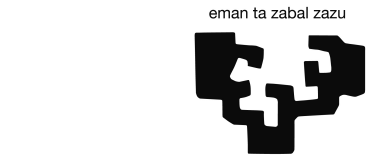

Grado en Ingeniería Informática

Ingeniería de Computadores

Proyecto de Fin de Grado

Comparación del rendimiento de plataformas para el desarrollo de sistemas de aprendizaje profundo

Autor

Iker Moya González

Directores

Ibai Gurrutxaga y Javier Muguerza

2017

Agradecimientos

Primeramente quiero agradecer el que este proyecto haya llegado a buen término a la labor, ayuda y apoyo de mis directores *Ibai Gurrutxaga* y *Javier Muguerza*.

Agradezco a mi madre, *Maite*, la confianza que tuvo en mí para que decidiese emprender los estudios de Ingeniería Informática, los consejos y apoyos que de ella he recibido en los momentos más dificultosos y duros que han ido apareciendo.

También quiero dejar mis agradecimientos aquí por escrito a la cuadrilla que hemos formado durante estos años de estudios en esta facultad. Con una mención especial para *Igor Santesteban*, por dedicarme su tiempo libre y acompañarme en las largas jornadas de dedicación al proyecto y a los estudios en la universidad.

Y en último lugar pero no menos importante, te agradezco a ti, por dedicarme parte de tu tiempo, por leer y consultar esta memoria.

Resumen

En este proyecto se ha realizado un estudio comparativo de tres librerías de aprendizaje profundo:

- *Theano*
- *Tensorflow*
- *Keras*

El objetivo de este estudio es poder establecer criterios de decisión para el uso de estas herramientas.

Para ello, se tendrán en cuenta los siguientes aspectos:

1. El tipo de red neuronal utilizado: en este proyecto se ha trabajado con redes multicapa totalmente conectadas (MLP) y redes convolucionales (CNN).
2. El sistema de cómputo utilizado: la experimentación se ha realizado sobre dos máquinas diferentes, una para ejecutar sobre la CPU (con y sin *multithreading*) y otra para ejecutar en la GPU.

La realización de este estudio ha tenido un trabajo previo de aprendizaje tanto de las herramientas como de los conceptos teóricos subyacentes.

Índice general

Índice de figuras

Índice de tablas

1. CAPÍTULO

Introducción

Este proyecto se centra en el campo del aprendizaje profundo (*deep learning*). Este campo engloba un conjunto de técnicas y algoritmos que son comúnmente utilizados para procesar grandes cantidades de datos.

La finalidad de este procesamiento es el desarrollo de sistemas que puedan ser utilizados para resolver problemas de ámbitos tan diversos como la biomedicina, la robótica y la visión por computador.

Dada la dimensión de las bases de datos con las que se suele trabajar, el uso de herramientas eficientes juega un papel fundamental.

En este proyecto se han analizado tres plataformas para el desarrollo de sistemas de aprendizaje profundo: *Theano, Tensorflow* y *Keras*.

A continuación, se presentan cuáles han sido los objetivos propuestos y la manera en la que se ha planificado el proyecto para cumplirlos.

1.1. Objetivos del proyecto

Los principales objetivos de este proyecto son los siguientes:

- Entender el funcionamiento de las redes neuronales y el aprendizaje profundo.
- Ser capaz de implementar las redes mediante diferentes librerías: *Keras*, *Theano* y *Tensorflow*.

- Realizar un análisis de las diferencias de rendimiento de las tres librerías de aprendizaje profundo analizadas sobre bases de datos estándar en aprendizaje automático.

1.2. Fases del proyecto

El proyecto se ha planificado en cuatro fases: aprendizaje, implementación, experimentación y redacción de la memoria.

La fase de aprendizaje ha servido para subsanar la falta de conocimiento previo en el campo del aprendizaje profundo.

Una vez adquirido este conocimiento se ha podido avanzar al resto de fases.

En los siguientes apartados se describe de manera más detallada cuáles han sido las tareas realizadas en cada fase.

1.2.1. Aprendizaje

Las tareas llevadas a cabo en esta fase han sido:

- Familiarizarse con el lenguaje de programación *Python*.
- Leer el libro *"Neural Networks and Deep Learning"* [Nielsen, 2017].
- Leer diferentes publicaciones relacionadas con el campo del aprendizaje profundo ([He et al., 2015], [LeCun et al., 1998a], etc.).
- Aprender el funcionamiento de *Keras*.
- Aprender el funcionamiento de *Theano*.
- Aprender el funcionamiento de *Tensorflow*.
- Realizar pruebas de los conceptos teóricos aprendidos.

La pruebas realizadas se han hecho utilizando la base de datos *MNIST* (Sección 2.3.1) que se centra en el reconocimiento de dígitos manuscritos.

El resto del conocimiento adquirido en esta fase está recogido en el capítulo 2.

1.2.2. Implementación

La fase de implementación se ha centrado en realizar programas con *Keras*, *Theano* y *Tensorflow* para crear redes neuronales totalmente conectadas y redes convolucionales de diferentes dimensiones.

Las partes más relevantes de los programas realizados se prestan en capítulo 3.

1.2.3. Experimentación

La fase de experimentación ha consistido en la obtención de tiempos de ejecución de todos los programas. Estas ejecuciones han abarcado distintas casos de prueba que se detallan en la sección 4.1.

Una vez obtenidos los tiempos de ejecución, la siguiente tarea ha consistido en la creación de tablas que faciliten el análisis de estos datos. Las conclusiones obtenidas de este análisis se presentan en el capítulo 4 acompañadas de las correspondientes gráficas.

1.2.4. Redacción de la memoria

La creación de este documento se ha realizado en dos etapas:

1. La creación del borrador de la memoria, el cual se ha ido realizando de forma paralela al resto de fases del proyecto.
2. La revisión final, en la que se han repasado todos los apartados desde una perspectiva de conjunto.

1.3. Dedicación

En la tabla 1.1 se muestran los tiempos dedicados a cada una de las fases junto con las horas dedicadas a las reuniones con los directores del proyecto.

Fase	**Dedicación**
Aprendizaje	92 h
Implementación	87 h
Experimentación	25 h
Redacción de la memoria	86 h
Reuniones	17 h

Tabla 1.1: Tiempo dedicado a cada fase del proyecto

2. CAPÍTULO

Tecnologías y bases de datos

2.1. Deep Learning

Deep learning (aprendizaje profundo) es un conjunto de algoritmos de aprendizaje automático jerárquicos con unidades de procesamiento no lineal que permiten aprender múltiples niveles de representación de la entrada que se corresponden con diferentes niveles de abstracción.

Por su estructura en capas formadas por unidades de proceso relativamente simples, es un paradigma que se sitúa en el mundo de las redes neuronales artificiales.

A diferencia de éstas, contienen un número elevado de capas (al menos dos transformaciones intermedias) y pueden ser utilizadas directamente sobre la información de entrada ya que son capaces de generar automáticamente las características sobre las que realizar la clasificación (o agrupamiento en caso de aprendizaje no supervisado).

El algoritmo de aprendizaje estándar para redes neuronales, conocido como **descenso estocástico de gradiente** es un modelo que permite a una computadora aprender de los datos de entrada.

Hoy en día, las redes neuronales y el aprendizaje profundo proporcionan las mejores soluciones a muchos problemas en reconocimiento de imágenes, reconocimiento de voz y procesamiento del lenguaje natural.

Pero, ¿qué se entiende por una red neuronal?

"Una red neuronal es un sistema de computación compuesto por un gran número de elementos de proceso simples, denominados **neuronas**, los cuales procesan información por medio de su estado dinámico como respuesta a entradas externas. Estos elementos siguen una organización jerárquica y están interconectados entre si" [Red, 2001].

Para entender las tareas realizadas, antes es necesario entender algunos conceptos generales.

2.1.1. Elementos básicos

2.1.1.1. Elementos básicos que componen una red neuronal

Para explicar los elemento básicos de una red neuronal en la figura 2.1 se presenta, a modo de ejemplo, la estructura una red neuronal.

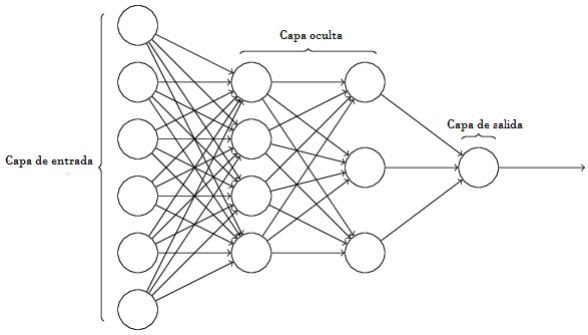

Figura 2.1: Estructura de una red neuronal [Nielsen, 2017]

La red está compuesta por neuronas interconectadas y organizadas en tres capas:

- La información proveniente de las fuentes externas es proporcionada a la red a través de la capa izquierda (denominada **capa de entrada**), las neuronas de esta capa se denominan neuronas de entrada.
- La **capa oculta**, es la parte central de la red neuronal y reciben datos de la capa de entrada, y proporcionan información procesada a la capa de salida. El número de niveles ocultos puede estar entre cero y un número elevado. Las neuronas de las capas ocultas pueden estar interconectadas de diferentes maneras, lo que determina las distintas topologías de redes neuronales.

- Finalmente, en la última capa se encuentra la **capa de salida**. Ésta es la responsable de dar los resultados finales de la red hacia el exterior.

Si analizamos la estructura de una neurona artificial (Figura 2.2), vemos que tiene asociadas unas entradas, unos pesos y unas funciones que utiliza para transformar los datos de entrada en los datos de salida (normalmente un único dato).

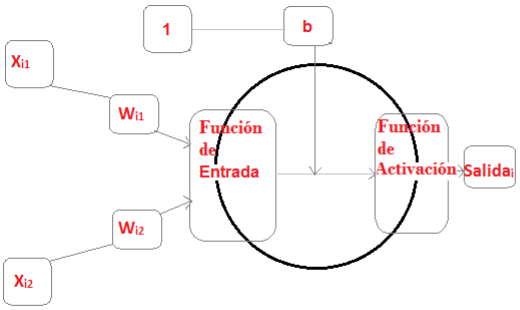

Figura 2.2: Ejemplo de una neurona con 2 entradas y 1 salida

2.1.1.2. Función de entrada

La neurona recibe diversos valores de entrada y los procesa mediante la expresión siguiente (producto interno entre las entradas y los pesos asociados a cada una de ellas):

$$input_i = (x_{i1} \cdot w_{i1}) + (x_{i2} \cdot w_{i2}) + ...(x_{in} \cdot w_{in})$$

Donde i es la neurona, x_i es el vector de entrada, w_i es el vector de pesos y n es el número de entradas a dicha neurona.

Los valores de entrada se multiplican por los pesos asociados a la neurona. De esta manera, los pesos permiten ajustar los valor de entrada en función de la relevancia que tenga cada característica para el problema que se desea resolver. Dicho de otro modo, los pesos es donde se almacena el 'conocimiento' de la red neuronal.

Normalmente, las neuronas tiene una entrada extra denominada *bias*. Este *bias* está formado por una entrada constante de magnitud 1 y un peso 'b'.

2.1.1.3. Función de Activación

La función activación determina el estado de actividad de una neurona; transforma la entrada global conocida como *net* en un valor (estado) de activación. El rango del valor puede estar acotado dependiendo de la función de activación empleada, por ejemplo en la sigmoidea [0,1], en la tangente hiperbólica [–1,1] o en la ReLU $[0,\infty)$.

Las funciones de activación más comunes son las siguientes, donde $z = net_i = input_i + b_i$:

1. Función Lineal:
$$f(z) = \begin{cases} -1 & z \leq \frac{-1}{a} \\ z * a & \frac{-1}{a} < z < \frac{1}{a} \\ 1 & z \geq \frac{1}{a} \end{cases}$$
2. Función Sigmoidea
$f(z) \equiv \frac{1}{1+e^{-z}}$
3. Función tangente hiperbólica
$f(z) \equiv \frac{e^{g*z} - e^{-g*z}}{e^{g*z} + e^{-g*z}}$
4. Función ReLU
$f(z) = max(0,z)$

2.1.2. Tipos de neuronas artificiales

Las neuronas artificiales se pueden clasificar en función del valor de salida que producen.

Principalmente, se pueden distinguir dos tipos [Nielsen, 2017], neurona con salida binaria o neurona con salida continua.

2.1.2.1. Neurona con salida binaria (Perceptrón)

Este tipo de neuronas producen una salida binaria, esto es, solamente pueden tomar valores 0 o 1.

Esto conlleva a que cuando se produzca un pequeño cambio en los pesos o en el *bias*, la salida puede generar un gran cambio en el funcionamiento de la red y en el resultado final. Este inconveniente hace que no se empleen los perceptrones en las redes neuronales.

2.1.2.2. Neurona con salida continua

A diferencia de los perceptrones, la salida en las neuronas continuas va sujeta a la función de activación.

Así, cuando se produce un pequeño cambio en algún peso o en el *bias*, la salida generada varía de manera continua. De este modo, se resuelve la inestabilidad que puede suceder en las binarias (sin que el funcionamiento de la red neuronal apenas se vea afectado).

Además, el hecho de que la función de activación utilizada sea derivable es algo deseable a la hora de diseñar los algoritmos de aprendizaje.

2.1.3. Mecanismo de aprendizaje

Cuando una red neuronal artificial es utilizada para clasificar, tiene que aprender a calcular la salida correcta para cada entrada sobre un conjunto de datos de entrenamiento. A este proceso de aprendizaje se denomina: **proceso de entrenamiento**.

Se pueden identificar dos tipos de procesos de aprendizaje: supervisado y no supervisado.

Este proyecto se centra en el contexto del aprendizaje supervisado (clasificación). En este tipo de aprendizaje se dispone de un conjunto de datos ya clasificados que permite entrenar la red de forma controlada. Esto posibilita que se pueda evaluar el rendimiento de la red comparando los resultados obtenidos de la red con los resultados ya clasificados.

Aunque no han sido objeto de estudio de este proyecto, existen redes neuronales artificiales que se aplican en el contexto de aprendizaje no supervisado (por ejemplo, clustering). En este tipo de aprendizaje no se dispone de una variable clase (etiqueta, salida deseada) asociada a cada patrón de entrenamiento.

2.1.3.1. Aprendizaje supervisado

El aprendizaje supervisado consiste en ajustar los pesos de las conexiones y *bias* de las neuronas de la red en función de la diferencia entre los valores deseados y los obtenidos a la salida de la red. Es decir, en función del error cometido en la salida.

En el proceso de aprendizaje se distinguen dos fases. La **fase de entrenamiento** y la **fase de testeo**, existiendo un conjunto de datos de entrenamiento y un conjunto de datos de test utilizados en la correspondiente fase.

Según la metodología empleada los datos de entrenamiento se pueden dividir en dos, reservando un subconjunto de ellos como datos de validación que se utilizan para determinar los parámetros óptimos (o pseudo-óptimos) de la red neuronal.

Durante la fase de entrenamiento de una red neuronal artificial los pesos de las conexiones y los *bias* de las neuronas de la red sufren modificaciones. La expresión que describe la actualización de los pesos suele ser similar a la siguiente:

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

o

$$\text{Peso Nuevo} = \text{Peso Viejo} + \text{Cambio de Peso}$$

Este proceso se repite de forma iterativa hasta que se cumpla alguno de los criterios de finalización establecidos.

Los criterios más comunes son los siguientes;

- Cuando los valores de los pesos y los *bias* se mantengan estables.
- Cuando se cumplen las iteraciones máximas de entrenamiento definidas anteriormente.
- Cuando la red sea capaz de clasificar correctamente más de un número determinado de casos de entrenamiento.

Existen diferentes algoritmos a seguir en el proceso de entrenamiento de una red neuronal. Un algoritmo muy conocido y fácil de entender es el **descenso de gradiente**.

2.1.3.2. Descenso de gradiente

Normalmente, para estimar la bondad de la red se suele utilizar una **función de coste**.

Un ejemplo sería la siguiente expresión, donde w denota la colección de todos los pesos en la red, b todos los *bias*, N el número total de entradas de entrenamiento y a el vector de salidas de la red cuando x es la entrada

$$C(w,b) \equiv \frac{1}{2N} \sum_x \|y(x) - a\|^2$$

El propósito del **descenso de gradiente** es buscar la dirección y la posición del mínimo global del valor de C.

También se puede expresar la función de coste del siguiente modo:

$$\triangle C \approx \bigtriangledown C * \triangle v$$

Donde:

- $\bigtriangledown C$ es el vector de gradientes: $\bigtriangledown C \equiv (\frac{\partial C}{\partial v_1}, ..., \frac{\partial C}{\partial v_m})^T$
- $\triangle v$ es el vector de cambios de los pesos y *bias*: $\triangle v = (\triangle v_1, ..., \triangle v_m)$.

  Para optimizar el coste de la red se modifican los pesos y *bias* en la dirección opuesta al gradiente.

  Por ello el vector de cambios será: $\triangle v = -\eta * \bigtriangledown C$, siendo η un parámetro positivo (conocido como la tasa de aprendizaje).
- El siguiente movimiento a realizar sería $v \rightarrow v' = v - \eta * \bigtriangledown C$

El proceso de aprendizaje es iterativo, se le presentan a la red sucesivamente los datos de entrenamiento (épocas) y se calcula repetidamente $\bigtriangledown C$ y luego se mueve hacia la dirección del mínimo, realizando cambios en los pesos o en los *bias*. La figura 2.3 muestra de manera visual este procedimiento.

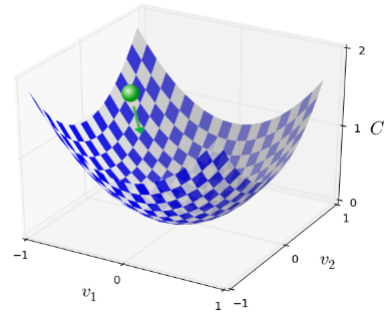

Figura 2.3: Descenso de Gradiente [Nielsen, 2017]

Por lo tanto, el *descenso de gradiente* puede ser visto como una forma de tomar pequeños pasos en la dirección en la que la función C se minimiza más rápidamente.

Desafortunadamente, cuando el número de entradas de entrenamiento es muy grande, este proceso puede tomar mucho tiempo y el aprendizaje ocurre lentamente.

Una variante de este método es el **descenso de gradiente estocástico**, el cual puede usarse para acelerar el aprendizaje. La idea de este método es estimar el gradiente $\bigtriangledown C$ calculando $\bigtriangledown C_x$ para una pequeña muestra de entradas de entrenamiento elegidas al azar.

$$\bigtriangledown C \approx \frac{1}{m} \sum_{j=1}^{m} \bigtriangledown C_{x_j}$$

Esto permite agilizar el proceso de aprendizaje con unos resultados aceptables frente a la aproximación anterior

2.1.4. Testeo de la Red Neuronal

Después del proceso de entrenamiento los pesos de las conexiones de la red y los *bias* de las neuronas quedan fijos. El siguiente paso consiste en verificar si red neuronal es capaz de resolver de manera adecuada los problemas para los que ha sido entrenada.

Por lo tanto, se requiere de otro conjunto de datos diferente al utilizado en la fase de entrenamiento con el propósito de testear la red neuronal, denominado **conjunto de test**.

Cada ejemplo del conjunto de test contiene los valores de las variables de entrada y su correspondiente salida deseada. Luego se compara la solución calculada para cada ejemplo de test con la solución conocida.

2.1.5. Elección del conjunto inicial de pesos y *bias*

Al inicio del proceso de entrenamiento se debe determinar un estado inicial, esto es, seleccionar un conjunto inicial de pesos para las conexiones entre las neuronas y los *bias* de las neuronas de la red neuronal.

Esto puede realizarse por varios criterios:

- **Valor constante**

  Se inicializa los pesos y/o los *bias* de la red neuronal con un valor fijo.

- **Valor aleatorio**

  Este criterio se basa en asignar un peso a cada conexión de manera aleatoria.

  Existen dos casos típicos en la asignación de los pesos de la red:

  1. Distribución uniforme entre $[-n, n]$.

2. Distribución Gaussiana con media 0.

Por lo general la inicialización de los *bias* no tiene tanta relevancia como la de los pesos. Por ello, es común inicializarlos a cero o con valores cercanos a cero.

Cabe mencionar que durante el transcurso del entrenamiento los pesos y los *bias* no se encuentran restringidos por la condición inicial.

2.1.6. Topologías Principales

2.1.6.1. Topología de las redes neuronales

La arquitectura de una red neuronal consiste en la organización y la distribución de las neuronas de la misma, formando diferentes tipos de capas o niveles.

En este sentido, los parámetros fundamentales de la topología de la red son: el número de capas, el número de neuronas por capa y el patrón de conectividad.

2.1.6.2. Redes Monocapa

Las redes monocapa, como expresa su nombre, son redes neuronales compuestas por una única capa. Este tipo de redes se utilizan para regenerar información de entrada que se presenta a la red de forma incompleta o distorsionada [Red, 2001].

2.1.6.3. Redes Multicapa

En las redes multicapas las neuronas están agrupadas en varios (2, 3, etc.) niveles o capas. Para distinguir la capa a la que pertenece una neurona hay que fijarse en el origen de las señales que reciben a la entrada y el destino de la señal de la salida.

Normalmente, todas la neuronas de una capa reciben señales de entrada desde otra capa anterior y envían señales de salida a una capa posterior.

2.1.6.4. Redes Convolucionales

Estas redes utilizan una arquitectura especial que está particularmente bien adaptada para clasificar imágenes. Son rápidas de entrenar y permiten generar redes profundas con muchas capas.

Las redes neuronales convolucionales se caracterizan por tres ideas básicas: los campos receptivos locales, los pesos compartidos y el agrupamiento de las capas.

- **Campos receptivos locales**

  En las redes que procesan imágenes es habitual que la entrada se represente como una matriz de dos dimensiones $n \times n$.

Figura 2.4: Capa de entrada en las redes convolucionales [Nielsen, 2017]

  Cada una de las neuronas de la capa oculta se conectan a una pequeña región de las neuronas de entrada denominada **el campo receptivo local de la neurona oculta**. Este campo es una pequeña ventana sobre los píxeles de entrada que es utilizada para aprender los pesos y *bias* asociados a dicha neurona. Se podría ver como una neurona oculta particular que aprende a analizar su campo receptivo (Figura 2.5).

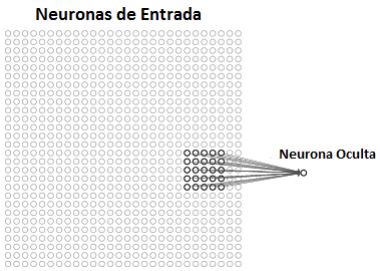

Figura 2.5: Conexión de la entrada con la neurona de la capa oculta en las redes convolucionales [Nielsen, 2017]

- **Pesos compartidos**

  Las conexiones de cada una de las neuronas ocultas a la capa de entrada comparten el mismo peso y *bias*. Al conjunto de neuronas ocultas se le denomina **mapa de características**, ya que permiten detectar características concretas de los datos de entrada.

Para el reconocimiento de imágenes se necesita más de un mapa de características. De este modo, una capa de convolución completa se compone de varios mapas de características diferentes:

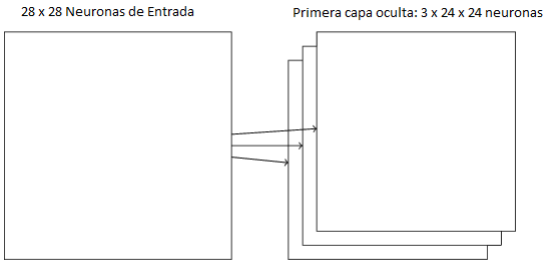

Figura 2.6: Estructura de la capa convolucional [Nielsen, 2017]

Una gran ventaja de compartir pesos y sesgos es que reduce en gran medida el número de parámetros involucrados en una red convolucional.

Como ejemplo, la imagen de entrada será una matriz 28×28 y el campo receptivo local de las neuronas ocultas de la primera capa de 5×5.

Para cada mapa de características se necesitarán $25 = 5 \times 5$ pesos compartidos, más un solo *bias* compartido. Así que cada mapa de características se requiere 26 parámetros. Si hay 20 mapas de características se tienen un total de $20 \times 26 = 520$ parámetros que definen la capa convolucional.

En comparación, si se define una capa completamente conectada de 30 neuronas ocultas, el número de parámetros asciende radicalmente. Si se tiene $784 = 28 \times 28$ datos de entrada, supone un total de 784×30 pesos más 30 *bias*, esto es, un total de 23.550 parámetros.

En otra palabras, la capa completamente conectada tendría aproximadamente 40 veces más parámetros que la capa convolucional.

Por esta misma razón, reduciendo el número de pesos a almacenar y calcular, se reduce de manera significativa el tiempo de ejecución y permite generar redes de mayores dimensiones.

- **Capas de agrupamiento**

  Las capas de agrupamiento ("*pooling*") se usan generalmente inmediatamente después de capas convolucionales. Lo que hacen las capas de agrupación es simplificar

la información en la salida de la capa convolucional, elevando el nivel de abstracción de la información aprendida.

En detalle, una capa de *pooling* toma cada mapa de características de salida de la capa convolucional y prepara un mapa de características condensado.

Por ejemplo, cada unidad en la capa de agrupación puede resumir la región de 2×2 neuronas en la capa anterior. La operación que más se utiliza para realizar la agrupación es la del valor máximo de la región y se conoce como *max-pooling*. La figura 2.7 ilustra la manera en la que se realiza el *max-pooling*:

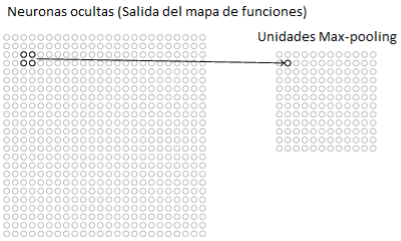

Figura 2.7: Max-pooling [Nielsen, 2017]

Una capa convolucional normalmente implica más de un solo mapa de características. Por lo que se aplica *max-pooling* a cada mapa de características por separado.

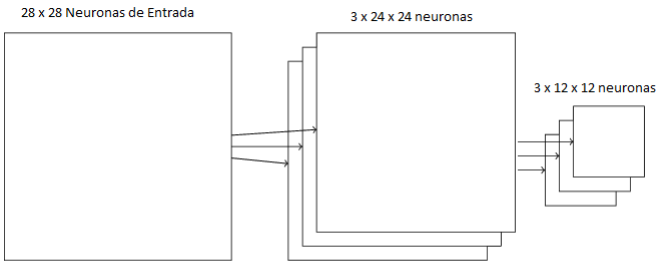

Figura 2.8: Ejemplo de una capa convolucional de una red CNN [Nielsen, 2017]

El *max-pooling* se puede describir como un método para que la red pregunte si la característica dada se encuentra en cualquier parte de una región de la imagen.

2.1.6.5. Capa *Softmax*

La idea del *softmax* es definir un nuevo tipo de capa de salida para las redes neuronales que se basa en la misma idea de una capa de neuronas totalmente conectadas. Sin embargo, se aplica la denominada función *softmax* para obtener las salidas, que se pueden interpretar como probabilidades.

Esto es muy útil en la clasificación, ya que da una medida de la probabilidad en las clasificaciones.

La función de activación softmax es:

$$a_j = \frac{e^{z}j}{\sum_k e^{z}k}$$

Donde, a_j es la activación de la neurona j, z_j es el valor calculado en la neurona j y el denominador es la suma de todos los valores calculados de las neuronas de salida.

2.1.6.6. LeNet-5

Dado que la arquitectura de una red convolucional puede tener ligeras variantes, vamos a mostrar, como ejemplo, la arquitectura LeNet-5 que utilizaremos como base para nuestra implementación.

La arquitectura de una red neuronal puede variar notablemente en función del ámbito en el que se quiera utilizar (detección de caracteres, reconocimiento facial, etc.). La implementación realizada en este proyecto (que se presenta en el capítulo 3) se inspira en la arquitectura de LeNet-5.

LeNet-5 es una red convolucional diseñada para reconocimiento de caracteres manuscritos e impresos por máquina [LeCun et al., 1998a].

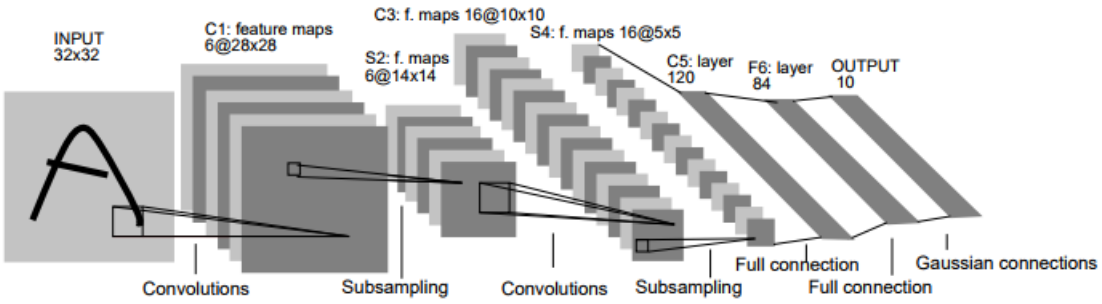

Figura 2.9: Imagen obtenida de [LeCun et al., 1998a] que muestra la arquitectura de LeNet-5

Como se puede observar en la figura 2.9, esta topología de red está formada por varias capas:

1. Una capa convolucional de 28×28 y pooling de (2,2).
2. Otra capa convolucional de 10×10 y pooling de (2,2).
3. Una capa de 120 neuronas totalmente conectadas.
4. Otra capa de 84 neuronas totalmente conectadas.
5. Una capa final en la que en número neuronas coincide con el número de clases de la base de datos utilizada.

2.1.7. Retropropagación

En 1986, *David Rumelhart*, *Geoffrey Hinton* y *Ronald Williams*, formalizaron un método ("*Backpropagation*") para que una red neuronal multicapa aprendiera la asociación que existe entre los patrones de entrada y las clases correspondientes [Reyes et al., 2016].

Este algoritmo consiste en un aprendizaje de un conjunto de pares de entradas-salida dados como ejemplo, empleando un ciclo de propagación-adaptación de dos fases:

1. Primera fase

   Se aplica un patrón de entrada como estímulo para la primera capa de las neuronas de la red, se va propagando a través de todas la capas superiores hasta generar una salida.

   Después, se compara el resultado obtenido en las neuronas de salida con la salida que se desea obtener y se calcula un valor del error para cada neurona de salida.

2. Segunda fase

   Estos errores se transmiten hacia atrás, partiendo desde la capa de salida, hacia todas las neuronas de la capa intermedia que contribuyan directamente a la salida. Este proceso se repite, capa por capa, hasta que todas las neuronas de la red hayan recibido un error que describa su aportación relativa al error total.

   Basándose en el valor del error recibido, se reajustan los pesos de conexión de cada neurona, de manera que en la siguiente vez que se presente el mismo patrón, la salida esté más cercana a la deseada.

La importancia del algoritmo de retropropagación consiste en su capacidad de auto adaptar los pesos de las neuronas de las capas intermedias para aprender la relación que existe ente un conjunto de patrones de entrada y sus salidas correspondientes.

2.2. Software

Existen diversos tipos de software, librerías o programas para el aprendizaje profundo. De esta amplia lista [Sof, 2017] se realizará la experimentación con *Keras*, *Theano* y *Tensorflow*.

En la (Tabla 2.1) se muestran algunas características de los softwares mencionados anteriormente.

Software	**Keras**	**Theano**	**Tensorflow**
Creador	François Chollet	Université de Montréal	Google Brain team
Open Source	Sí	Sí	Sí
Plataforma	Linux, Mac OS X, Windows	Plataforma cruzada	Linux, Mac OS X, Windows
Escrito en	Python	Python	C++, Python
Interfaz	Python	Python	Python, C/C++, Java, Go, R
Soporte de OpenMP	Sí	Sí	No
Soporte de Cuda	Sí	Sí	Sí
Redes Convolucionales	Sí	Sí	Sí

Tabla 2.1: Características de los tres softwares.

2.2.1. Theano

La última versión de *Theano* es la 0.9.0 (20 de Marzo, 2017).

Theano es una librería de *Python* que permite definir, optimizar y evaluar expresiones matemáticas, especialmente aquellas con matrices multidimensionales [Team, 2017].

Usando *Theano* es posible alcanzar velocidades que rivalizan implementaciones de C hechas a mano para los problemas que implican grandes cantidades de datos. Además, es capaz de usar eficientemente las capacidades computacionales de las GPUs.

Theano combina aspectos de un sistema de álgebra computarizada (CAS) con aspectos de un compilador de optimización. También, puede generar código C personalizado para muchas operaciones matemáticas.

Esta combinación de CAS con la optimización de la compilación es particularmente útil para tareas en las que las expresiones matemáticas complicadas se evalúan repetidamente y la velocidad de evaluación es crítica.

El compilador de *Theano* aplica muchas optimizaciones de complejidad variable a estas expresiones simbólicas. Estas optimizaciones incluyen, pero no se limitan a:

1. Uso de GPU para cálculos.
2. Fusión de subgrafos similares, para evitar el cálculo redundante.
3. Uso de aliasing de memoria para evitar el cálculo.
4. Fusión de bucle para sub-expresiones de elementos.

2.2.2. Tensorflow

Tensorflow es una librería de código abierto desarrollada por *Google*. La librería se puede utilizar tanto en programas escritos en código *Python* como en C++, aunque para este proyecto se ha optado por utilizar la versión de *Python*.

Desde el punto de vista de las funcionalidades Tensorflow es muy similar a Theano:

1. Permite construir y entrenar redes neuronales con distintos tipos de arquitecturas.
2. Es capaz de usar eficientemente las capacidades computacionales de las GPUs, lo cual consigue utilizando la librería *cuDNN* de *Nvidia* (Sección 2.2.4).

3. Incluye rutinas optimizadas para las operaciones que se realizan durante la creación y evaluación de una red neuronal.

En lo que a las diferencias respectan, la más notable es la falta de soporte para *OpenMP* (ejecución *multithreading*).

Por otro lado, *Tensorflow* se puede utilizar para desarrollar redes que se puedan ejecutar tanto en PCs y servidores como en dispositivos móviles.

Para más información sobre cómo utilizar Tensorflow visitar la página oficial [Abadi et al., 2015].

2.2.3. Keras

Keras es una API de redes neuronales escrita en *Python*, que proporciona un nivel de abstracción superior a las dos librerías vistas anteriormente. Los programas escritos en *Keras* no se ejecutan directamente, sino que son traducidos a otras APIs de redes neuronales como *TensorFlow*, *CNTK* o *Theano*.

Keras ofrece las siguientes ventajas:

- Facilidad de uso

  Al ofrecer un nivel de abstracción superior, *Keras* permite trabajar con redes neuronales sin un conocimiento extenso de las estructuras de datos subyacentes.

- Modularidad

  Los programas escritos en *Keras* siguen una estructura modular, es decir, la librería dispone de un conjunto de módulos autónomos y configurables que se pueden combinar para generar redes más complejas. Las ventajas de esta modularidad se aprecian con más claridad en el capítulo 3, en el que se presentan los programas realizados en este proyecto con *Keras*.

- Extensibilidad

  Además de los módulos incluidos por defecto también es posible desarrollar nuevos.

2.2.4. *NVIDIA cuDNN*

"NVIDIA CUDA® Deep Neural Network (cuDNN) es una librería de primitivas para redes neuronales profundas aceleradas mediante la GPU" [NVIDIA, 2017].

En este proyecto no se ha programado directamente con *cuDNN*, pero las librerías de redes neuronales con las que se ha trabajado lo utilizan para la ejecución en GPUs.

cuDNN proporciona implementaciones altamente optimizadas para rutinas como capas de convolución, agrupación, normalización y activación. *cuDNN* es parte del *SDK* de aprendizaje profundo de *NVIDIA*.

2.3. Base de Datos

En el transcurso de proyecto se han empleado tres bases de datos de diferentes ámbitos de aplicación referentes en la comunidad internacional. A continuación, se exponen las características de cada una.

2.3.1. MNIST

La base de datos MNIST es una gran base de datos de dígitos manuscrita que se utiliza comúnmente para el entrenamiento de varios sistemas de procesamiento de imágenes. Esta base de datos también es utilizada para la formación y pruebas en el campo del aprendizaje automático.

La base de datos MNIST contiene un conjunto de entrenamiento de 60.000 ejemplos, y otro conjunto de prueba de 10.000 ejemplos, siendo entre ellos conjuntos disjuntos.

Los dígitos están normalizados en imágenes de tamaño fijo de 28×28 en escala de grises.

El MNIST está formado por 10 clases, esto es, las entradas representan un dígito de 0 a 9 como se muestra en la figura 2.10.

Figura 2.10: Muestra de la base de datos MNIST

Es una buena base de datos para personas que quieren probar técnicas de aprendizaje y métodos de reconocimiento de patrones en datos del mundo real.

En el sitio web oficial [LeCun et al., 1998b] se presentan más detalles sobre esta gran base de datos.

2.3.2. CIFAR-10

El conjunto de datos CIFAR-10 consta de 60.000 imágenes de 32×32 píxeles en color. Estas imágenes están organizadas en 10 clases, con 6.000 imágenes por cada una de ellas.

De las 60.000 imágenes, 50.000 imágenes son de entrenamiento y 10.000 imágenes de test. El conjunto de datos se divide de la siguiente manera:

- Cinco lotes de entrenamiento, cada uno con 10.000 imágenes. Estos lotes contienen las imágenes en orden aleatorio y algunos lotes de entrenamiento pueden contener más imágenes de una clase que otra. En total, sumando las imágenes de todos los lotes de entrenamiento hay 5.000 imágenes de cada clase.
- Un lote de prueba que contiene exactamente 1.000 imágenes seleccionadas aleatoriamente de cada clase.

En la figura 2.11, se muestra un ejemplo sobre el conjunto de datos con 10 imágenes aleatorias de cada uno:

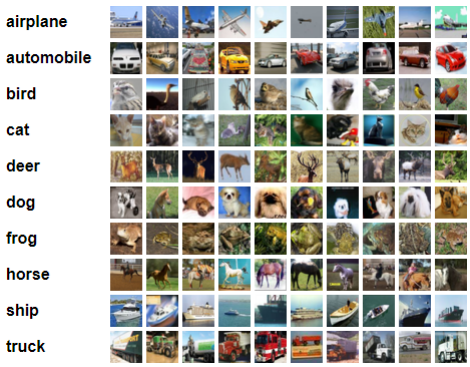

Figura 2.11: Muestra de la base de datos CIFAR-10

Las clases son completamente exclusivas, es decir, no hay superposición entre ellas. La especificación completa de la base de datos se puede consultar en la página oficial de la misma [Krizhevsky et al., 2009].

2.3.3. Statlog (*Shuttle*)

La base de datos Statlog (*Shuttle*) está formada de 58.000 imágenes, donde 43.500 forman el conjunto de entrenamiento y 14.500 el conjunto de prueba.

En la base de datos se pueden distinguir imágenes pertenecientes a 7 diferentes clases, pero aproximadamente el 80 % de los datos pertenecen a la clase 1. Por lo tanto, la precisión por defecto es de alrededor del 80 %.

El conjunto de datos *Shuttle* contiene 9 parámetros numéricos enteros, donde el último es la clase que ha sido codificada la imagen.

Para más información visitar el sitio web oficial [Lichman, 2013].

3. CAPÍTULO

Implementación

Para realizar las pruebas experimentales que se presentan en el capítulo 4 ha sido necesario realizar la implementación de dos tipos de redes neuronales en las tres librerías presentadas.

En este capítulo se describen las partes más relevantes de las implementaciones realizadas.

3.1. Redes Neuronales implementadas

En este proyecto se ha trabajado con dos topologías muy comunes: MLP (*Multi Layer Perceptron*) y CNN (*Convolutional Neural Network*).

3.1.1. Multi layer perceptron (MLP)

Las redes MLP son redes totalmente conectadas, es decir, las neuronas de capas adyacentes están todas conectadas entre sí. La red MLP desarrollada en este proyecto se ha utilizado para trabajar con la base de datos Statlog (*Shuttle*) presentado en la sección 2.3.3.

El programa que genera las redes neuronales recibe una serie de parámetros de entrada. Estos parámetros se pueden utilizar para controlar las siguientes características de la red:

- Parámetro 1

Indica el número de capas ocultas totalmente conectadas que poseerá la red.

- Parámetro 2

  Señala el número de neuronas que contendrá cada capa oculta.

La función de activación utilizada en la red implementada es la función *ReLU*.

La última capa, siendo la salida de la red, será una capa *Softmax* de 7 neuronas.

Los *bias* se inicializan con un valor de 0, y los pesos de las conexiones de la red irán sujetas a una distribución normal con media 0 y desviación típica $\sqrt{\frac{2}{n_l}}$, donde n_l es la longitud de entrada de datos en la capa l.

La manera en la que se inicializan los pesos y los *bias* está recogida en la publicación *"Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification"* [He et al., 2015].

3.1.2. Red convolucional (CNN)

El segundo tipo de red que se ha implementado es una red neuronal convolucional. Como se ha mencionado anteriormente en el apartado 2.1.6.4, este tipo de redes son adecuadas para el reconocimiento de imágenes en 2D.

La red desarrollada está inspirada en *LeNet-5* (Sección 2.1.6.6) y se ha utilizado para procesar la base de datos *Cifar-10* (Sección 2.3.2).

De manera similar a la red MLP, el programa implementado genera redes neuronales de diversos tamaños en base a dos parámetros de entrada:

- Parámetro 1

  Indica el número de capas convolucionales que poseerá la red desde una capa hasta un máximo de dos.

- Parámetro 2

  Señala el número de mapas de características por color que contendrá cada capa convolucional, esto es, como las imágenes de entrada tienes tres colores (RGB), cada capa estará formada por 3 $\times$ Parámetro 2.

La función de activación utilizada es la función *ReLU*, y para el agrupamiento de capas se ha utilizado un *max-pooling* con una ventana de dimensiones $(2,2)$.

La última capa convolucional estará totalmente conectada a una capa de 120 neuronas. Esta capa se hallará totalmente conectada a otra capa de 84 neuronas.

Finalmente, la última capa de la red estará formada por el mismo número de clases que la base de datos, es decir, 10 neuronas. Esta capa será de tipo *Softmax*.

Tanto la incialización de los *bias* como los pesos de las conexiones de la red se realizarán siguiendo el mismo criterio que en la red MLP.

3.2. Creación de las Redes Neuronales

En este apartado se presenta la manera en la que se han creado las redes neuronales descritas en la sección anterior.

La manera en la que se realiza la creación varía en función de la librería utilizada (*Theano*, *Tensorflow* o *Keras*) aunque los conceptos subyacentes son los mismos.

El código que se presenta sigue el estilo de los programas de ejemplo estudiados durante la fase de aprendizaje, pero en algunos casos puede haber formas alternativas para implementar las redes.

3.2.1. Creación en Theano

Primeramente, es necesario importar las funciones de activación que existen en las librerías de Theano y definir las que no estén.

```
from theano.tensor.nnet import softmax
def ReLU(z):
    return T.maximum(0.0, z)
```

Después, se detallan las clases de los diferentes tipo de capas que se van a utilizar, especificando la inicialización (pesos, *bias*, activaciones, etc. que formarán esta capa) y definiendo una función de salida.

1. Capa totalmente conectada

```
class FullyConnectedLayer(object):
    # Inicialización de la capa
    def __init__(self, n_in, n_out, activation_fn=ReLU):
        self.n_in = n_in
        self.n_out = n_out
        self.activation_fn = activation_fn

        # Inicializar pesos y bias
        self.w = theano.shared(
            np.asarray(
                np.random.normal(loc=0.0, scale=np.sqrt(2.0/n_in), size=(n_in, n_out)),
                dtype=theano.config.floatX),
            name='w', borrow=True)
        self.b = theano.shared(
            np.asarray(
                        np.zeros(n_out,),
                        dtype=theano.config.floatX),
            name='b', borrow=True)
        self.params = [self.w, self.b]

    # Definición de la salida
    def set_inpt(self, inpt, mini_batch_size):
        self.inpt = inpt.reshape((mini_batch_size, self.n_in))
        self.output = self.activation_fn(T.dot(self.inpt, self.w) + self.b)
        self.y_out = T.argmax(self.output, axis=1)
```

2. Capa convolucional

```
class FullyConnectedLayer(object):
    # Inicialización de la capa
    def __init__(self, filter_shape, input_shape, poolsize=(2, 2),
                 activation_fn=ReLU):
        self.filter_shape = filter_shape
        self.input_shape = input_shape
        self.poolsize = poolsize
        self.activation_fn=activation_fn

        # Inicializar pesos y bias
        n_in = input_shape[2]*input_shape[2]*input_shape[1]
        self.w = theano.shared(
            np.asarray(
                np.random.normal(loc=0, scale=np.sqrt(2.0/n_in), size=filter_shape),
                dtype=theano.config.floatX),
            borrow=True)
        self.b = theano.shared(
            np.asarray(
                 np.zeros((filter_shape[0],)),
                dtype=theano.config.floatX),
```

```
            borrow=True)
        self.params = [self.w, self.b]

    # Definición de la salida
    def set_inpt(self, inpt, mini_batch_size):
        self.inpt = inpt.reshape(self.input_shape)
        conv_out = conv.conv2d(
            input=self.inpt, filters=self.w, filter_shape=self.filter_shape,
            input_shape=self.input_shape)
        pooled_out = pool.pool_2d(
            input=conv_out, ws=self.poolsize, ignore_border=True)
        self.output = self.activation_fn(
            pooled_out + self.b.dimshuffle('x', 0, 'x', 'x'))
```

3. Capa Softmax

```
class SoftmaxLayer(object):
    # Inicialización de la capa
    def __init__(self, n_in, n_out):
        self.n_in = n_in
        self.n_out = n_out

        # Inicializar pesos y bias
        self.w = theano.shared(
            np.random.normal(loc=0.0, scale=np.sqrt(2.0/n_in), size=(n_in, n_out)),
            #np.zeros((n_in, n_out), dtype=theano.config.floatX),
            name='w', borrow=True)
        self.b = theano.shared(
            np.zeros((n_out,), dtype=theano.config.floatX),
            name='b', borrow=True)
        self.params = [self.w, self.b]

    # Definición de la salida
    def set_inpt(self, inpt, mini_batch_size):
        self.inpt = inpt.reshape((mini_batch_size, self.n_in))
        self.output = softmax(T.dot(self.inpt, self.w) + self.b)
        self.y_out = T.argmax(self.output, axis=1)

    # Definición del coste
    def cost(self, net):
        return -T.mean(T.log(self.y_out)[T.arange(net.y.shape[0]), net.y])

    # Definición del accuracy del mini-batch
    def accuracy(self, y):
        return T.mean(T.eq(y, self.y_out))
```

Posteriormente, se crea la clase *Network* siendo ésta la encargada de crear la red neuronal completa.

```
# type --> variable binaria que indica si se trata de una red CNN (0) o una red MLP (1)
class Network(object):
    def __init__(self, layers, mini_batch_size, type):
        self.layers = layers
        self.mini_batch_size = mini_batch_size
        self.params = [param for layer in self.layers for param in layer.params]
        if type == 1:
             self.x = T.matrix("x")
        else:
             self.x = T.tensor4("x")
        self.y = T.ivector("y")
        init_layer = self.layers[0]
        init_layer.set_inpt(self.x, self.x, self.mini_batch_size)
        for j in range(1, len(self.layers)):
            prev_layer, layer  = self.layers[j-1], self.layers[j]
            layer.set_inpt(
                prev_layer.output, prev_layer.output, self.mini_batch_size)
        self.output = self.layers[-1].output
```

Una vez definidas las capas y la constructora de la red, se genera cada capa y se almacenan en un *array*.

```
array = [FullyConnectedLayer(n_in=8, n_out=int(sys.argv[2]))]
for i in range(int(sys.argv[1])-1):
        array.extend([FullyConnectedLayer(n_in=int(sys.argv[2]), n_out=int(sys.argv[2]))])
array.extend([SoftmaxLayer(n_in=int(sys.argv[2]), n_out=7)])
```

Finalmente, se llama a la constructora de la red.

```
net = Network(array, mini_batch_size,1)
```

3.2.2. Creación en Tensorflow

Del mismo modo que en la implementación realizada en *Theano*, *Tensorflow* también requiere definir una a una las diferentes capas de la red y especificar los pesos, los *bias* y la función de activación de sus neuronas.

El primer paso para generar la red neuronal en *Tensorflow* es definir las funciones encargadas de inicializar las variables de cada capa.

```
# Inicializador de los pesos en redes CNN
def weight_variable_conv(shape, valor):
    initial = tf.random_normal(shape, stddev=np.sqrt(2 /(valor*valor*shape[2])))
    return tf.Variable(initial)

# Inicializador de los pesos en redes MLP
def weight_variable(shape):
    initial = tf.random_normal(shape, stddev=np.sqrt(2 /(shape[0])))
    return tf.Variable(initial)

# Inicializador de los bias
def bias_variable(shape):
    initial = np.zeros(shape,dtype=np.float32)
    return tf.Variable(initial)

def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='VALID')

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
```

Con las funciones definidas, se pueden ir generando paso a paso cada capa.

- Capa convolucional

```
# X -> características de entrada
# Y -> características de salida
# Z -> dimensión de la entrada
# D -> entrada de datos (Salida de la capa anterior) a la capa actual

# Pesos de la capa convolucional
W_conv1 = weight_variable_conv([5, 5, X, Y, Z)

# Bias de la capa convolucional
b_conv1 = bias_variable([Y])

# Función de Activación de la capa convolucional
h_conv1 = tf.nn.relu(conv2d(D, W_conv1) + b_conv1)

# Pooling de la capa convolucional
h_pool1 = max_pool_2x2(h_conv1)
```

Si se quisiese conectar la salida de una capa CNN con la entrada de una capa MLP se debe reestructurar la salida del siguiente modo:

```
# Dim -> dimensiones de la entrada de datos
h_pool2_flat = tf.reshape(h_pool1, [Dim])
```

- Capa totalmente conectada

  La creación de una capa totalmente conectada sigue el mismo proceso que una capa convolucional, con la diferencia de que no se realizan ni la convolución ni el *pooling*.

```
# n_in -> neuronas de la capa anteior
# n_out -> neuronas de la capa actual
# D -> entrada de datos (Salida de la capa anterior) a la capa actual

# Pesos de la capa MLP
W_fc2 = weight_variable([n_in, n_out])

# Bias de la capa MLP
b_fc2 = bias_variable([n_out])

# Función de Activación de la capa MLP
h_fc2 = tf.nn.relu(tf.matmul(D, W_fc2) + b_fc2)
```

En la última capa se calcula la salida de cada neurona y después, se procesa como una salida *Softmax* aplicando la correspondiente función de Tensorflow.

```
# D -> entrada de datos (Salida de la capa anterior) a la capa actual
# y -> salida deseada de la red

# Función de Activación de la última capa MLP
y_ = tf.matmul(D, W_fc2) + b_fc2

# Aplicar la salida softmax con las librerías de tf.
softmax = tf.contrib.layers.softmax(logits=y_)
```

3.2.3. Creación en Keras

Como se ha mencionado anteriormente, *Keras* es una API de alto nivel que permite al usuario con un número reducido de acciones crear una red neuronal.

La creación de las redes neuronales sigue el siguiente patrón:

- Inicialmente, se crea un modelo secuencial vacío.

```
model = Sequential()
```

- Se añade al modelo la primera capa oculta, indicando el tipo de capa y añadiendo los parámetros necesarios para cada tipo de capa. Una vez añadida la capa se especifica su función de activación.

  - En el caso de una capa totalmente conectada sería de la siguiente manera.

```
model.add(Dense(int(sys.argv[2]), input_shape=(x_train.shape[1:]),
    kernel_initializer=keras.initializers.RandomNormal(mean=0.0, stddev=np.sqrt(2
    /(8)), seed=None)))
model.add(Activation('relu'))
```

  - Si la capa creada fuese una convolucional y se desea añadir el *pooling*.

```
model.add(Conv2D(int(sys.argv[2])*3, (5, 5), padding='valid', input_shape=x_train.
    shape[1:], kernel_initializer=keras.initializers.RandomNormal(mean=0.0, stddev=
    np.sqrt(2 /(32*32*3)), seed=None)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
```

- Después, se crean tantas capas ocultas como se puedan o se deseen, prescindiendo del parámetro *"input_shape"*.

```
for i in range(int(sys.argv[1])-1):
    model.add(Dense(int(sys.argv[2]), kernel_initializer=keras.initializers.RandomNormal(
     mean=0.0, stddev=np.sqrt(2 /(int(sys.argv[2]))), seed=None)))
    model.add(Activation('relu'))
```

- En el caso de las redes convolucionales antes de conectar una capa convolucional (2D) con una de una dimensión es necesario utilizar la siguiente función.

```
model.add(Flatten())
```

- Finalmente, se añade la capa final cambiando la función de activación por *Softmax*.

```
model.add(Dense(num_classes, kernel_initializer=keras.initializers.RandomNormal(mean=0.0,
    stddev=np.sqrt(2 /(int(sys.argv[2]))), seed=None)))
model.add(Activation('softmax'))
```

3.3. Entrenamiento de las Redes Neuronales

Una vez creadas la redes neuronales, el siguiente paso consiste en entrenarlas haciendo uso de las correspondientes bases de datos:

- Statlog (*Shuttle*) para la red MLP
- CIFAR-10 para CNN

En este apartado se presenta el código que implementa el proceso de entrenamiento en cada una de las librerías utilizadas.

3.3.1. Entrenamiento en Theano

Dentro de la clase *Network* se crea una función *SGD* donde se preparan los parámetros y se realiza el entrenamiento.

Primeramente, se definen los conjuntos de entrenamiento, de testeo y el número de iteraciones a realizar al entrenar.

```
def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data, lmbda=0.0):
        training_x, training_y = training_data
        test_x, test_y = test_data

        num_training_batches = size(training_data)/mini_batch_size
        num_test_batches = size(test_data)/mini_batch_size
```

Después, se definen la función de coste, los gradientes simbólicos y las actualizaciones.

```
        cost = self.layers[-1].cost(self)
        grads = T.grad(cost, self.params)
        updates = [(param, param-eta*grad)
                   for param, grad in zip(self.params, grads)]
```

Una vez definidos los parámetros anteriores, se especifican las funciones para los lotes de datos de cada iteración (*mini-batch*).

Estos *mini-batch* se utilizan tanto para entrenar la red como para calcular la precisión de las pruebas.

```
        i = T.lscalar()
        train_mb = theano.function(
            [i], cost, updates=updates,
            givens={
                self.x:
                training_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size],
                self.y:
                training_y[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
            })
        test_mb_accuracy = theano.function(
            [i], self.layers[-1].accuracy(self.y),
            givens={
                self.x:
                test_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size],
                self.y:
                test_y[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
            })
```

Una vez preparado todo, se repite el entrenamiento un número determinado de épocas (en este caso *epochs*=10).

Por cada iteración de *minibatch_index*, se escoge datos del tamaño de *mini-batch*, se entrena la red y se calcula el coste.

Cada *num_training_batches*, se verifica el estado de la red calculando el *accuracy* momentáneo de la red.

Una vez completadas todas las épocas (*epochs*), se muestra el *accuracy* final de la red.

```
        for epoch in range(epochs):
            print("Epoch {}/{}".format(epoch+1,epochs))
            for minibatch_index in range(int(num_training_batches)):
                iteration = int(num_training_batches)*epoch+minibatch_index
                cost_ij = train_mb(minibatch_index)

                if (iteration+1) % int(num_training_batches) == 0:
                    test_accuracy = np.mean(
                         [test_mb_accuracy(j) for j in range(int(num_test_batches))])
                    print('The corresponding test accuracy in interation {0}/{1} is {2:.2%} en
    {3} segundos'.format(int(iteration+1),int(num_training_batches),test_accuracy,int(fin_for-
    inic_for)))
        print("Finished training network.")
```

3.3.2. Entrenamiento en Tensorflow

De una manera similar a *Theano*, primero se define la función de coste, el optimizador, el número de lotes para el entrenamiento y el objeto *Tensor* para la inicialización de las variables.

```
cost = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits=matriz_red[last_layer
    ][2], labels=y))

optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate).minimize(cost)

init = tf.global_variables_initializer()

max_arr=np.zeros(y_test.shape[0])

total_batch = int(x_train.shape[0] / batch_size)
```

A continuación, se crea un objeto *Session* de *Tensorflow*, el cual encapsula el entorno en el que se ejecutan los objetos de tipo *Operation* y se evalúan los objetos *Tensor*.

Del mismo modo que en *Theano*, el entrenamiento se repite un número determinado de épocas (*epochs*).

En cada iteración de *total_batch* se seleccionan datos aleatorios para el entrenamiento, se entrena la red, se calcula el coste y se calcula el *accuracy*.

Una vez finalizadas todas las iteraciones, se muestra el *accuracy* final de la red.

```
with tf.Session() as sess:
    sess.run(init)

    for epoch in range(epochs):
        avg_cost = 0

        orden = np.arange(x_train.shape[0])
        np.random.shuffle(orden)
        _images = x_train[orden]
        _labels = y_train[orden]

        for i in range(total_batch):
          batch_xs, batch_ys = next_batch(batch_size, i, _images, _labels)
          _, iteration_cost = sess.run([optimizer, cost], feed_dict = {x: batch_xs, y: batch_ys})
          avg_cost += iteration_cost / total_batch

        print('Epoch:', '%d' % (epoch + 1), 'cost=', '{:.9f}'.format(avg_cost))
        resultado=softmax.eval({x: x_test})
```

```
        for i in range(y_test.shape[0]):
          max_arr[i]=int(np.argmax(resultado[i]))

        prediction = tf.equal(max_arr.astype(np.int32), y_test)
        acc = tf.reduce_mean(tf.cast(prediction, 'float'))

        print('Accuracy:', acc.eval({x: x_test, y: y_test}))
```

3.3.3. Entrenamiento en Keras

Al igual que sucedía en la creación de redes neuronales, implementar el entrenamiento es mucho más sencillo en *Keras* que en las otras herramientas.

Primero se decide el optimizador (en este caso, "Descenso de gradiente estocástico" con tasa de aprendizaje 10^{-3}).

```
opt = keras.optimizers.SGD(lr=1e-3)
```

Después, se compila la red creada anteriormente indicando la métrica utilizada para evaluar la bondad del sistema (para este proyecto se ha elegido el *accuracy*), la función de pérdida a usar y el optimizador definido previamente.

```
model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])
```

Para finalizar, se entrena la red llamando a la función *fit* con los siguientes datos:

- Los datos de entrenamiento
- El tamaño de lote
- Número de épocas a realizar en el entrenamiento
- Los datos de testeo

```
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          validation_data=(x_test, y_test),
          shuffle=True)
```

4. CAPÍTULO

Experimentación

En este capítulo se presentan las pruebas realizadas y los resultados obtenidos en ellas. Estas pruebas han consistido en la ejecución de los programas presentados en el capítulo anterior en dos máquinas diferentes.

La especificación de estas máquinas se describe en la sección 4.1.2 de este capítulo.

4.1. Descripción de las pruebas

4.1.1. Objetivo de las pruebas

La meta de la experimentación es observar el comportamiento así como el rendimiento de las tres librerías (*Theano*, *Tensorflow* y *Keras*) para diferentes tipos de redes neuronales (MLP y CNN).

Para la comparativa se ha creado un banco de pruebas que evalúa el rendimiento de estas herramientas con redes de distintos tamaños.

Con el fin de automatizar la ejecución de las pruebas se han realizados una serie de *scripts* en *shell*.

4.1.2. Especificación de las máquinas

La experimentación se ha realizado en dos máquinas para observar el funcionamiento de las librerías ejecutándolas sobre diferente *hardware*.

Una de las máquinas se ha utilizado para ejecutar los programas sobre una GPU y la otra para ejecutarlos sobre una CPU con diferentes números *threads*.

4.1.2.1. Máquina 1

Esta computadora se ha encargado de realizar las ejecuciones sobre una GPU, concretamente, en una ***NVIDIA GeForce GTX TITAN Black***.

En las siguientes tablas se muestran las especificaciones de la GPU (Tabla 4.1), los detalles de su memoria (Tabla 4.2) y algunas características adicionales de la tarjeta gráfica (Tabla 4.3).

Núcleos CUDA	2880
Frecuencia de reloj normal	889 MHz
Frecuencia acelerada	980 MHz
Tasa de relleno de texturas	213 GigaTexels/s

Tabla 4.1: Especificaciones de la *NVIDIA GeForce GTX TITAN Black*

Frecuencia de la memoria (Gbps)	7,0
Cantidad de memoria	6144 MB
Interfaz de memoria	384-bit GDDR5
Ancho de banda máx.	336 GB/s

Tabla 4.2: Memoria de la *NVIDIA GeForce GTX TITAN Black*

Entorno de programación	*CUDA*
Temperatura máxima	95 ℃
Consumo	250 W
Requisitos mínimos de potencia	600 W

Tabla 4.3: Características adicionales de la *NVIDIA GeForce GTX TITAN Black*

Para más información sobre la GPU visitar la página oficial [NVIDIA, 2017].

4.1.2.2. Máquina 2

Esta máquina ejecuta los programas en la CPU, ***AMD Opteron(tm) Processor 6168***, la cual dispone de 4 núcleos.

En la tabla 4.4 se presenta la información general sobre este procesador.

Tipo	CPU/Microprocesador
Familia	AMD Opteron 6100 series
Número de Modelo	6168
Frecuencia	1900 MHz
Velocidad de Bus	3200 MHz
Fecha de Presentación	29 de Marzo del 2010

Tabla 4.4: Información general sobre *AMD Opteron(tm) Processor 6168*

La tabla 4.5 contiene información detallada sobre la arquitectura del procesador.

Micro-Arquitectura	K10
Arquitectura	*x*86_64
Orden de bytes	Little Endian
Tamaño de caché de nivel 1	$12 \times 64KB$ Cachés de instrucciones asociativas de conjuntos de 2 vías $12 \times 64KB$ Cachés de datos asociativos conjuntos de 2 vías
Tamaño de caché de nivel 2	$12 \times 512KB$ Cachés exclusivos asociativos de 16 vías
Tamaño de caché de nivel 3	$2 \times 6MB$ Cachés exclusivos compartidos
Latencia del caché	3 ns (L1 caché)
Número de núcleos de la CPU	12
Número de threads	12
Multiprocesamiento	Hasta 4 procesadores

Tabla 4.5: Arquitectura / Micro-Arquitectura de *AMD Opteron(tm) Processor 6168*

La especificación completa de la CPU se puede consultar en la siguiente entrada de la bibliografía [AMD, 2017].

4.1.3. Casos de prueba

Como se ha mencionado anteriormente, para evaluar el rendimiento de las librerías se han realizado pruebas con redes neuronales de diferentes tipos y tamaños.

Dado que con la GPU se obtienen tiempos de ejecución mucho menores que con la CPU, los casos de prueba en la tarjeta gráfica abarcan redes de mayores dimensiones.

Este motivo también ha influido en el número de repeticiones realizadas por cada prueba: 5 en el caso de la GPU y 2 en el caso de la CPU.

4.1.3.1. Parámetros utilizados para la red MLP

- Nº de capas ocultas: 2, 4, 8, 16.

  En ejecuciones realizadas sobre la CPU con *multithreading* se han omitido las pruebas con 16 capas ocultas, ya que los tiempos de ejecución eran muy elevados.

- Nº de neuronas por capa: 25, 50, 100, 200.

  En las ejecuciones realizadas sobre la GPU también se ha probado con 400 y 1.600 neuronas por capa.

4.1.3.2. Parámetros utilizados para la CNN

- Nº de capas ocultas: 1, 2.

- Nº mapas de características por capa: 2, 4, 8, 16, 32.

  En las ejecuciones realizadas sobre la GPU también se ha probado a utilizar 64 mapas de características por capa.

  En ejecuciones realizadas sobre la CPU con *multithreading* se han omitido las pruebas con 32 mapas de características debido a tiempos de ejecución elevados.

El banco de pruebas está formado por todas la combinaciones posibles de los parámetros descritos. En total, se han realizado 1.408 ejecuciones.

4.2. Resultados obtenidos

En esta sección se presentan los resultados obtenidos en la fase de experimentación. Estos resultados están representados mediante gráficas que muestran los tiempos de ejecución de las pruebas de mayor interés.

De este modo se puede analizar de forma visual el rendimiento de cada herramienta, que será inversamente proporcional al tiempo de ejecución obtenido.

4.2.1. Nomenclatura utilizada en las gráficas

Para representar gráficamente los resultados más destacables de las pruebas realizadas se ha hecho uso de distintos tipos de líneas, formas y colores.

La convención seguida en el uso de estos elementos ha sido la siguiente:

- Tipos de líneas
  - **Línea continua**: ejecuciones realizadas directamente sobre *Tensorflow* o *Theano*.
  - **Línea discontinua**: ejecuciones realizadas con *Keras* utilizando cualquiera de los dos *backends*.
- Formas
  - **Círculo**: ejecuciones realizadas en *Theano* (o *Keras* con *backend* de *Theano*).
  - **Cuadrado**: ejecuciones realizadas en *Tensorflow* (o *Keras* con *backend* de *Tensorflow*).
- Colores
  - **Gris**: ejecuciones realizadas con 8 capas en el caso de MLP o con 1 capa en el caso de CNN. Además de utilizar el color gris, estos casos se indican también mediante el uso de formas vacías.
  - **Negro**: ejecuciones realizadas con 16 capas en el caso de MLP o con 2 capas en el caso de CNN. Además de utilizar el color negro, estos casos se indican también mediante el uso de formas rellenas.

Los nombres de las librerías están indicadas de forma abreviada (Tabla 4.6) acompañados del número de capas ocultas utilizadas.

Término	Significado
th	*Theano*
tf	*Tensorflow*
kth	Keras con el *backend* de *Theano*
ktf	Keras con el *backend* de *Tensorflow*

Tabla 4.6: Abreviaturas utilizadas en las gráficas

4.2.2. Pruebas realizadas sobre redes MLP

4.2.2.1. Estudio del rendimiento ejecutando en la GPU

La figura 4.1 muestra los tiempos de ejecución por época obtenidos durante la fase de entrenamiento de una red MLP con *Theano* y *Tensorflow* (ejecutado en la GPU).

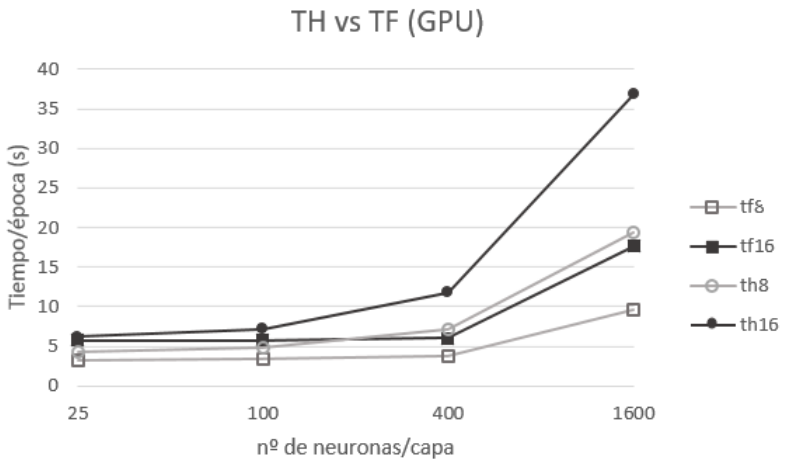

Figura 4.1: Tiempos de ejecución de una red MLP con *Theano* y *Tensorflow* (ejecutado en la GPU)

En las pruebas en las que el número de neuronas es pequeño la GPU parece estar infrautilizada, ya que hasta que no se superan las 400 neuronas por capa (aproximadamente) no hay variaciones notables en los tiempos.

Una vez se alcanza esta cifra, además de aumentar los tiempos de ejecución, también empiezan a hacerse más aparentes las diferencias entre las dos librerías. Concretamente, cuando el número de neuronas por capa asciende a 1.600, *Tensorflow* es hasta 2 veces más rápido que *Theano*.

Por otro lado, una ventaja de ejecutar en una GPU es la paralelización a gran escala.

Sin embargo, no todo el proceso se beneficia de igual manera de esta paralelización:

- Durante el proceso de *backpropagation* las capas tienen que recorrerse de forma secuencial, ya que las correcciones en cada una de ellas dependen de la capa anterior.
- En cada una de las capas, la corrección de los pesos y *bias* de cada neurona se puede realizar de forma paralela.

En la figura 4.2 se muestran los tiempos de ejecución por época obtenidos durante la fase de entrenamiento de una red MLP utilizando dos posibles *backends* de *Keras* (ejecutado en la GPU).

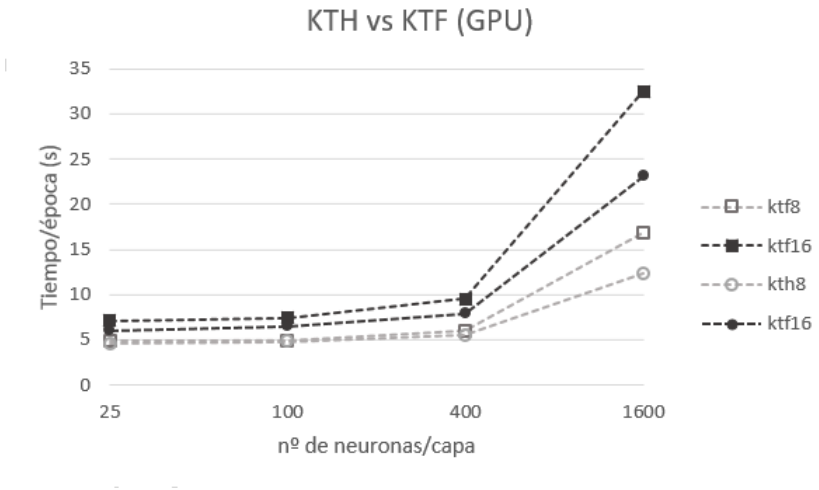

Figura 4.2: Tiempos de ejecución de una red MLP con *Keras* (*backend Theano*) y *Keras* (*backend Tensorflow*) (ejecutado en la GPU)

El aspecto más destacable de esta comparativa respecto a la anterior es que, a pesar de que por debajo *Keras* está utilizando las mismas herramientas, el resultado es el opuesto (el tiempo de ejecución de *Theano* es menor que el de *Tensorflow*).

Esto es debido a que *Keras* no se limita solo a traducir, sino que también aplica sus propias optimizaciones. Es posible que estas optimizaciones estén dando una ventaja a *Theano* para redes MLP.

La siguiente comparativa (Figura 4.3) presenta los tiempos de ejecución por época obtenidos utilizando *Theano* y *Keras* con *backend* de *Theano* (ejecutado en la GPU).

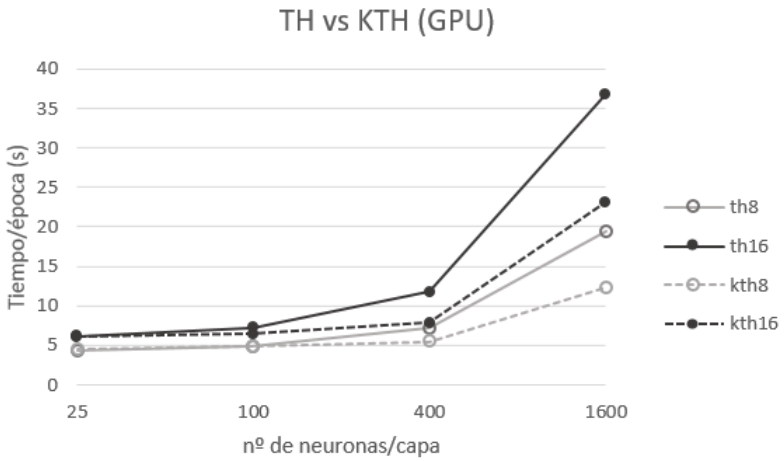

Figura 4.3: Tiempos de ejecución de una red MLP con *Theano* y *Keras* (*backend Theano*) (ejecutado en la GPU)

En esta comparativa se aprecia claramente lo mencionado en la gráfica anterior: el programa implementado con *Keras* (*backend Theano*) rinde mucho mejor que el programa directamente implementado en *Theano*. Esto puede deberse a que *Keras* es capaz de optimizar el código de *Theano* para este tipo de redes.

Finalmente, en la figura 4.4 se muestra una comparativa similar entre *Tensorflow* y *Keras* con *backend* de *Tensorflow*.

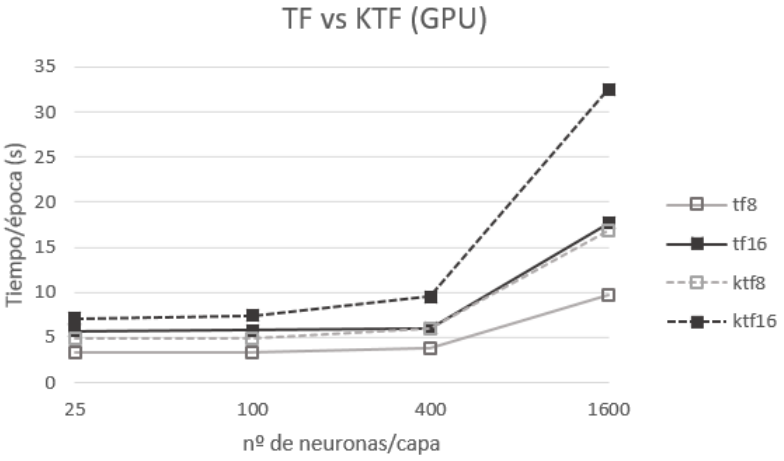

Figura 4.4: Tiempos de ejecución de una red MLP con *Tensorflow* y *Keras* (*backend Tensorflow*) (ejecutado en la GPU)

En este caso sucede lo contrario que en el anterior: el haber realizado el programa en *Keras* ha supuesto un perjuicio para el rendimiento de la red neuronal.

Este perjuicio se produce independientemente del número de neuronas por capa, pero es especialmente significativo cuando este número es elevado. Por ejemplo, en una red MLP con 1.600 neuronas por capa la implementación realizada directamente en *Tensorflow* es casi el doble de rápida que la realizada en *Keras*.

4.2.2.2. Estudio del rendimiento ejecutando en la CPU

En este apartado se muestran comparativas similares a las anteriores, con la diferencia de que en este caso se ejecutan sobre la CPU (utilizando un único *thread*). Ejecutar sobre la CPU supone un aumento significativo de los tiempos de ejecución, por lo que en las gráficas que se presentan a continuación se ha utilizado una escala logarítmica (base 10) para representar el tiempo.

En la figura 4.5 se muestran los tiempos de ejecución por época obtenidos durante la fase de entrenamiento de una red MLP utilizando *Theano* y *Tensorflow*.

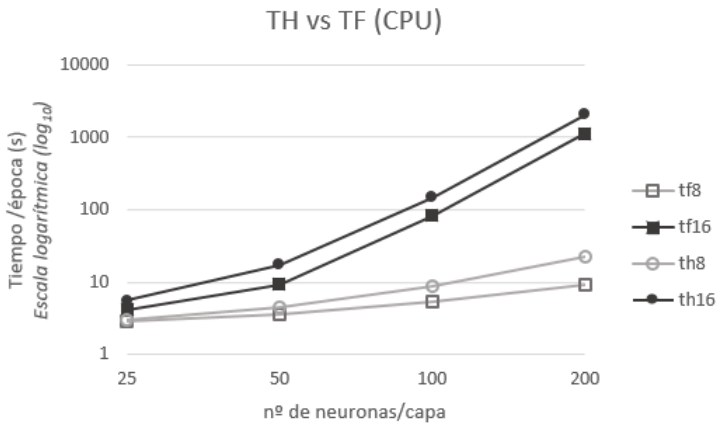

Figura 4.5: Tiempos de ejecución de una red MLP con *Theano* y *Tensorflow* (ejecutado en la CPU)

Al igual que sucedía ejecutando sobre la GPU, el programa implementado en *Tensorflow* tiene mejores resultados que el programa escrito con *Theano*. Esta ventaja se mantiene para cualquier número de neuronas por capa.

Otra observación a destacar es que, ejecutando sobre la CPU, el número de capas tiene un impacto mucho mayor en el rendimiento que el que tenía utilizando la GPU. Esto podría deberse a que cuando la red tiene un número elevado de capas se satura la memoria RAM y se tiene que hacer uso del disco (*Swap*).

En el siguiente gráfico (Figura 4.6) se muestran los tiempos de ejecución por época obtenidos durante la fase de entrenamiento de una red MLP utilizando dos *backends* distintos de *Keras* (*Theano* y *Tensorflow*).

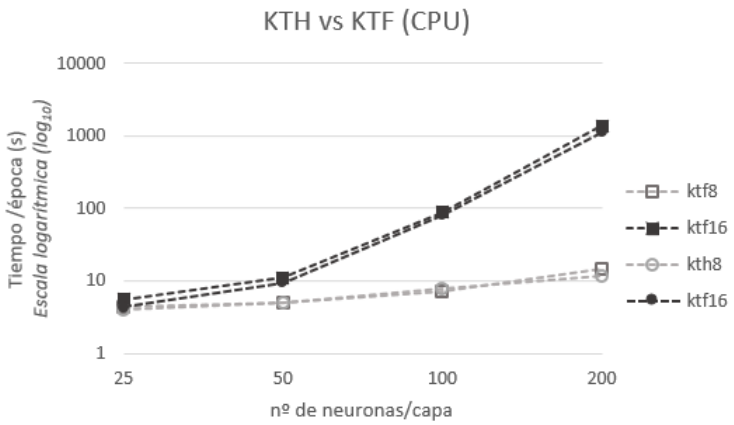

Figura 4.6: Tiempos de ejecución de una red MLP con *Keras* (*backend Theano*) y *Keras* (*backend Tensorflow*) (ejecutado en la CPU)

Al ejecutar con *Keras* las diferencias vistas en el caso anterior (Figura 4.5) se reducen.

En el caso de la GPU (Figura 4.2) la ejecución sobre el *backend* de *Theano* era notablemente mejor que la ejecución sobre el *backend* de *Tensorflow*.

Esto se debía principalmente a dos motivos. Por un lado, utilizar *Keras* con *backend* de *Theano* producía una mejoría respecto a utilizar *Theano* directamente. Por otro lado, utilizar *Keras* con *backend* de *Tensorflow* resultaba en una pérdida de rendimiento.

En el caso de la CPU ambos casos rinden de forma similar porque, a pesar de que se sigue produciendo una mejoría al utilizar *Keras* con *Theano*, no se produce la pérdida que tenía lugar al combinar *Keras* y *Tensorflow*.

La siguiente comparativa (Figura 4.7) muestra los resultados obtenidos utilizando *Theano* y *Keras* con el *backend* de *Theano*.

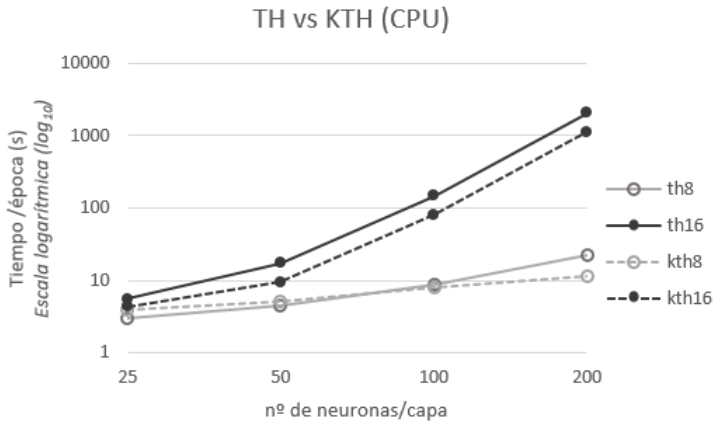

Figura 4.7: Tiempos de ejecución de una red MLP con *Theano* y *Keras* (*backend Theano*) (ejecutado en la CPU)

En esta comparativa ocurre lo mismo que sucedía en la realizada sobre la GPU (Figura 4.3): implementar la red en *Keras* (*backend Theano*) da lugar a tiempos de ejecución menores que los obtenidos utilizando *Theano* directamente.

Finalmente, la figura 4.8 muestra una comparativa similar entre las ejecuciones realizadas utilizando *Tensorflow* y *Keras* con el *backend* de *Tensorflow*.

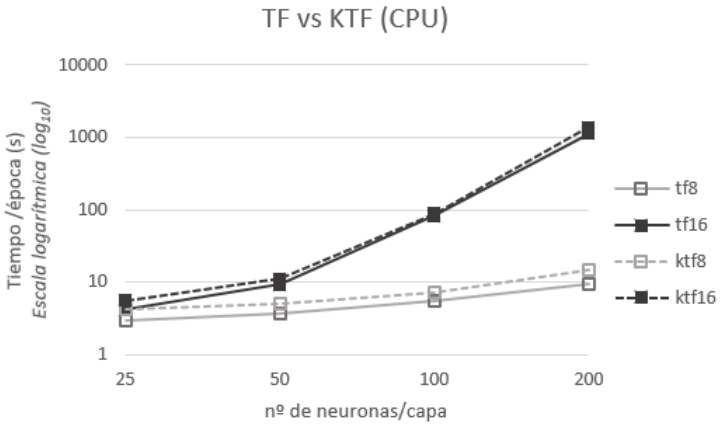

Figura 4.8: Tiempos de ejecución de una red MLP con *Tensorflow* y *Keras* (*backend Tensorflow*) (ejecutado en la CPU)

Ejecutando en la CPU los tiempos obtenidos son muy parecidos tanto utilizando *Keras* como implementado directamente en *Tensorflow*. Esto contrasta con lo que sucedía al ejecutar sobre la GPU (Figura 4.4) donde se producía una pérdida significativa de rendimiento.

4.2.2.3. Comparación entre GPU y CPU

En los dos apartados anteriores se han mostrado una serie de comparativas de diversos casos de prueba ejecutados sobre la GPU y la CPU (Secciones 4.2.2.1 y 4.2.2.2 respectivamente).

La tabla 4.7 muestra de forma compacta los *Speedup* obtenidos al utilizar la GPU en vez de la CPU para el entrenamiento de redes MLP de 8 y 16 capas.

	Dimensiones			
Tipo de red	**25**	**50**	**100**	**200**
th8	3,02	4,45	8,73	22,47
th16	5,57	17,43	147,89	2045,85
tf8	2,89	3,64	5,40	9,34
tf16	4,13	9,21	82,18	1127,96
kth8	4,00	5,08	7,92	11,47
kth16	4,33	9,38	80,13	1139,80
ktf8	4,30	4,95	7,05	14,87
ktf16	5,55	10,97	88,45	1381,30

Tabla 4.7: *Speedups* obtenidos con el uso de la GPU en redes MLP

En todos lo casos que se presentan en esta tabla el uso de la GPU resulta en un mejor rendimiento. Esta mejoría es mayor a medida que aumenta la dimensión de la red, y se aprecia especialmente cuando se incrementa el número de capas que forman la red neuronal.

Finalmente, mencionar que en el caso de redes MLP de dimensiones pequeñas (entre 2 y 4 capas con menos de 100 neuronas en cada una de ellas) el uso de la GPU puede producir *Speedups* inferiores a 1. Estos casos no se han tenido en consideración, ya que no son representativos del tipo de redes neuronales que se suelen utilizar en el mundo real (de grandes dimensiones).

4.2.2.4. Ejecución en CPU con *multithreading*

En la sección 4.2.2.2 se ha hecho un estudio del rendimiento utilizando un único *thread* de la CPU. El objetivo de esta sección es complementar estos datos analizando la manera en la que varía el rendimiento al utilizar *threads* adicionales.

Estas pruebas se han realizado únicamente para redes MLP de 8 capas. Además, se ha excluido de estas pruebas la librería *Tensorflow* por no tener soporte para *OpenMP*.

En la figuras 4.9 y 4.10 se muestran los resultados obtenidos con *Theano* y *Keras* respectivamente.

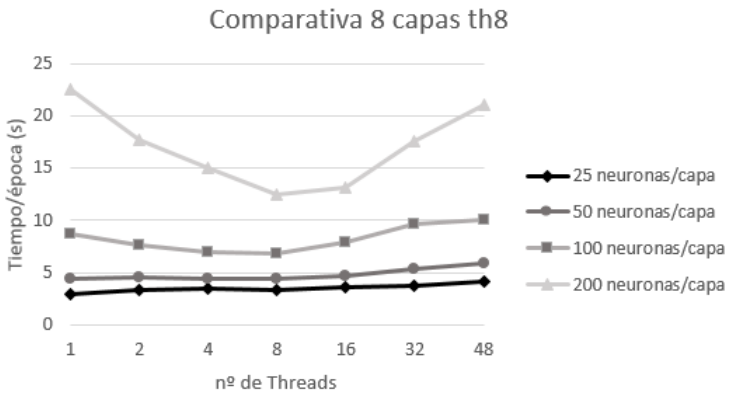

Figura 4.9: Tiempos de ejecución de una red MLP con *Theano* utilizando distintos números de *threads*

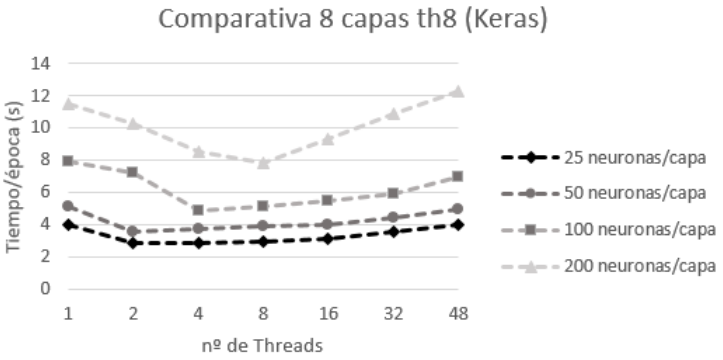

Figura 4.10: Tiempos de ejecución de una red MLP con *Keras* (*backend Theano*) utilizando distintos números de *threads*

En el caso de las redes de menor dimensión (menos de 100 neuronas por capa) el uso de *threads* adicionales no proporciona ninguna mejoría reseñable.

Cuando el número de neuronas por capa es mayor a 100 el tiempo de ejecución disminuye progresivamente hasta alcanzar los 8 *threads* (donde se consigue un *Speedup* de 1,81 con una eficiencia del 23 %). Sin embargo, una vez superado este número el rendimiento decae.

Esta pérdida de rendimiento podría ser debida a que se produce compartición falsa de memoria caché entre los diferentes *threads*.

La valoración general de estas pruebas es que el uso de múltiples *threads* no resulta provechoso para redes MLP, ya que la eficiencia obtenida es muy baja.

4.2.3. Pruebas realizadas sobre redes CNN

En esta sección se presenta un estudio del rendimiento para redes CNN similar al realizado para redes MLP. Como las pruebas realizas son similares, se ha tratado que los comentarios que acompañan a las gráficas sean también más concisos (para agilizar la lectura).

4.2.3.1. Estudio del rendimiento ejecutando en la GPU

La figura 4.11 muestra los tiempos de ejecución obtenidos durante la fase de entrenamiento de una red CNN con *Theano* y *Tensorflow* (ejecutando en la GPU).

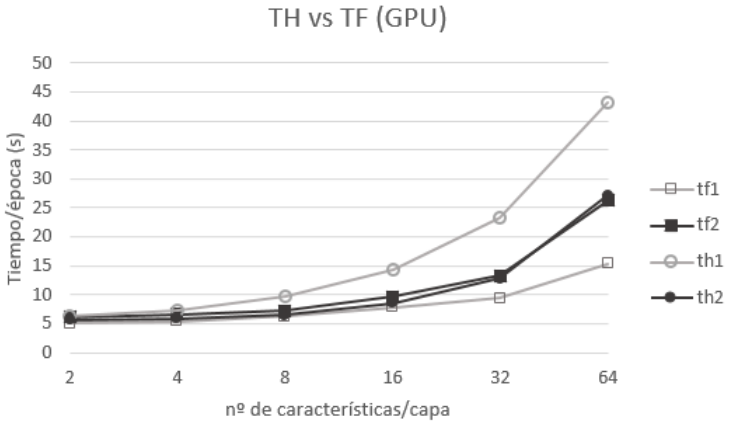

Figura 4.11: Tiempos de ejecución de una red CNN con *Theano* y *Tensorflow* (ejecutado en la GPU)

Cuando la red CNN está compuesta por dos capas ambas librerías rinden de forma similar. Sin embargo, en las pruebas realizadas con una única capa ocurre algo incoherente:

- Los tiempos de ejecución obtenidos utilizando una capa con *Theano* son superiores a los obtenidos al ejecutar las mismas pruebas utilizando dos capas. Este comportamiento no es el esperado ya que, al ser una red de menor dimensión, el coste

computacional del entrenamiento también debería disminuir (tal y como ocurre en el caso de *Tensorflow*).

La primera hipótesis que se ha barajado al observar estos resultados es que pudiese haber una incorrección en el programa de pruebas. Sin embargo, el resto de pruebas realizadas sobre *Theano* no presentan esta anomalía.

Tampoco se ha detectado ninguna errata durante la revisión del código, por lo que es posible que la causa pueda estar ligada a la propia librería.

En la figura 4.12 se muestran los tiempos de ejecución por época obtenidos durante la fase de entrenamiento de una red CNN utilizando dos posibles *backends* de *Keras* (ejecutado en la GPU).

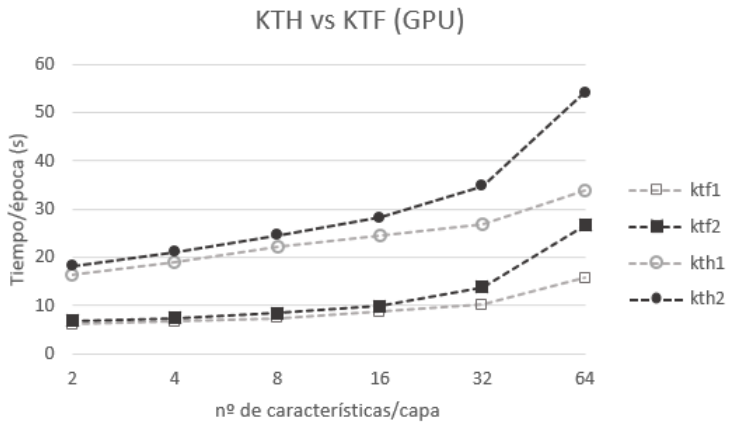

Figura 4.12: Tiempos de ejecución de una red CNN con *Keras* (*backend Theano*) y *Keras* (*backend Tensorflow*) (ejecutado en la GPU)

Los resultados de estas pruebas son opuestos a los obtenidos en el caso de redes MLP. Al utilizar *Keras* para entrenar una red MLP se observaba que el rendimiento era mejor utilizando el *backend* de *Theano*.

Sin embargo, en el caso de las redes CNN las pruebas realizadas con el *backend* de *Tensorflow* se ejecutan más rápido. Por ejemplo, utilizando 64 mapas de características por capa *Keras* con *Tensorflow* rinde aproximadamente el doble de rápido que *Keras* con *Theano*.

Esta ventaja no es inesperada si se tiene en cuenta que *Tensorflow* es una librería desarrollada por *Google*. Concretamente, las redes CNN son ampliamente utilizadas en el campo del reconocimiento de imágenes, en el que *Google* invierte muchos recursos para conseguir redes neuronales eficientes.

Estas optimizaciones están implementadas en la versión *Core* de *Tensorflow*, que es la que utiliza *Keras* [Thomas, 2017].

La siguiente comparativa (Figura 4.13) presenta los tiempos de ejecución por época obtenidos utilizando *Theano* y *Keras* con *backend* de *Theano* (ejecutado en la GPU).

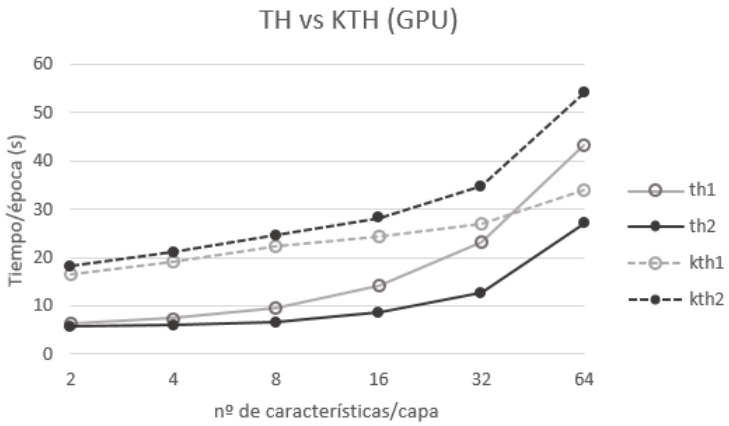

Figura 4.13: Tiempos de ejecución de una red CNN con *Theano* y *Keras* (*backend Theano*) (ejecutado en la GPU)

Ignorando el caso anómalo (red CNN de una capa ejecutado en *Theano*), se observa que utilizar *Keras* supone un perjuicio al rendimiento en comparación a utilizar *Theano* directamente.

Finalmente, en la figura 4.14 se muestra una comparativa similar entre *Tensorflow* y *Keras* con *backend* de *Tensorflow*.

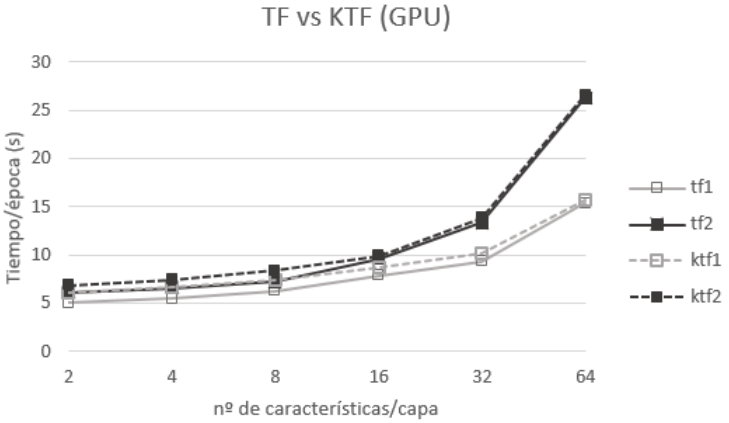

Figura 4.14: Tiempos de ejecución de una red CNN con *Tensorflow* y *Keras* (*backend Tensorflow* (ejecutado en la GPU)

En este caso, la diferencia entre el rendimiento de cada una de las librerías es insignificante.

4.2.3.2. Estudio del rendimiento ejecutando en la CPU

Al igual que en las redes MLP, también se han realizado pruebas de redes CNN ejecutando sobre la CPU (utilizando un único *thread*).

En la figura 4.15 se muestran los tiempos de ejecución por época obtenidos durante la fase de entrenamiento de una red MLP utilizando *Theano* y *Tensorflow*.

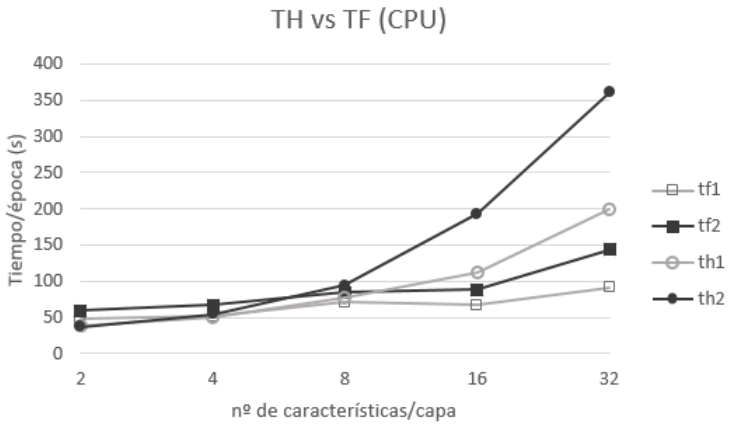

Figura 4.15: Tiempos de ejecución de una red CNN con *Theano* y *Tensorflow* (ejecutado en la CPU)

Cuando el número de mapas de características es mayor a 8, *Tensorflow* obtiene mejor rendimiento que *Theano* (ejecutándolos sobre la CPU). En caso contrario, hay situaciones en las que *Theano* rinde ligeramente mejor, pero la diferencia es poco relevante.

En el siguiente gráfico (Figura 4.16) se muestran los tiempos de ejecución por época obtenidos durante la fase de entrenamiento de una red MLP utilizando dos *backends* distintos de *Keras* (*Theano* y *Tensorflow*).

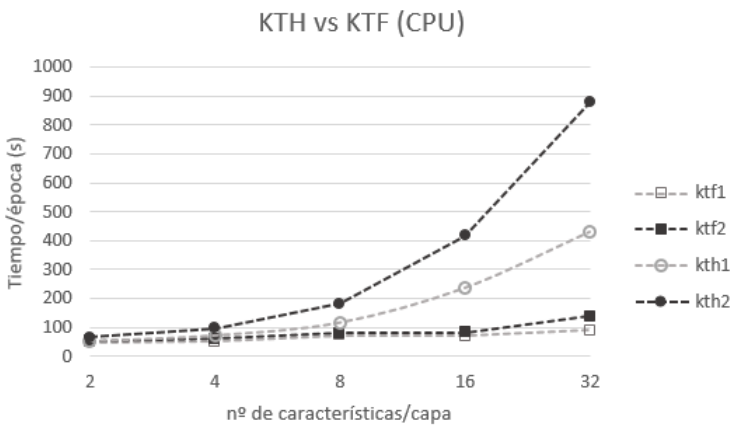

Figura 4.16: Tiempos de ejecución de una red CNN con *Keras* (*backend Theano*) y *Keras* (*backend Tensorflow*) (ejecutado en la CPU)

Al igual que las ejecuciones realizadas sobre la GPU (Figura 4.12), *Keras* obtiene mejor rendimiento cuando el *backend* es *Tensorflow*.

La siguiente comparativa (Figura 4.17) muestra los resultados obtenidos utilizando *Theano* y *Keras* con el *backend* de *Theano*.

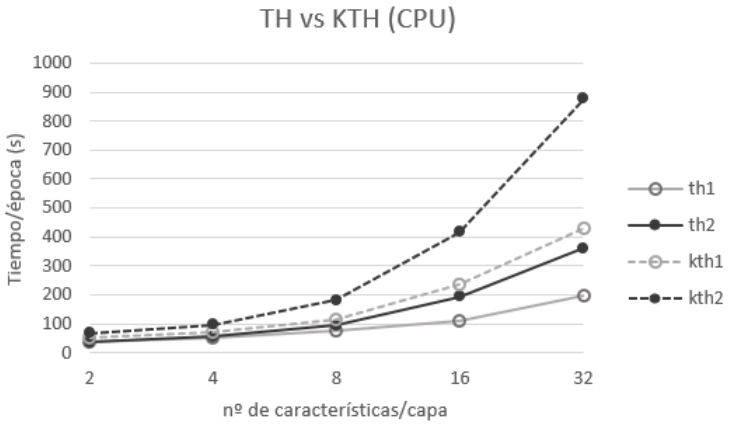

Figura 4.17: Tiempos de ejecución de una red CNN con *Theano* y *Keras* (*backend Theano*) (ejecutado en la CPU)

En esta comparativa se da la misma situación que se daba al ejecutar sobre la GPU (Figura 4.13): *Keras* sobre *Theano* no realiza las optimizaciones adecuadas y añade una carga de trabajo que repercute en el tiempo de ejecución.

La última prueba ejecutada sobre la CPU compara el rendimiento de *Tensorflow* y *Keras* con *backend* de *Tensorflow* (Figura 4.18).

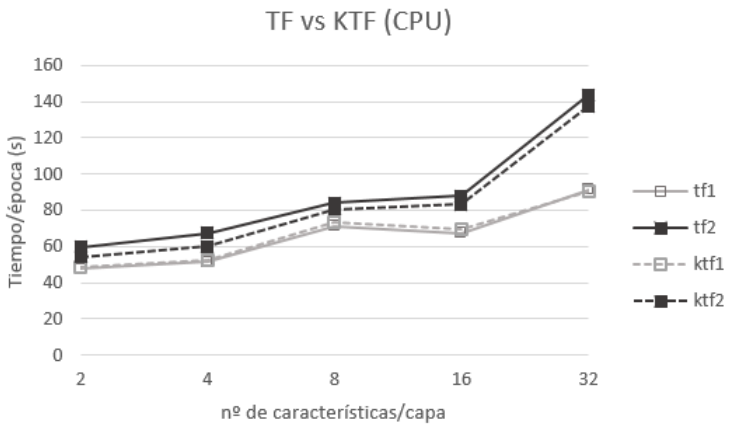

Figura 4.18: Tiempos de ejecución de una red CNN con *Tensorflow* y *Keras* (*backend Tensorflow* (ejecutado en la CPU)

En esta prueba ambas librerías rinden de forma similar, tal y como sucedía ejecutando sobre la GPU (Figura 4.14).

Otro aspecto reseñable de esta figura es que, al aumentar el número de mapas de características de 8 a 16, se produce una subida muy leve (o incluso una bajada) del tiempo de ejecución. Es posible que esto sea debido a la aplicación de distintas optimizaciones y rutinas en base a ese parámetro, y sea con 16 cuando se hace el cambio.

4.2.3.3. Comparación entre GPU y CPU

	Dimensiones				
Tipo de red	**2**	**4**	**8**	**16**	**32**
th1	6,26	6,93	7,99	7,87	8,53
th2	6,59	9,25	14,41	22,56	28,34
tf1	9,45	9,43	11,39	8,59	9,77
tf2	9,76	10,21	11,62	9,12	10,78
kth1	3,23	3,82	5,25	9,69	15,94
kth2	3,69	4,57	7,43	14,77	25,26
ktf1	7,90	7,87	9,79	8,02	8,84
ktf2	7,92	8,06	9,60	8,39	9,93

Tabla 4.8: *Speedups* obtenidos con el uso de la GPU en redes CNN

En todos los casos que se muestran en la tabla 4.8 el uso de la GPU conlleva un *Speedup*

significativo. Del mismo modo que sucedía en las redes MLP (Tabla 4.7) el factor de aceleración es mayor para las redes de mayor tamaño.

4.2.3.4. Ejecución en CPU con *multithreading*

Las últimas pruebas de este capítulo han consistido en ejecutar los programas de prueba para redes CNN utilizando varios *threads* de la CPU.

En la figura 4.19 se representa gráficamente los tiempos de ejecución obtenidos utilizando *Theano* (CNN de 1 o 2 capas), mientras que en la figura 4.20 se muestra lo mismo utilizando *Keras* (*backend Theano*).

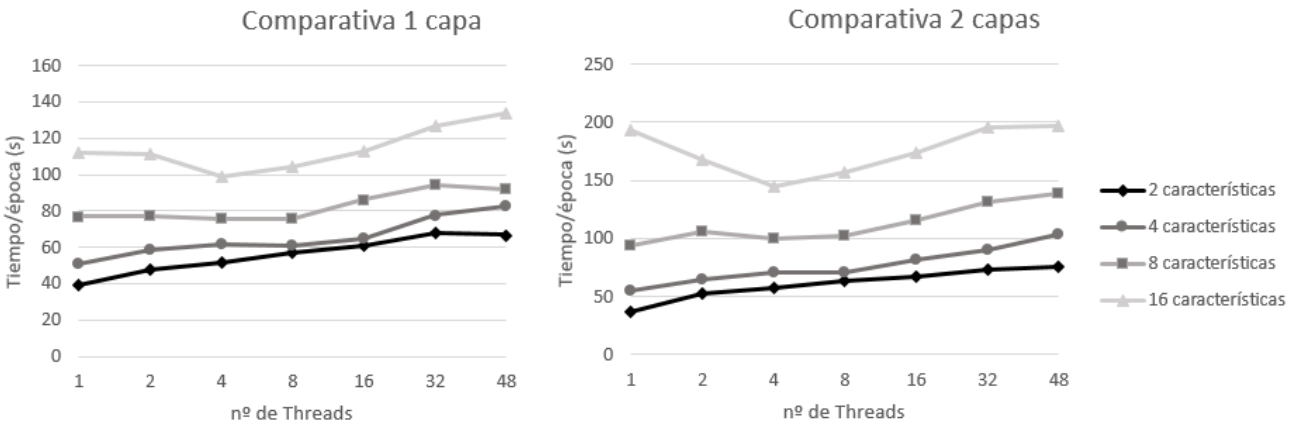

Figura 4.19: Tiempos de ejecución de una red CNN con *Theano* utilizando distintos números de *threads*

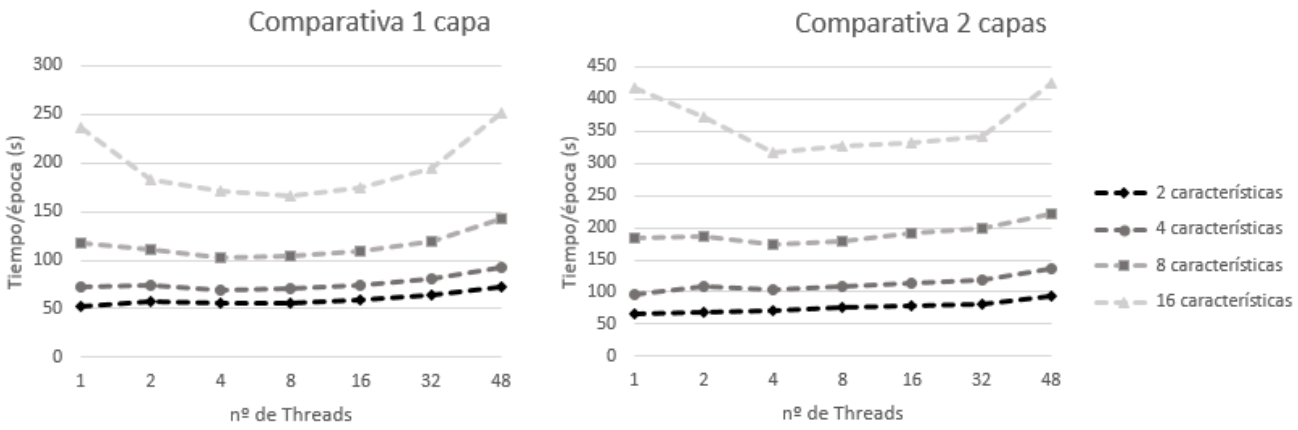

Figura 4.20: Tiempos de ejecución de una red CNN con *Keras* (*backend Theano*) utilizando distintos números de *threads*

El comportamiento que se observa en estas figuras varía en función del número de mapas de características de la red CNN:

- Entre 2 y 4, aumentar el número de *threads* supone un aumento del tiempo de ejecución.
- Entre 8 y 16, se produce una mejora del rendimiento hasta alcanzar los 4 *threads*. Una vez superado esta cantidad, el uso de *threads* adicionales conlleva mayores tiempos de ejecución.

  Este comportamiento es el mismo que el observado en las redes MLP (Sección 4.2.2.4), por lo que es probable que las causas sean las mismas (compartición falsa en memoria caché).

Del mismo modo que sucedía en las redes MLP, el factor de aceleración obtenido está muy alejado del número de *threads*, y la eficiencia va en declive a medida que este número aumenta.

5. CAPÍTULO

Conclusiones

Este capítulo presenta, a modo de cierre, un resumen de los resultados obtenidos en la fase de experimentación y varias propuestas para proyectos futuros.

5.1. Valoración de los resultados

Los resultados obtenidos en la fase de experimentación permiten establecer un criterio para el uso de estas herramientas. Este criterio se presenta en la tabla 5.1.

Situación	Recomendación
MLP (GPU)	Tensorflow
MLP (CPU)	Tensorflow / Keras (*backend Theano*)
CNN (GPU)	Tensorflow / Keras (*backend Tensorflow*)
CNN (CPU)	Tensorflow / Keras (*backend Tensorflow*)

Tabla 5.1: Recomendación de las librerías en función del caso de uso

Las recomendaciones que se muestran en esta tabla se establecen en base a los resultados de las redes de mayor dimensión, ya que son las más representativas del tipo de redes que se utilizan en aplicaciones reales. Dicho esto, también hay que tener en cuenta que estas recomendaciones están basadas en la experimentación realizada, con las limitaciones que esto pueda implicar a su nivel de generalidad.

La primera conclusión que se extrae de esta tabla es que *Tensorflow* es una opción adecuada tanto para redes MLP como CNN (independientemente de si se utiliza la GPU o la CPU).

La segunda es que *Keras* es capaz de alcanzar un rendimiento similar a *Tensorflow* en la mayoría de situaciones. La recomendación en los casos de empate es utilizar *Keras*, ya que se trata de una librería mucho más sencilla de utilizar.

Los resultados experimentales también han servido para comprobar y cuantificar las ventajas de utilizar una GPU. El uso de una tarjeta gráfica cobra especial relevancia cuando las redes son de dimensión elevada, ya que el uso de la CPU da lugar a tiempos de ejecución inasumibles.

Finalmente, de las pruebas realizadas en la CPU con *multithreading* se puede concluir que esta opción no aporta grandes beneficios.

5.2. Líneas de trabajo futuras

El trabajo realizado en este proyecto ha satisfecho los objetivos propuestos al comienzo del mismo. Sin embargo, está abierto a otros trabajos que permitan complementar y justificar de manera más detallada los resultados obtenidos. Concretamente, las tareas que se podrían desarrollar son las siguientes:

1. Utilizar herramientas de *profiling* para estudiar y analizar de manera más detallada las optimizaciones que realiza cada una de las librerías.
2. Realizar pruebas con redes que utilicen otro tipo de arquitecturas.
3. Analizar cómo afecta al rendimiento el uso de diferentes base de datos.
4. Experimentar en máquinas con múltiples GPUs y *clusters*.

Bibliografía

[Red, 2001] (2001). Redes neuronales: Conceptos básicos y aplicaciones. https://www.frro.utn.edu.ar/repositorio/catedras/quimica/5_anio/orientadora1/monograias/matich-redesneuronales.pdf. Accessed: 2017-07-14.

[AMD, 2017] (2017). Amd opteron 6168 specifications. http://www.cpu-world.com/CPUs/K10/AMD-Opteron%206168%20-%20OS6168WKTCEGO%20(OS6168WKTCEGOWOF).html. (Acceso: 2017-07-04).

[Sof, 2017] (2017). Deep learning software. http://deeplearning.net/software_links/. Accessed: 2017-07-03.

[Abadi et al., 2015] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852.

[Krizhevsky et al., 2009] Krizhevsky, A., Nair, V., and Hinton, G. (2009). The CIFAR-10 dataset. http://www.cs.utoronto.ca/~kriz/cifar.html. (Acceso: 2017-07-14).

[LeCun et al., 1998a] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998a). Gradient-based learning applied to document recognition.

[LeCun et al., 1998b] LeCun, Y., Cortes, C., and Burges, C. J. (1998b). The mnist database. http://yann.lecun.com/exdb/mnist/. (Acceso: 2017-07-14).

[Lichman, 2013] Lichman, M. (2013). UCI machine learning repository.

[Nielsen, 2017] Nielsen, M. (2017). Neural Networks and Deep Learning. http://neuralnetworksanddeeplearning.com/. (Acceso: 2017-07-04).

[NVIDIA, 2017] NVIDIA (2017). NVIDIA cuDNN. https://developer.nvidia.com/cudnn. (Acceso: 2017-07-04).

[NVIDIA, 2017] NVIDIA (2017). Tarjeta gráfica GeForce GTX TITAN Black. http://www.nvidia.es/object/geforce-gtx-titan-black-es.html. (Acceso: 2017-07-04).

[Reyes et al., 2016] Reyes, M. A. V., Márquez, C. Y., and Fernández, L. P. S. (2016). Algoritmo backpropagation para redes neuronales: conceptos y aplicaciones.

[Team, 2017] Team, T. D. (2017). Theano. http://deeplearning.net/software/theano/. (Acceso: 2017-07-12).

[Thomas, 2017] Thomas, R. (2017). Big deep learning news: Google tensorflow chooses keras. http://www.fast.ai/2017/01/03/keras/. (Acceso: 2017-07-26).