk and J. Chrzszczyk, *Matrix computations on the GPU CUBLAS and*

MAGMA by example. 2013.

[40] NVIDIA Corporation, “cuDNN Library, User Guide,” 2016.

[41] C. Gulcehre, “Deep Learning - Software Links.” [Online]. Available: http://deeplearning.net/software_links/. [Accessed: 27-Aug-2016].

[42] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, “Comparative Study of Deep Learning Software Frameworks,” vol. 2, no. 1, pp. 1–14, Nov. 2015.

[43] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional Architecture for Fast Feature Embedding,” in *Proceedings of the ACM International Conference on Multimedia - MM ’14*, 2014, pp. 675–678.

[44] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, Y. Bengio, A. Bergeron, J. Bergstra, V. Bisson, J. B. Snyder, N. Bouchard, N. Boulanger-Lewandowski, X. Bouthillier, A. de Brébisson, O. Breuleux, P.-L. Carrier, K. Cho, J. Chorowski, P. Christiano, T. Cooijmans, M.-A. Côté, M. Côté, A. Courville, Y. N. Dauphin, O. Delalleau, J. Demouth, G. Desjardins, S. Dieleman, L. Dinh, M. Ducoffe, V. Dumoulin, S. E. Kahou, D. Erhan, Z. Fan, O. Firat, M. Germain, X. Glorot, I. Goodfellow, M. Graham, C. Gulcehre, P. Hamel, I. Harlouchet, J.-P. Heng, B. Hidasi, S. Honari, A. Jain, S. Jean, K. Jia, M. Korobov, V. Kulkarni, A. Lamb, P. Lamblin, E. Larsen, C. Laurent, S. Lee, S. Lefrancois, S. Lemieux, N. Léonard, Z. Lin, J. A. Livezey, C. Lorenz, J. Lowin, Q. Ma, P.-A. Manzagol, O. Mastropietro, R. T. McGibbon, R. Memisevic, B. van Merriënboer, V. Michalski, M. Mirza, A. Orlandi, C. Pal, R. Pascanu, M. Pezeshki, C. Raffel, D. Renshaw, M. Rocklin, A. Romero, M. Roth, P. Sadowski, J. Salvatier, F. Savard, J. Schlüter, J. Schulman, G. Schwartz, I. V. Serban, D. Serdyuk, S. Shabanian, É. Simon, S. Spieckermann, S. R. Subramanyam, J. Sygnowski, J. Tanguay, G. van Tulder, J. Turian, S. Urban, P. Vincent, F. Visin, H. de Vries, D. Warde-Farley, D. J. Webb, M. Willson, K. Xu, L. Xue, L. Yao, S. Zhang, and Y. Zhang, “Theano: A Python framework for fast computation of mathematical expressions,” p. 19, 2016.

[45] The HDF Group, “Introduction to HDF5,” 2010. [Online]. Available: https://www.hdfgroup.org/HDF5/doc/H5.intro.html#Intro-WhatIs. [Accessed: 20-Jun-2016].

[46] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” *Adv. Neural Inf. Process. Syst.*, pp. 1097–1105, 2012.

[47] K. Funahashi and Y. Nakamura, “Approximation of dynamical systems by continuous time recurrent neural networks,” *Neural Networks*, vol. 6, no. 6, pp. 801–806, 1993.